# Parallelism in Contextual Processing

Paul Stephen Rehfuss
B.A., Reed College, 1975
M.A. University of Oregon, 1981

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

April 1999

The dissertation "Parallelism in Contextual Processing" by Steve Rehfuss has been examined and approved by the following Examination Committee:

Dan Hammerstrom
Professor
Thesis Research Adviser

Todd Leen
Associate Professor

Pieter Vermeulen
Associate Professor

Nelson Morgan
Professor
ICSI

# Acknowledgements

Thanks to Barak Pearlmutter for telling me about Rockoff's work, to T. Rockoff for encouragement when it was needed, and to Dan Hammerstrom, Todd Leen, John Moody, and Emilie Kroen for all their support and encouragement. Thanks to Ellen and the entire CSE department for getting fed up and making me finish this.

# List of Tables

# List of Figures

# Abstract


**Parallelism in Contextual Processing**

**Paul Stephen Rehfuss**

**Supervising Professor: Dan Hammerstrom**

In this dissertation, I study the use of *context* in sensory processing, and specifically, cost-effective parallel implementations of contextual processing. Taking contextual information to be represented in the form of discrete, compact *models*, application of contextual knowledge then occurs as models are *matched* to input. Model-matching occurs at the interface between bottom-up classification and feature extraction and top-down modeling and interpretation. Bottom-up classification and feature extraction can be well supported by cost-effective parallel hardware. The central thrust of this dissertation is to show how such parallel hardware can be inexpensively modified to support model-matching, thus extending its range of applicability.

For contextual processing of ordered input, I derive "Higher Order Viterbi Search (HOVS)", a Markov approximation to Viterbi search using higher-order source models. Simulations show HOVS captures most of the benefit of using higher order source models, while being more time and space efficient. I give an SIMD implementation of HOVS, and discuss some restrictions on the source model required for a practical implementation.

From analysis of algorithm requirements and VLSI trends, with area as cost measure, I derive a cost-performance model for on-chip parallelism. I conclude that for the applications considered, *there are only two viable architectural alternatives*: if the required system memory fits on-chip, using many simple processors may be preferred. Otherwise, off-chip bandwidth limitations imply an architecture of a few complex processors.

I introduce the *SFMD* class of parallel architectures, extending SIMD processing to better handle the irregular, data-dependent computation typical of contextual processing. SFMD extends SIMD processing by giving each processing element separate control within small loop bodies. The extra processor complexity is modest. To preserve SIMD semantics and programming simplicity, interprocessor communications may complete only after all processors have synchronized at a barrier.

When area cost is not considered, SFMD is outperformed by an SPMD architecture on tasks with sparse communication and highly varying computation times. When communication is not too sparse, the ability of SFMD to allow more processors on a chip may compensate. Variance reduction techniques may also decrease the performance gap.

# Chapter 1

# Introduction

*"context: The whole structure of a connected passage regarded in its bearing upon any of the parts which constitute it" ... Oxford English Dictionary*

The overall topic of this dissertation is cost-effective hardware for the recognition of objects from sensory data, for such tasks as speech recognition, handwriting recognition, and image understanding. The specific problems addressed are how to perform *contextual* processing on current cost-effective parallel hardware designs for sensory processing, and how to extend these designs to better support contextual processing.

Sensory processing involves multiple stages of processing, moving from the data itself to "higher level" constructs of a more symbolic nature, such as individual pixels to a set of strokes comprising an "R". Processing in the stages closest to data may depend only on the data itself, but at some later stage, contextual effects come into play. Hardware suitable for the earliest stages of processing, those closest to data, already exists [Ham90, AJ97, WAK+96] but, I will argue, processing that involves context can make use of additional capabilities not found in these designs. Specifically, contextual processing will benefit from better support for *data-dependent control flow.* Analysis of current technology trends indicates that the needed support can be provided for a reasonable cost, around one percent of the area of a modest sized chip.

Examination of algorithms with data-dependent control flow, both for sensory processing, such as Interpretation Tree Search [Gri90], and of a more general nature, such as sparse matrix-vector multiplication, shows a substantial benefit from the additional support in many cases, frequently giving a performance boost by a factor of 1.5 - 2 over an otherwise equivalent design using current techniques.

In order to elaborate on the suggested solution, I must formalize contextual processing, determine the computational requirements of such processing, and see how to cost-effectively extend existing parallel hardware to support such processing. So what is context and how is it used in recognition?

### 1.0.1 Example

As an example of the kind of tasks of interest, consider figure 1.1, illustrating the process of recognizing a hand printed "R". Data (pixels) are organized into *features* (strokes) via edge recognition, construction of connected components, and so on. These operations take place *bottom-up*, the assignment of a given pixel to a feature object involving only operations on pixel values in a local neighborhood of that pixel. Features such as strokes have *attributes*, such as direction, length and curvature. Features are *grouped* into sets hypothesized to belong to a single superobject (corresponding, say, to an "R", or some other letter). This may be driven bottom-up, by grouping, say, adjacent features, or *top-down*, by using information contained in a *model* of the superobject. While they may be hierarchical, in general, at some level, models correspond to symbolic *interpretations* of the data (e.g., "R"). *Structural* models, as shown in the example, have a number of parts or *components*; in the illustrative model for "R" shown in the figure, these are "stem", "bowl" and "descender". When the model is related to the data, each of these components is matched to some feature (or submodel). Components have attributes (slots) that correspond to attributes of the features to which they are matched. For example, the "stem" component might have a "length" attribute, giving the length of the matching stoke feature. Models contain information about the inter-relationships of components and about the values of components' attributes, in the form of *constraints* on allowable values, for example, that "descender" is angled at about 45 degrees with respect to "stem". Models can be hierarchical, so that components can correspond either to sub-models or to features.

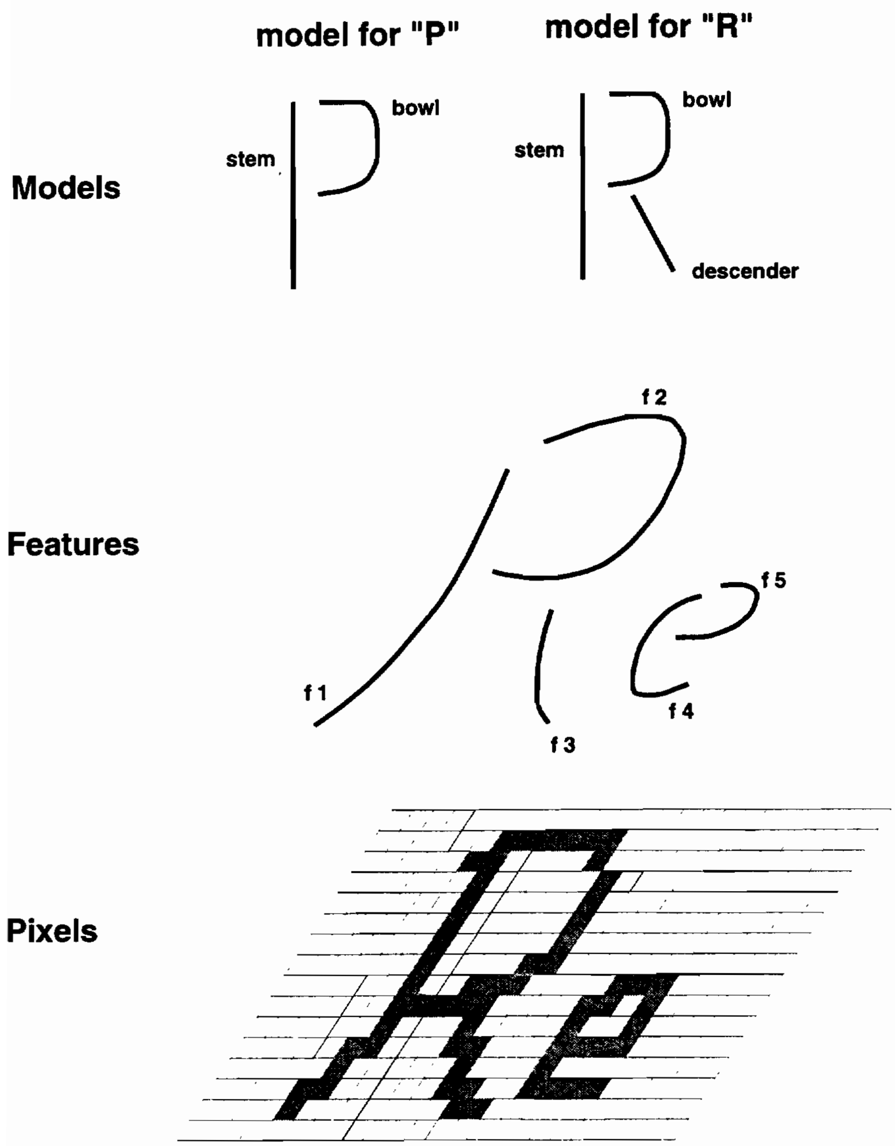


Figure 1.1: Recognizing an "R".

The process of putting a group of features into correspondence with the components of a model, and checking feature attributes against model constraints is referred to as *model matching.* As features are put into correspondence with components, attributes of the features determine attributes of the components and the model, thus (partially) "filling in" or *instantiating* the model by setting or constraining some of its parameters. These other, hypothesized, features can then be sought in an *associative lookup* process, finding features based on their location and on other attributes, such as angle of the stroke from vertical. When features corresponding to those suggested by the model are found, other inter-feature relationships and constraints specified by the model can be checked to *verify* the interpretation. This may again involve finding features by associative lookup.

With respect to figure 1.1, this top-down model-matching might proceed as follows: when a given feature is hypothesized to belong to an "R", a model for "R" is be used to make predictions about the relative location and shape of other features belonging to that "R". Perhaps the "stem" stroke is processed first: a stroke feature is hypothesized to correspond to the "stem" component of an "R". Information in the model for "R" then might suggest ranges of values for the size, orientation and location attributes of the "bowl" so that the top ends of the "bowl" and "stem" are close, and the lower end of the "bowl" is near the midpoint of the "stem". A stroke having these properties is then sought. If a satisfactory candidate is found, the meeting point of the "bowl" and "stem" stroke then suggests ranges for attributes of the "descender", and so on.

Of course, there are likely to be competing hypotheses about the grouping and interpretation of a set of features: "does this stroke come from an 'R' or a 'P' ? " Here further context can come into play from higher level word and language models: perhaps the word "Rent" makes more sense in the current passage than does "Pent", and so "R" is a more likely interpretation than "P". Thus the matching process may be hierarchical. Finally, note that there are models without a geometric character, for example, *Hidden Markov Models (HMMs)* and pronunciation models in speech recognition.

### 1.0.2 Recognition: the interplay of data and models

In the view exemplified above, recognition is an interplay between data and interpretations, mediated by the constraints contained in models. Bottom-up, recognition involves extraction of (possibly hierarchical) features from the data. Top-down, it is a process of hypothesis construction, refinement (instantiation) and verification, driven by current (hypothetical, partial) interpretations, including prior knowledge, and constraints imposed by a (hypothesized) model. In the interplay between bottom-up and top-down processing, data may suggest groupings and possible models, while hypothesized models may suggest both possible groupings of existing features, and also other features to be sought. Based on this view, *context* can be formalized as the implications of a partially instantiated model about the values of the model's components' attributes, these implications being derived from the model's constraints. Thus both data and existing interpretations give rise to an hypothesis about a particular model being in correspondence with that data, the data also refining or partially instantiating the model. This partially instantiated model then is the context within which further interpretation of data occurs.

#### Features

Some examples of feature extraction techniques are: one- and two-dimensional convolutional filtering, extraction of spectral coefficients, thinning and edge finding, and construction of hyperpixels [GW92]. From these, we see that features are generally constructed from data in a *neighborhood* that is local in time or space. Also, to construct a given feature, typically the same processing is done on each neighborhood, and independently of other neighborhoods, to that features for different neighborhoods can be constructed in parallel. The main point, though, is that feature extraction proceeds bottom-up from the data, *independently of any interpretation* of that data, and so independently of any context.

### Models

Models represent hypotheses about the interpretation and generation of data, for example, "this set of pixels comes from someone writing an 'R"'. In this dissertation, I am concerned with *structural* models, that describe an object in terms of parts or *components*, and in terms of relationships between attributes of its components, or *constraints*. For example, the model for "R" shown in figure 1.1 has three components, describing strokes for the bowl, stem and descender, and (not shown) would also specify constraints about the relative size and position of the three components.

I am interested in structural models for several reasons: structural models decompose the relations between objects into manageable and efficiently computable pieces, structural models are commonly used both in speech and vision recognition, and, most importantly, structural models embody the notion of context.

Structural models provide a top-down conditioning influence on the interpretation of the data corresponding to their components. A partially matched model can have implications for the existence, class, temporal or geometric location, or other attributes of its unmatched components. The model may thus influence the grouping of components; it may be used to improve the classification of components, especially when the implications are probabilistic; or, when implications involve location, the model may be used in organizing search and selecting data neighborhoods for attempting to match any unmatched components. Models used to organize search in this way are sometimes referred to as *active models* in the vision literature [BCKM90].

There are several uses for the knowledge contained in structural models: In *recognition*, models are matched to find the type (or *label*) of an object. Models may be passive, so that the algorithm directs the matching process, or active, where the model contains information to direct the search, such as a search order for components In *correction*, models provide context for classification of their components. The set of all models, or *model library*, is used to generate a source model for components. A source model is useful when the library is incomplete (e.g., when novel words may be encountered in text or speech recognition). In *grouping* or *segmentation*, one determines which features belong to the same object. Segmentation happens more or less implicitly in the matching process. Object boundaries may be derived explicitly from the data as certain kinds of features, *feature-based segmentation*, or may be determined implicitly as the matching process groups features as parts of the same object, *recognition-based segmentation*. Finally, in *registration*, once an object is matched to a model, the model may be used to find various parts of the object. I will look only at recognition and correction and grouping, viewing registration as involving only a single model.

As processes, model matching is distinguished from simple labeling or classification. The latter just gives a probability distribution over model labels (classes), it does not use implications about components for any purpose, and hence it does not involve contextual processing. As an example, consider classification by nearest-neighbor feature vector matching to a set of classes *prototypes*, for example, when the features are pixel values, template (correlation) matching of images. Prototypes are not usually thought of as models in the sense of this thesis: the result of matching is simply the winning prototype or its class label, and any implications for the values of components is ignored. One could, however, get something like our notion of model matching behavior from prototypes. Consider the set of vectors, $S_P$, "belonging" to a given prototype, $P$, i.e., the vectors which are closest to that prototype. This set of vectors defines a joint density, $D_P$, over the values of the vector elements. Each prototype corresponds to a model, $M_P$: the models have identical structures, with one component for each dimension of the space. Each component has a single attribute, its value. The models, $M_P$, differ in having different joint densities, $D_P$, over the values of their components' attributes (i.e., over the values of the elements of vectors in $S_P$). With this setup, one could get model matching behavior, for example, by projecting a vector, $v$, onto the subspace of components with "adequately determined" values, find the nearest (projected) prototype, $P$, and using the probability model, $D_P$, associated with the prototype's class to estimate or correct the values of the inadequately determined components of $v$.

Model matching and grouping are ubiquitous processes that occur at the meeting place of top-down and bottom-up sensory processing. Models modularize *a priori* knowledge and its top-down application. It is possible that, even with a "cortical" processing model, involving activation within a uniform matrix of neuron-like elements, models may reappear as recurring patterns of activation, and computational support for simulation of the cortical model may also need to support model-matching and grouping kinds of tasks. The conclusion is that support for grouping and model matching is of great importance for machines for sensory processing.

### 1.0.3 Computation

Feature extraction typically involves *regular* computation: the control flow for processing each data element is the same. Feature extraction occurs bottom-up, so any "context" used is other data, not (partial, data-dependent) interpretations. Furthermore, which other data elements (if any) to use is known in advance: generally these are adjacent data in one or more dimensions. Some examples of features involving adjacent data are differences or rates of change, filters computed over neighborhoods, and hyperpixels. These examples, and feature extraction in general, involve performing the same computation for each data element, over data whose location is typically known in advance and can be incorporated into the code for the algorithm. Thus, in feature extraction, the control flow for processing each data element is typically the same, and the computation is regular. A second point is that the processing for feature extraction can frequently be performed independently on each data element.

Conversely, model matching typically involves data-dependent control flow. At a given point in processing, only some of the components of the model may have been matched to the data. The model's relations between these (hypothesized) components, and yet unmatched ones, may suggest features to be sought and processed. Thus the flow of processing of the model depends on the results of processing and matching so far, and on the structure of the model itself, and hence is data-dependent. Further, the features are sought according to values of their attributes, in an *associative lookup* process, typically involving indexes of various kinds, and traversing such data structures also involves data-dependent control flow. In the processing illustrated in figure 1.1, after, say, the stem stroke is hypothesized to be part of an "R", the next step might be to search for the descender, by seeking a stroke with approximately the right location, length and angle. Satisfying such queries for objects with features in some specified range typically involves traversal of a tree-based index. Thus an important difference between computation in the top-down, model-matching part of recognition, and computation in the bottom-up part, is data-dependent control flow. However, like bottom-up feature extraction, the matching of one model to a group of features typically proceeds independently of the matching of other models to that group. Model-matching thus involves multiple independent *irregular* computations, each with control flow determined by its model, the data it has seen so far, and the interpretations that have been made of that data.

I have noted that features extracted from different neighborhoods can typically be constructed independently of one another. Similarly, matching different models to the same group of features is also independent (prior to any intermodel competition). As well, associative lookup of different features, or the same feature by different attributes (in different indexes) occurs independently. Thus feature extraction, associative lookup, and model matching are potentially parallelizable.

The fundamental computational requirements of the feature extraction, grouping, and model matching stages of sensory processing must be isolated, in order to design machines for sensory processing that are as simple and general purpose as possible. The preceding discussion has highlighted the fact that executing the code for a given processing task is typically performed many times. Extracting a given feature from a neighborhood is done for many neighborhoods, looking up a feature by its attributes can be done independently for all indexes used in the lookup, and matching models to groups of features can be done for many models and many groups. So a large amount of computation is needed. Furthermore, the multiple computations are independent, and so parallel execution is a natural way of meeting the computational demand.

However, while extracting a feature typically proceeds with predetermined flow of control through the code for the task, both associative lookup and model matching involve data-dependent execution. Current parallel architectural designs for sensory processing [Ham90, AJ97, WAK$^+$96] typically are optimized for feature extraction, and hence for executing codes *without* data-dependent control flow. To extend these designs to support a wider range of sensory processing tasks, including model matching, support is needed for data-dependent execution. I therefore take *cost-effective support for parallel data-dependent execution* as my key goal in extending current parallel sensory data processing designs.

## 1.1 Parallelism

To proceed, we need some background about types of parallelism and parallel architectures.

### 1.1.1 Types of parallelism

The notion of parallelism is that of concurrent execution: different pieces of a task being performed simultaneously by distinct entities. From the entire set of instructions for a task, some can be done simultaneously: two instructions can be executed at the same time if the data used by one is unaffected by the other, and *vice versa*, that is, if the two instructions have no *dependencies.*

Several types of parallelism can be distinguished: In *inter-task parallelism*, such as pipelining or communicating sequential processes, instruction dependencies are managed by explicit communication between distinct sequential tasks. In *instruction level parallelism (ILP)*, the technique used by contemporary processors, a stream of instructions is analyzed to find non-dependencies allowing some instructions in the stream to be performed simultaneously. More specialized, but still very common, especially for sensory processing, are *data parallelism*, where a single program operates on multiple independent data partitions, and *knowledge parallelism*, where multiple pieces of knowledge (models) are applied to a single piece of data[1]. Here non-dependence of instructions is inferred from the fact that they are acting on entirely separate sets of data. Model matching algorithms are typically either data- or knowledge- parallel, or both, and these are the forms of parallelism on which I will concentrate. The utility of data parallelism for the earlier stages of sensory processing is well established. I will examine the potential for extending parallel hardware to knowledge parallelism and the model matching stage.

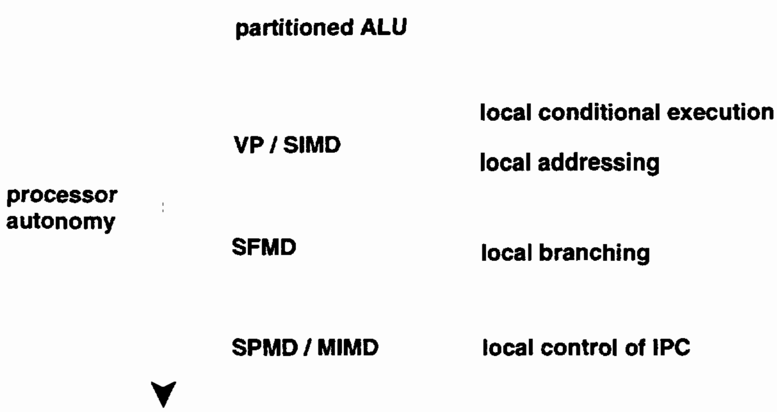


Figure 1.2: Range of parallel architectures, organized by increasing *processor autonomy*, i.e., decreasing global control over the computation. The various terms are defined in the text.

### 1.1.2 Parallelism: feature extraction, grouping and models

Let us relate these types of parallelism to the tasks discussed previously, in more detail. In feature extraction there is generally regular computation and communication, and independent and identical control flow, thus current architectures targeted at data parallelism provide cost-effective support for these computations.

For grouping, the main task is associative lookup, and potential parallelism comes from traversing multiple indexes for one or more feature. When extending a data-parallel

[1]There is a sort of "duality" between knowledge and data parallelism: if one views multiple models as instantiating multiple "algorithms", then knowledge parallelism applies multiple algorithms to single data, while data parallelism applies a single algorithm to multiple data.

architecture to handle grouping, an important point is that the data may be distributed among the multiple processors. This is the reason there may be multiple indexes to be traversed for a given lookup – there may be an index local to each processor. indexing the data of that processor. Traversal of, say, a tree-based index involves evaluating conditions at the nodes of the tree and using the result to either stop or to select the next node to examine. This implies data-dependent control flow, choosing between stopping and continuing to traverse the tree. However, traversing the indexes on separate processors can be done independently, in parallel, requiring communication only when a satisfactory solution is found. Also, the same computational kernel is executed on each processor in processing each node. So support for parallel execution of data-dependent computational kernels, as will be seen to be needed for model matching, will also benefit grouping.

There are three ways one might parallelize matching a structural model. In *model parallelism*, (some set of) entire models are matched in parallel, independently of one another. This gives a coarse-grain, "trivial" parallelism. Evaluating a model on a processor proceeds independently of the evaluation of an alternative model on another processor, and model evaluation involves data-dependent control flow based on the data seen and the model structure, so, as for grouping, what is needed is support for parallel execution of data-dependent computational kernels. Discussion of this form of parallelism, and support for it make up the bulk of this work, chapters 3, 4 and 5, and appendix B.

In *node parallelism* the individual nodes of a single model are matched or evaluated in parallel, and then the interrelations between them are computed or propagated. The irregular interrelation between nodes makes this problematic: there is a high communication to computation ratio, and likely an irregular assignment of data to processors, both of which tend to diminish the advantages of the parallel evaluation. "Clumping" multiple nodes to a single PE may convert this to something like model-parallelism, but problems of irregular communication and data assignment remain. I do not examine this form of parallelism in this work.

Finally, in *state parallelism* all nodes have the same set of states (attributes). Parallel execution occurs by evaluating states (determining attribute probabilities) in parallel. This appears an unusual case, but gives a fine-grained parallel approach to recursive, HMM-like models. I discuss this in chapter 2.

### 1.1.3 Parallel architectures

Somewhat loosely related to the varieties of parallelism listed above are the various types of parallel architectures. In *Single Instruction Multiple Data (SIMD)* architectures, multiple *processing elements (PEs)* execute a single shared stream of instructions to process individual streams of data. Each instruction is executed simultaneously on all PEs, so-called *lockstep execution.* To accommodate branching and conditional execution, instructions are distributed first for one branch of an "if" statement, and then the other branch, with PEs only performing the instructions for one branch, and being turned off or otherwise ignored for the other. SIMD architectures are optimized for data (and, dually, knowledge) parallelism, with lockstep execution providing a substantial cost savings in that instruction handling is shared between all processors, allowing the individual processors to be less complex and so less expensive. A *Vector Processor (VP)* is a kind of SIMD architecture, where the lockstep execution of a single instruction on multiple data streams is formulated as a single operation on a *vector* of data. A very simple form of vector processing is used in a number of contemporary microprocessors (e.g., [Lee95]), where, say, a 32-bit adder can be partitioned to add a pair of 32-bit words, 2 pairs of 16-bit words, or 4 pairs of 8-bit words.

*Multiple Instruction Multiple Data (MIMD)* architectures have multiple PEs executing individual streams of instructions to process individual streams of data. This embodies general inter-task parallelism, although, due to its generality, it can be used for data and knowledge parallelism as well. The typical application of MIMD architectures to data and knowledge parallelism is by *Single Program Multiple Data (SPMD)* execution, where a single *program* executes on all PEs, each with different data. Due to branching, different PEs see different instruction streams, and there is no requirement of lockstep execution. Current *superscalar* microarchitectures embody instruction level parallelism. These microarchitectures contain multiple *functional units (FUs)*, such as multiple integer adders or multiple floating point multipliers. In operation, multiple instructions from a single instruction stream are dispatched simultaneously to the multiple functional units, depending on compile time and run time detection of non-dependencies between the instructions. A recently developed variant of superscalar execution, *Simultaneous Multi-Threading (SMT)*, allows the instructions dispatched at a given time to come from multiple instruction threads [TEE+95]. The processor maintains multiple program counters, sharing other resources between the executing threads. This flexible and cost-effective strategy rather blurs the line between a single processor with multiple functional units, and a multiprocessor.

In this thesis, I introduce the *Single Function Multiple Data (SFMD)* architecture class, intermediate between SIMD and MIMD processing. SFMD architectures extend SIMD processing to better support tasks with data-dependent control flow, by relaxing the SIMD architectural requirement for lockstep execution. Figure 1.2 arranges some of these architectural alternatives along a spectrum of increasing *processor autonomy*. In an SIMD-like parallel architecture, some things are done globally, such as the distribution of instructions to the individual processors, by a *host* or *control* processor. Other things are done locally by the individual PEs. "Processor autonomy" refers to this global / local distinction, and specifically to how much is done locally. With *local conditional execution*, individual PEs determine which branch of an if statement they will execute; without it, the global processor determines this and sets or distributes a global mask vector to suspend execution for PEs not performing a given branch. With *local addressing*, individual PEs compute the memory addresses they will use, while without it this is done by the global processor. SFMD processing increases processor autonomy and decreases global control by breaking lockstep execution and letting individual PEs have control over some of the branching within their individual instruction streams. However, in order to keep the simple semantics of SIMD execution, communication between processors is restricted to only occur during phases of global lockstep execution. Full-fledged SPMD or MIMD relaxes this last notion of global execution by allowing individual PEs to communicate at any time, with local control of interprocessor communication (IPC).

We will see that the added cost of adding SFMD execution to an SIMD design is minor: the hardware cost will be perhaps one percent of chip area in the near future, while programming complexity remains the same. Moving to full-scale SPMD/MIMD execution, however, introduces the full range of parallel programming complexities. As to performance, I will show SFMD execution giving a substantial (1.5 - 2X) performance improvement over SIMD execution on several tasks exhibiting data-dependent control flow. SPMD execution will outperform SFMD execution on tasks with highly variable interprocessor communication, but this can sometimes be ameliorated by averaging out the communication variability, arranging the SFMD computation so that many communications are saved up during a "computation phase" and then executed in a "communication phase".

As to actual hardware for sensory processing, the CNAPS processor [Ham90] is an archetype for the type of parallel hardware I am thinking about. CNAPS is a highly cost-efficient SIMD architecture that puts 16 small DSP-like PEs on a single chip. These PEs each have a small (4KB) private memory they can address independent of each other. Instructions are fed to the PEs from off-chip. The PEs can individually decide not to participate in an instruction, depending on their private data. Access to external memory is shared among the PEs through a bus. There is hardware support for finding the PE holding the maximum value of a particular register ("parallel max").

## 1.2 Overview of the dissertation

Chapter 2 discusses the case of "ordered" input, where there is a natural ordering on the matching of model components with data. The archetypal example is matching Hidden Markov Models (HMMs) in optical character or speech recognition. The natural ordering gives rise to efficient dynamic programming implementations of the matching process. A reasonable view of context is then to condition probability estimates of the current input on previously classified inputs. These conditional probability estimates may be constructed using source modeling techniques.

It has recently been shown that HMMs are special cases of "probabilistic independence networks" (PINs) [SHJ97]. Using the PIN framework, I discuss how to integrate classifier outputs, viewed as probability estimates, with source models derived from sets of word models. I derive *Higher Order Viterbi Search (HOVS)*, a first-order Markov approximation to (the PIN generalization of) Viterbi search using higher-order source models. By simulation, I show that HOVS captures most of the benefit of using higher order source models, while being more time and space efficient than some reasonable competitors.

The PIN models are recursive, with each node having the same set of states, and one can make use of state parallelism. I give an SIMD implementation of HOVS, and discuss some restrictions on the source model required for a practical implementation.

Chapter 3 overviews the use of models for unordered input, specifically in vision algorithms. It looks at a variety of model-matching and grouping algorithms, and extracts some general characteristics relevant to parallelization:

1. feature grouping (object hypothesis generation) involves construction and traversal of irregular data structures such as lists and trees,

2. model matching algorithms exhibit irregular control flow mediated by the data and/or model,

3. most model matching and feature grouping algorithms are simple, with computational complexity coming from applying a small code "kernel" many times, and

4. model search and indexing techniques are "imperfect", in the sense that they generally restrict the number of models to be matched, but not to a single candidate.

The implications of these characteristics are (i) SIMD execution will sustain substantial performance degradation on these algorithms due to data dependent execution, (ii) only branching within a small kernel is important for performance, and (iii) there is model-parallelism to be exploited.

Chapter 4 reviews current technology trends with respect to VLSI, microprocessor architectures, and parallelism. The purpose is to describe the relevant factors and provide some reasonable numbers for measuring cost and performance. One conclusion of the chapter is that, with decreasing process size and concomitant increase in the number of objects that can be put on chip, adding complexity and functionality to processors is relatively inexpensive in terms of silicon area used. However, the more important conclusion from this review is that off-chip I/O bandwidth will be the most limiting technological factor in future microarchitectures.

Using area as the measure of cost, I derive a cost-performance model for on-chip parallelism for a class of architectures consisting of multiple processing elements per chip, each with some amount of private memory. The task for the architecture is assumed to be decomposable into a large number of independent subtasks, each of which first fetches some data and then processes it. The data may be fetched either from the processing element's private memory, or from off-chip. The subtasks are independent in that they can be executed without interprocessor communication. This task generalizes that of independently matching a large number of models to the data. The essential result of the chapter is that on-chip parallelism is fundamentally limited by the I/O bandwidth across the chip boundary. Effective parallelism degrades quickly as off-chip I/O is needed, unless the off-chip I/O bandwidth scales with the number of processors on the chip. One implication is that increasing chip area will result in not in adding more processors to the chip, but in making the individual processors more complicated, or in adding more per-processor memory.

The important conclusion is that, for the applications I consider, *there are only two viable alternatives*: for on-chip parallelism in the near future, if the range of target applications allows each model set to fit entirely on-chip, then an architecture of many small processors may be preferred, for its increased parallelism. This could be a vector processor or another SIMD architecture, where the simplicity of the individual processors allow many to be put on a single chip. In all other cases, bandwidth limitations imply that a few complex PEs will be preferred. In particular, a "middle ground" of many processors on a chip, each with external rather than on-chip memory, will not be viable, due to memory bandwidth limitations. With the coming practicality of embedding logic in a DRAM process, a design with all models on-chip is feasible and I henceforth concentrate on such designs, feeling that a design with a few complex PEs will be too close to a mainstream processor for economic viability.

Chapter 5 introduces and evaluates the *Single Function Multiple Data (SFMD)* class of parallel architectures, targeted toward extending SIMD processing to better support irregular computation and sparse communication, as is shown to be typical of model matching and feature grouping algorithms in chapter 3, and targeted toward designs with many relatively simple processing elements, as suggested by chapter 4.

SFMD extends the SIMD class by giving each processing element its own program counter and a small instruction memory, allowing separate control for within small loop bodies. The extra processor complexity is quite modest, on the order of one percent additional chip area, so that many processors will still fit on-chip. To preserve SIMD semantics, interprocessor communication is allowed to complete only after all PEs have synchronized. Preserving SIMD semantics keeps the programming and debugging simplicity of SIMD code given by its lack of race and deadlock conditions. However, compared to SIMD execution, the independent branching allowed by SFMD gives better performance on tasks with data-dependent execution, such as model matching, index traversal and sparse matrix computations.

Conversely, SFMD is outperformed by an SPMD architecture on tasks with sparse communication and highly varying computation and communication times. I derive asymptotic formulae bounding the speedup of SPMD over SFMD in a fairly general setting. Simulations show that the formulae give reasonable approximations in non-asymptotic domains. The analysis implies that when communication times are not too sparse, the performance gap between SFMD and SPMD may be counteracted by averaging out task and communication time variation, by performing multiple tasks between phases of multiple communications. In any case, SFMD outperforms SIMD on these kinds of tasks, and can extend the range of an existing SIMD design into the realm of tasks with irregular control flow at small cost, either in hardware, programming, or programming environment.

# Chapter 2

# Context for ordered input

## 2.1 Introduction and Goals

In this chapter I look at applying contextual information within a single "large" model. As always, the "context" used is the set of current interpretations of other components of the model. I assume that components of the model are interpreted sequentially, as for the interpretation of ordered input such as text or speech. Thus I wish to use the interpretations of previously evaluated components to better the interpretation of the current one. As interpretations are probabilistic, this leads to the use of a variable-order Markov source coding model for the sequence of component states.

As there is a single model, we are not interested in inter-model comparisons, but rather the interpretations of the components themselves. Thus, the task I examine is that of finding the most likely states of the components of the network, given some observations. For speech, this would correspond to finding the most likely phoneme or word string within a single HMM.

My definition of context implies that we look at situations where components' states are context-dependent, not where *observations* are. Context-dependent observations are appropriate for situations involving observation process phenomena analogous to coarticulation in speech (even though the HMM formalism used in speech assumes context-independent observations). The assumption of context-independent observations is reasonable for such domains as OCR and genome sequence modeling. The use of state-based context to improve classifier performance is appropriate, as we shall see, for constructing "unknown word" models and for improving classifier performance on novel (or unmodeled) words.

A further goal is to combine contextual information with a classifier, such as an MLP, to improve the classifier results. I take the usual interpretation of classifier outputs as conditional probabilities (e.g., [HP90]). We can then look at the single large model as a mechanism for integrating contextual information about likely state sequences with probabilistic information given by a classifier about the state values implied by the classifier inputs.

Given a single large model, parallelism is over evaluation of either nodes or states. Here, the ordered evaluation of the components precludes node parallelism, and we look at fine-grained parallelism over states. The fine-grained nature of the parallelism leads us to look for low-overhead "regular" algorithms; in particular I develop a dynamic programming recurrence, *Higher Order Viterbi Search (HOVS)*, suitable for SIMD implementation.

This chapter first reviews Probabilistic Independence Networks and their relation to HMMs as background for the definition of HOVS. I then define HOVS and talk about the relation to source modeling. Next, an SIMD implementation is given and its space and time complexity shown. Finally, I discuss the results of simulations comparing HOVS, simple Viterbi search, the more exact "stack decoding", and exhaustive per-model evaluation.

## 2.2 Background

The HOVS algorithm was originally developed independently of the Bayesian Networks, or *Probabilistic Independence Networks (PINs)* as they are now known [SHJ97]. However, HOVS is clearly related to Hidden Markov Models (HMMs), which are now known to be special cases of PINs [SHJ97], and the PIN formalism is a useful formalism for discussing HMM-type algorithms. Table 2.1 gives some notation that will be used in discussing PINs.

Table 2.1: PIN notation

| | |
|---|---|
| $\|X\|$ | number of values of a discrete random variable |
| $H$ | hidden state variable |
| $O$ | observable state variable |
| $h$ | value of a hidden state variable |
| $o$ | value of an observable state variable |
| $x_m^n$ | conjunction or "path" of values, $x_m^n \doteq \{x_m, \ldots x_n\}$ |
| $\cdot$ | concatenation of paths $x_m^l \cdot x_{l+1}^n = x_m^n$ |
| $m(x;y)$ | "mutual likelihood" of between $x$ and $y$, $m(x;y) \doteq p(x\|y)/p(x) \equiv p(y\|x)/p(y)$ |

### 2.2.1 Probabilistic Independence Networks

PINs provide a graphical formalism for expressing conditional independence assumptions about a set of variables, allowing the joint distribution to be factored into a product of conditional distributions[1]. Graphically, there are two formalisms for PINs, based on either undirected graphs (*UPINs*) or on DAGs (*DPINs*). I will use the directed graph formalism as it is somewhat simpler to describe; traditional Bayesian Networks use the DAG formalism. In a PIN graph, nodes correspond to variables, and arcs to conditional dependence assumptions. Arcs in a DPIN represent dependence in the following way: a node is conditionally independent of its ancestors given its parents. Thus, for example, in figure 2.1 (a) each $O_t$ is conditionally independent of all other variables, given $H_t$. This turns out to imply that the joint distribution over all variables can be written as a product of "local" distributions:

$$p(x_1^N) \doteq p(x_1 \ldots x_N) = \prod_{i=1}^{N} p(x_i|pa(x_i)) \tag{2.1}$$

where $pa(x_i)$ denotes the parents of node $x_i$, and $x_i^j$ denotes the conjunction of variables $\{x_i, x_{i+1}, \ldots, x_j\}$, for any (time-indexed) variable $x_t$. This factorization implies that only the local distributions need be estimated; these, of course, are typically much lower-dimensional than the entire joint distribution.

[1]This section reviews existing theory; see [SHJ97] for further details.

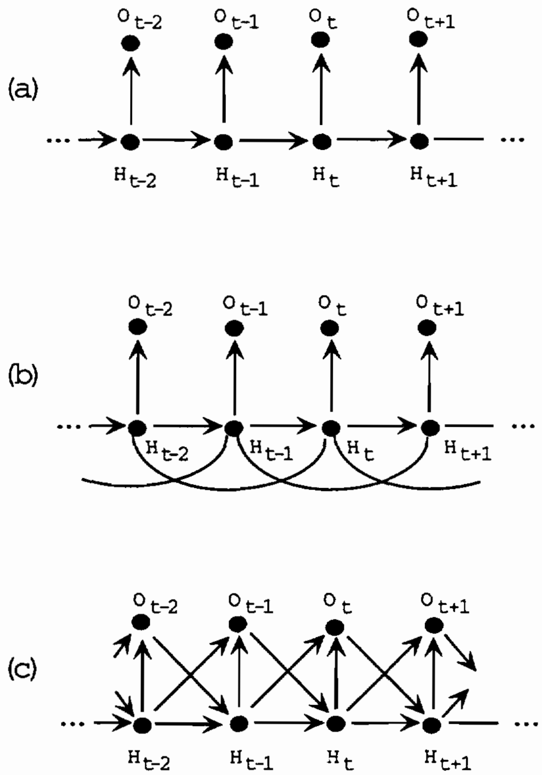


Figure 2.1: HMM-like PINs

As well as providing a formalism for factorizing joint distributions, the PIN framework also provides algorithms for incorporating new evidence about the states of nodes, computing the joint distribution of the variables, conditioned on the new evidence. The fundamental form is the "JLO" algorithm, which computes the MAP estimate of the joint distribution. A variant, the "Dawid" algorithm, computes the most likely set of states given the evidence; this is sometimes called the *most probable explanation (MPE)* in the Bayesian Network literature [Pea88].

### 2.2.2 HMM-like PINs

HMMs are special cases of PINs, where the ordering of the observations leads to a recursive network structure. In an HMM there are two types of nodes, corresponding to observations, $O_t$, and hidden states, $H_t$, at particular points in time $t$. [2] The graphical form is shown in figure 2.1 (a), where it can be seen that observations depend only on the contemporaneous state, and states depend only on their neighboring states[3]. In this setup, the most general notion of context in our sense is the current probability distribution over the values (interpretations) of the $H_t$ and $O_t$. As above, the directed arcs of the PIN describe conditional dependencies. However, viewed as undirected arcs, they describe *context dependencies*: the interpretation (value) of a node is determined only by the context consisting of the interpretations of all nodes to which it is connected. Of course, the interpretation of the nodes to which it is connected depends on the interpretations of all nodes to which *they* are connected, and so on, so that the full context is the distribution of values over all states. However, only the distribution of values of the states connected to a node are used in computing the distribution of values for that node.

For figure 2.1 (a), each observation, $O_t$ is context dependent only on the hidden state $H_t$, while each $H_t$ is context dependent on $O_t$, $H_{t-1}$ and $H_{t+1}$. However, each $H_t$ is *conditionally* dependent only on the previous $H_{t-1}$. Figure 2.1 also shows two HMM-like PINs that use context in different ways. The graph in figure 2.1 (b) gives a model where the state $H_t$ at a given time is context-dependent on some number of previous states (two in this case), as well as on the corresponding $O_t$. This is the form of context I will be concerned with: as nodes are processed in temporal order, previous interpretations affect the current one. Henceforth in this chapter, the *context* of a node will be the set of previous nodes with which it is context dependent, so in figure 2.1 (b), the context (for the interpretation) of $H_t$ is $\{H_{t-2}, H_{t-1}\}$. Figure 2.1 (c) shows a model where *observations*

[2]HMMs that explicitly model duration as described in [Rab90] are equivalent to averages of HMMs of the type described. Given a maximum duration, D, consider the set of HMM PINs where each state node is connected to between 1 and D observation nodes. The algorithm in [Rab90] is equivalent to model averaging over this set of PINs, although more efficiently implemented than by explicit construction and evaluation of all the models.

[3]This is different from the usual graphical representation of an HMM as a probabilistic automaton, where nodes refer to states, not states at a particular time, and time is implicit. For example, the automaton formalism allows self-loops corresponding to the probability of remaining in the same state at the next timestep. In a PIN, the same information would be carried as part of the conditional distribution $p(H_{t+1}|H_t)$.

are context dependent on the previous, current, and following states (although they are conditionally dependent only on previous and current state). This might be a reasonable model for coarticulation-like situations, and has been proposed as such by Saul and Jordan [SHJ97].

For HMM PINs, the JLO algorithm reduces to the usual forward-backward algorithm, and the Dawid algorithm reduces to Viterbi search. The forward recursions of the Dawid algorithm for the structures in figure 2.1 is given in table 2.2. The corresponding formulae for the JLO algorithm are obtained by replacing the maximizations by summations over the same variables. Here, lowercase $o_t$ and $h_t$ refer to specific (discrete) values for the variables $O_t$ and $H_t$, respectively.

Table 2.2: Forward recursions for HMM-like PINs, from the Dawid algorithm. The recursion variable is marked with an asterisk.

| structure | forward recursion |
|---|---|
| a | $p^*(h_t, o_1^t) = p(o_t \mid h_t) \max_{h_{t-1}} p(h_t \mid h_{t-1}) p^*(h_{t-1}, o_1^{t-1})$ |
| b | $p^*(h_{t-1}^t, o_1^t) = p(o_t \mid h_t) \max_{h_{t-2}} p(h_t \mid h_{t-2}^{t-1}) p^*(h_{t-2}^{t-1}, o_1^{t-1})$ |
| c | $p^*(h_t^{t+1}, o_1^t) = \max_{h_{t-1}} p(o_t, h_{t+1} \mid h_{t-1}^t) p^*(h_{t-1}^t, o_1^{t-1})$ |

At the last observation, $t = N$, these recursions give us a value for $p(h_{N-k}^N, o_1^N)$, for some $k$. Summing over the possible state $k$-tuples gives the joint probability of the observations and the model (the model index has been omitted in the formulae given).

Practically, the recursions for (b) and (c) are problematic. First, the recursions for structures (b) and (c) are both over $h_t^{t+1}$, i.e. there are $|H|^2$ values to be computed at each step, where $|H|$ is the number of states. For general structures like (b) and (c), but with more context, the number of values grows as $|H|^n$, where $n$ is the amount of context. This becomes impractical for all but very small $n$.

Another useful way to look at the model described by HMM-like PIN structures is through product formula. PINs model the joint density of their variables; for HMM-like

PINs this is $p(o_1^N, h_1^N)$. Using the chain rule for probabilities, one can write

$$p(o_1^N, h_1^N) = p(o_1, h_1) \prod_{t=2}^{N} p(o_t, h_t | o_1^{t-1}, h_1^{t-1}).$$

Using the independence assumptions given by the graphical structures gives the product formulae in table 2.3.

Table 2.3: Product formulae for HMM-like PINs.

| model | product formula |
|---|---|
| a | $p(h_1^N, o_1^N) = p(o_1, h_1) \prod_{t=2}^N p(o_t \mid h_t) p(h_t \mid h_{t-1})$ |
| b | $p(h_1^N, o_1^N) = p(o_1^2, h_1^2) \prod_{t=3}^N p(o_t \mid h_t) p(h_t \mid h_{t-2}^{t-1})$ |
| c | $p(h_1^N, o_1^N) = p(o_1^2, h_1^2) \prod_{t=3}^N p(o_t, h_t \mid h_{t-2}^{t-1})$ |

### 2.2.3 Classifiers and Likelihoods

Recall that I am interested in integrating contextual information with classifier outputs. Consider a classifier with input vector $x$ and trained with target vector $t$, where the $i$'th output corresponds to membership of $x$ in the $i$'th class, $y_i$. It is well known (see, for example, [HP90]) for a variety of cost functions including mean squared error (MSE) and cross-entropy (CE), using targets in $t_i \in \{0, 1\}$, that network outputs converge to posterior probabilities, the $i$'th output approximating $p(y_i|x)$, The specific assumptions are sufficiently many training examples, convergence to a global minimum, and adequate network representational capability to approximate the posterior function.

In the HMM PIN case, the posteriors are $p(h_t|o_t)$. However, the terms in the recursions involve likelihoods like $p(o_t|h_t)$, so an application of Bayes rule is suggested: $p(o_t|h_t) = p(h_t|o_t)p(o_t)/p(h_t)$. This involves a term, $p(o_t)$, that is constant across models, and can generally be ignored. Denote the remaining term as $m(o_t; h_t) \doteq p(h_t|o_t)/p(h_t)$. Refer to $m(x;y)$ as the *mutual likelihood* of $x$ and $y$, noting that $m(x;y) = m(y;x)$. Note also that the normal mutual information is $\int p(x,y) \log m(x;y)$. One may estimate $m(o_t; h_t)$ by taking the classifier estimate of $p(h_t|o_t)$ and dividing by $p(h_t)$, estimated, say, over the training set. Experimentally, I find this problematic for low-probability states: these tend to be less well estimated, due to fewer training examples, and subsequent division by small $p(h_t)$ exacerbates the mis-estimation.

An alternative is to use the classifier to estimate $m(o_t; h_t)$ directly. From results of [HP90], or by trivial modification of the proofs therein, one has the following, under the same assumptions as for classifier outputs to approximate posterior probabilities. If a classifier is trained using MSE with the targets of the $i$'th output being either 0 or $a_i$, where $a_i > 0$, then the $i$'th classifier output will approximate $a_i$ times the posterior. The same result holds for the cross-entropy cost function, modified to be of the form $-\sum_i [t_i \log y_i + (a_i - t_i)\log(a_i - y_i)]$. Thus by taking $a_i = 1/p(y_i)$, where $y_i$ is the $i$'th class, the $i$'th classifier output will approximate $m(x; y_i)$. Again, $p(y_i)$ can be estimated from the training set. A straightforward gradient calculation shows that exactly the same computation may be achieved for a network with 0/1 outputs by using a weighted error measure, for example, $MSE_{\vec{a}} \doteq \sum a_i^2 (t_i - y_i)^2$, and then multiplying the outputs by $a_i$ : $y_i \to a_i y_i$ (but this is *not* the same as training with the usual error function and then multiplying the outputs).

The basic advantage of direct estimation of $m(o_t; h_t)$ is that it is the quantity which will actually be used in the recursions, and whose estimation should be optimized. If our classifiers closely approximated posteriors, there would be no difference between the two methods. However, the assumptions above cannot be expected to hold in practice, so the methods can be expected to differ: they apply differing costs to mis-estimation of the posterior for each class, leading to different estimates. It seems likely to be better to minimize the mis-estimation of the $m(o_t; h_t)$ rather than compound any mis-estimation of $p(h_t|o_t)$ by division by $p(h_t)$.

### 2.2.4 Context-dependent Observations

Although not my direct goal, it is of some interest to consider the case of context-dependent observations, as this is another form of contextual influence and as it one of the simplest alternative HMM PIN structures. Consider the structure given in figure 2.1 (c), with corresponding recursion from table 2.2 (c) and product formula from table 2.3 (c). As well as the fact that the recursion is over $|H^2|$ variables, a second problem with the recursion lies in the terms $p(o_t, h_{t+1}|h_{t-1}^t)$. Use of Bayes rule gives $p(o_t, h_{t+1}|h_{t-1}^t) = p(h_{t-1}^{t+1}|o_t)p(o_t)/p(h_{t-1}^t)$. The term to be estimated by the classifier is $p(h_{t-1}^{t+1}|o_t)$, which requires $|H^3|$ classifier outputs. In general, such a model is prohibitively large, requiring an impractically large amount of data to train.

A possible approach is as follows. We are interested in terms of the form $h_{t-m}^{t+n}$, for some $n$ and $m$. Assume there are equivalence classes of context, $C(h_{t-m}^{t+n})$, so that $p(o_t|h_{t-m}^{t+n}) \approx p(o_t|\widetilde{h_{t-m}^{t+n}})$ for all $o_t$, whenever $h_{t-m}^{t+n}$ and $\widetilde{h_{t-m}^{t+n}}$ are in the same equivalence class. A plausible example of possible equivalence classes would be those of the form $C(h_{t-m}^{t+n}) = \{\widetilde{h_{t-1}}h_t\widetilde{h_{t+1}}|\widetilde{h_{t-1}} \in c_{t-1}^{(l)}, \widetilde{h_{t+1}} \in c_{t+1}^{(r)}\}$, where $c_t^{(l)}$ and $c_t^{(r)}$ are classes of equivalent left and right contexts; for example, for speech, all bursts might be taken to be the same right context. An efficient approach to training a classifier for such "L-R context" equivalence classes is given by the CDNN network of [BM94].

Let $c$ and $\tilde{c}$ be two members of the same equivalence class, $C$, and let

$$\epsilon(\tilde{c}, c) \doteq p(o_t|\tilde{c}) - p(o_t|c)$$

be the error of approximating one probability by the other. Then

$$p(\tilde{c}|o_t) = \frac{p(\tilde{c})}{p(o_t)}[p(o_t|c) + \epsilon(\tilde{c}, c)] = \frac{p(\tilde{c})}{p(c)}p(c|o_t) + \frac{p(\tilde{c})}{p(o_t)}\epsilon(\tilde{c}, c),$$

and

$$\begin{aligned} p(C|o_t) &\equiv p(\vee_{\tilde{c}\in C}\tilde{c}|o_t) \\ &= \sum_C p(\tilde{c}|o_t) \qquad \text{(independence given the observation)} \\ &= \frac{p(c|o_t)}{p(c)}\sum_C p(\tilde{c}) + \frac{1}{p(o_t)}\sum_C p(\tilde{c})\epsilon(\tilde{c}, c) \\ &= p(c|o_t)\frac{p(C)}{p(c)} + \frac{(C)}{p(o_t)}\bar{\epsilon}(c), \end{aligned}$$

where $\bar{\epsilon}(c)$ is the average error over the class, with respect to $c$,

$$\bar{\epsilon}(c) \doteq \frac{\sum_{\tilde{c}\in C} p(\tilde{c})\epsilon(\tilde{c}, c)}{\sum_{\tilde{c}\in c} p(\tilde{c})},$$

and $p(C) = \sum_{\tilde{c}\in C} p(\tilde{c})$. So, for any $c \in C$, we have

$$p(c|o_t) = \frac{p(c)}{p(C)}[p(C|o_t) - \frac{p(C)}{p(o_t)}\bar{\epsilon}(c)], \tag{2.2}$$

or, in terms of the mutual likelihood, $m(x; y) \doteq p(x|y)/p(x)$,

$$m(c; o_t) = m(C; o_t) - \frac{\bar{\epsilon}(c)}{p(o_t)}. \tag{2.3}$$

Thus classifier estimates of $p(C|o)$ or $m(C; o)$ may be converted to estimates of $p(c|o)$ or $m(c; o)$, respectively. Conversion to $p(c|o)$ requires multiplication by $p(c)/p(C)$, which can be estimated from the training set. The advantage of this equivalence class formulation is that the number of classifier outputs equals the number of classes, rather than the number of $h_{t-m}^{t+n}$, $|H|^{m+n+1}$, and hence may be a practicable number. Note, however, that conversion may have poor error properties for low probability observations ($p(o_t)$ small). So if there are equivalence classes, one may be able to implement this efficiently, otherwise, however, even this simple alternative structure may be computationally intractable.

## 2.3 HOVS Algorithm

The Higher Order Viterbi Search (HOVS) algorithm can be seen as a variant of the Dawid algorithm that allows the cost-efficient use of more state-context information, by using a variable amount of context from the most likely path to a state, rather than a fixed number of previous states.

### 2.3.1 PIN, recursion, product form

The HOVS algorithm assumes a PIN model like that of figure 2.1 (b), but where the state-to-state arcs go arbitrarily far back in time. More formally, in equation (2.1) the parents of a state $H_t$ are $pa(H_t) = H_1^{t-1}$, and those of a state $O_t$ are $pa(O_t) = H_t$. To derive the HOVS recursion from first principles, write

$$\begin{aligned} p(h_1^t, o_1^t) &= p(h_1^t)p(o_1^t|h_1^t) \\ &= p(h_1^t)p(o_t|h_t)p(o_1^{t-1}|h_1^{t-1}) \\ &= p(h_1^t)p(h_t|o_t)p(h_1^{t-1}, o_1^{t-1})\frac{p(o_t)}{p(h_t)p(h_1^{t-1})} \\ &= p(h_t|h_1^{t-1})m(h_t; o_t)p(o_t)p(h_1^{t-1}, o_1^{t-1}), \end{aligned} \tag{2.4}$$

where the second equality is from the conditional independence assumptions of the PIN graph.

For each state, $h_t$, let

$$\pi(h_t) \doteq h_t \cdot \operatorname*{argmax}_{h_1^{t-1}} p(h_1^t, o_1^t)$$

denote the state sequence ending in $h_t$ having the largest joint probability with the observations. Let

$$\tilde{p}(h_1^t, o_1^t) = p(h_1^t, o_1^t) / \prod_{\tau=1}^{t} p(o_\tau),$$

then (2.4) implies

$$\tilde{p}(h_1^t, o_1^t) = p(h_t|h_1^{t-1})m(h_t; o_t)\tilde{p}(h_1^{t-1}, o_1^{t-1}).$$

The HOVS approximation to the optimal state sequence uses (2.4) to forcibly construct a dynamic programming recursion:

$$\begin{aligned} \pi(h_t) &\doteq h_t \cdot \operatorname*{argmax}_{h_1^{t-1}} p(h_1^t, o_1^t) \\ &= h_t \cdot \operatorname*{argmax}_{h_1^{t-1}} \tilde{p}(h_1^t, o_1^t) \\ &= h_t \cdot \operatorname*{argmax}_{h_1^{t-1}} [p(h_t|h_1^{t-1})m(h_t; o_t)\tilde{p}(h_1^{t-1}, o_1^{t-1})] \\ &= h_t \cdot \operatorname*{argmax}_{h_1^{t-1}} [p(h_t|h_1^{t-1})\tilde{p}(h_1^{t-1}, o_1^{t-1})] \\ &\approx h_t \cdot \pi(\operatorname*{argmax}_{h_{t-1}} [p(h_t|\pi(h_{t-1}))\tilde{p}(\pi(h_{t-1}), o_1^{t-1})]). \end{aligned} \tag{2.5}$$

The last line makes the HOVS approximation by replacing the *argmax* over the entire preceding path $h_1^{t-1}$ by one over the preceding single state $h_{t-1}$. Context deeper than 1 is assumed to be from the single best path to the preceding state rather than maximization over all paths to that state.

Clearly, the *argmax* of the last line can be viewed as being over $\pi(h_{t-1})$, and can be replaced by an *argmax* over multiple paths to each $h_{t-1}$, rather than the single best path. Of course, if $n$ such paths are used, then the recursion must maintain $2n$ state variables for each $h_t$, rather than 2 (namely, $n$ copies of $\pi(h_t)$ and $\tilde{p}(\pi(h_t), o_1^{t-1})$) However, in situations where the classifier tends to give the correct class a high score, even if not the highest, this technique should be useful.

The HOVS recursion is on the $|H|$ possible values of $H$. It can easily be extended to a recursion on $(n+1)$-tuples of values, $\{h_{t-n}^t\}$, allowing a potentially more accurate estimation, at the cost of more computation:

$$\begin{aligned}
\pi(h_{t-n}^t) &\doteq h_{t-n}^t \cdot \operatorname*{argmax}_{h_1^{t-n-1}} p(h_1^t, o_1^t) \\
&= h_{t-n}^t \cdot \operatorname*{argmax}_{h_1^{t-n-1}} [p(h_t|h_1^{t-1})\tilde{p}(h_1^{t-1}, o_1^{t-1})] \\
&\approx h_t \cdot \pi(h_{t-n}^{t-1} \cdot \operatorname*{argmax}_{h_{t-n-1}} [p(h_t|\pi(h_{t-n-1}^{t-1}))\tilde{p}(\pi(h_{t-n-1}^{t-1}), o_1^{t-1})]). \qquad (2.6)
\end{aligned}$$

A recursion on longer tuples should be more accurate as it allows the *argmax* for each tuple to be based on a longer lookahead.

Comparison shows that HOVS is essentially the Dawid algorithm for the assumed PIN model, derived in a different way, and allowing for the introduction of further state context. From table 2.2 (c), the Dawid algorithm for context depth 2 gives the probability recursion

$$p(h_{t-1}^t, o_1^t) = p(o_t|h_t) \max_{h_{t-2}} p(h_t|h_{t-2}^{t-1})p(h_{t-2}^{t-1}, o_1^{t-1}).$$

The optimal path is determined by recording the maximizing value of $h_{t-2}$ at each step. This implies the path recursion

$$\pi(h_{t-1}^t) = h_t \cdot \pi(h_{t-1} \operatorname*{argmax}_{h_{t-2}} p(h_t|h_{t-2}^{t-1})p(h_{t-2}^{t-1}, o_1^{t-1})). \qquad (2.7)$$

This is essentially the same as (2.6) for $n = 1$, except that HOVS uses $p(h_t|\pi(h_{t-2}^{t-1}))$ rather than $p(h_t|h_{t-2}^{t-1})$. Similar equivalences hold for all $n \geq 0$, so HOVS can be seen as a variant of the Dawid algorithm that allows the use of more state-context information when the modeling assumptions of the Dawid algorithm are inexact, i.e., when the Markov property does not hold strictly and there is dependence between variables further apart in time than the Dawid algorithm assumes. As the Dawid algorithm for $n = 0$ is the usual Viterbi search in an HMM, the same statement holds for that situation.

### 2.3.2 Relation to speech language modeling: A* search and stack decoding

As the PIN independence assumptions are the same, the product form for HOVS is that of table 2.3 (b). Ignoring initial conditions ($t < n$), where $n$ is the context depth,

$$\begin{aligned}
p(o_1^N, h_1^N) &= \prod_{t=n+1}^{N} p(o_t|h_t)p(h_t|h_{t-n}^{t-1}) \\
&= \prod_{t=n+1}^{N} p(o_t|h_t) \prod_{t=n+1}^{N} p(h_t|h_{t-n}^{t-1}) \\
&= p(o_1^N|h_1^N)p(h_1^N)
\end{aligned}$$

which in the speech recognition domain is the usual decomposition into acoustic and language models. So the variable-order aspect of HOVS is clearly related to the use of $n$-gram language models in speech. However, the algorithm is not that which is usually used in speech processing. Speech processing, both for isolated and continuous speech, requires time alignment of acoustic frames with the word sequence [RL90]. This leads to the use of "stack decode" or A* algorithms that evaluate and compare entire aligned word sequences [Pau91]. As (some approximation to) all preceding word sequences are available for each new word, rather than simply the last word, the HOVS approximation (2.5) need not be made: there is no need to enforce a Markovian assumption. Thus stack decode algorithms should be more accurate. The disadvantages are (i) the number of paths grows exponentially over time, thus some approximation is needed to keep the set of paths within reason, and (ii) these A* algorithms do not appear to parallelize in any useful way, and are much more space and computation intensive.

### 2.3.3 Variable order models

I have discussed estimating $m(o_t; h_t)$ in the HOVS recursion; it remains to discuss the term $p(h_t|h_1^{t-1})$. I will assume that the influence of a previous state only extends so far, so that

$$p(H_t|H_1^{t-1}) = p(H_t|H_{t-n}^{t-1}), \tag{2.8}$$

for some fixed $n$. Even so, the latter term is a distribution on $|H|^{n-1}$ state tuples, and hence becomes prohibitively large, both to estimate and to store, for relatively small $n$. I reduce the space needs by using variable-order models: the term $p(h_t|\hat{h}_{t-n}^{t-1})$ in the recursion is replaced by a term $p(h_t|\hat{h}_{t-\alpha}^{t-1})$, where $\alpha$ is a function of $\hat{h}_{t-n}^{t-1}$, $0 \leq \alpha \leq n$, and $\hat{h}_t^{t-1}$ is interpreted as the empty set (the *null* context). The function $\alpha$ is constructed to trade off the accuracy of the approximation $p(h_t|\hat{h}_{t-n}^{t-1}) \approx p(h_t|\hat{h}_{t-\alpha}^{t-1})$ against the complexity of the model $\mathcal{M}_\alpha \doteq \{h_{t-\alpha}^t | h \in H\}$, i.e., the number of contexts used (I use $h \in H$ to indicate that $h$ is one of the possible values of $H$).

Optimal methods for constructing models $\mathcal{M}_\alpha$ have been developed in the data compression literature [WRF95], under the assumption of a finite-state source, as given by equation 2.8. "Optimal" in this case means that the probabilities $p(h_t|\hat{h}_{t-\alpha}^{t-1})$ of the "learned" model $\mathcal{M}_\alpha$ converge to those of the true source, $p(h_t|\hat{h}_{t-n}^{t-1})$ at the fastest possible rate, as a function of the amount of training data [WRF95, Ris86].

#### Context trees

For a variable $H$ with states, $h^{(i)}$, $1 \leq i \leq |H|$, consider the full $|H|$-ary tree where each node is identified with a state sequence, or *context* as follows: the root corresponds to the null state sequence, and the $i$'th child of a node corresponds to appending $h^{(i)}$ to that node's state sequence. Thus, if the context for a node is $h^{(j_k)} \cdots h^{(j_1)}$, then the context for its $i$-th child is $h^{(i)} \cdot h^{(j_k)} \cdots h^{(j_1)}$. Given an information source generating a state sequence, to each node, attach a set of probability estimates. If the context for a node is $h^{(j_k)} \cdots h^{(j_1)}$, then the estimates are for the occurrence of a state as the next state after the context: $\{p(h^{(i)} \cdot h^{(j_k)} \cdots h^{(j_1)} | h^{(j_k)} \cdots h^{(j_1)}\}$, or written temporally, as previously,

$\{p(h_t^{(i)}|h_{t-1}^{(j_k)}\cdots h_{t-k}^{(j_1)}\}$. When no confusion will arise, I also refer to the combination of a state sequence and a set of probability estimates as a "context".

Call a tree of state sequences with such a set of probability estimates a *context tree* for the information source. Define a *fringe* of a tree to be a set of nodes such that all sufficiently long paths starting at the root of the tree pass through exactly one element of the fringe. Given a context tree for the source, each such fringe defines an $\alpha$ function as described above, and hence a model for the source. Algorithms for constructing context tree models differ in how they select the fringe of "active" contexts to use (and to some extent in how they estimate the probabilities). The essential idea of these algorithms is to expand the fringe by replacing a node on the fringe by its children whenever the resulting improvement in the probability model outweighs the increase in model complexity.

As used in data compression, context trees are constructed in an online fashion. In this case, I use a context tree constructed for a training set to capture state context statistics for use with a classifier. Of course, such a context tree can be continuously adapted to the state sequence so generated. However, there are two implementation issues in using a context tree for HOVS.

One simple point is that, both for starting up the recursion, and for dealing with state-sequences that were not seen during the construction of the tree, all ancestors of the active fringe must be included as potential contexts. The second issue is more complex.

Traversing a sequence of states defines a map $\gamma$ from the set of active contexts into itself by

$$\gamma_h(\alpha(h_1^n)) = \alpha(h \cdot h_2^n). \tag{2.9}$$

In implementing HOVS, one needs to implement something like $\gamma_h$ to compute the active context for the next state $h_{t+1}$ from the context for the current state. The difficulty is that, in general, computing $\gamma_h$ requires not just the current context and the next state, but also that part of $h_2^n$ not in the current context. The preceding states, $h_2^n$, are needed when the next active context is longer than the current active context by more than one. Suppose the current context is $h_t^1$, the next state is $h^{(0)}$, and the active contexts ending in $h^{(0)}$ are $\{h^{(0)}h^{(1)}h^{(i)}|h^{(i)} \in H\}$. Computing the next active context requires knowing which $h^{(i)}$ occurs in the appropriate place in $\pi(h_t^1)$.

So if context lengths can differ by more than 1, computing $\gamma_h$ generally requires back-tracing through $\pi(h_t^1)$, which may be computationally expensive, especially in a parallel implementation where the back-tracing information may be distributed. An alternative is to use a finite state machine approximation to $\gamma_h$. Define $\hat{\gamma}_h(\alpha(h_1^n))$ to be the shorter of $\alpha(h \cdot h_2^n)$ and $h \cdot \alpha(h_1^n)$. Since $\alpha()$ computes a (non-strict) prefix of its argument, one of $\alpha(h \cdot h_2^n)$ and $h \cdot \alpha(h_1^n)$ is a prefix of the other, so $\hat{\gamma}_h(\alpha(h_1^n))$ is a prefix of $\gamma_h(\alpha(h_1^n))$ and is a reasonable, if not optimal, context for $h$. Also, $\hat{\gamma}_h$ and $\gamma_h$ agree whenever the next active context can be computed from only the next state and the current context, i.e., without back-tracing. The advantage of $\hat{\gamma}$, of course, is that it admits a finite state machine implementation[4].

## 2.4 SIMD HOVS

In this section, I examine SIMD implementations of the HOVS algorithm. Equations (2.5) and (2.5) suggest either

1. parallelizing over the max when computing the recursion for each $\pi(h_t)$, or

2. computing several $\pi(h_t)$ in parallel.

Both implementations are possible, the choice being determined by such factors as number of PEs, $P$, the number of states, $|H|$, the amount of contextual information used (available storage for $\mathcal{M}_\alpha$), and architectural features such as the amount of local memory per PE and whether one can compute the maximum over two sets of PEs simultaneously. Given an algorithm for (1), assuming the algorithm that can be executed in parallel on the

[4]In the data compression literature, there is a notion of a *finite state machine source model* [WRF95]. Constructing $\hat{\gamma}$ from the active contexts in a context tree, as above, does build a finite state machine model, but not necessarily an optimal one, as the active contexts have not been chosen with the FSM restriction in mind. In particular, it may be better to choose a deeper context at some point so that deeper successor contexts can be used.

architecture, (2) is unproblematic, so I will concentrate on the first approach. The basic algorithm is straightforward, with minor variants for (i) $|H| \approx P$, (ii) $|H| \ll P$, (iii) $|H| \gg P$ and (iv) recursion on $n$-tuples, rather than single states (equation (2.6)).

### 2.4.1 Algorithm

I sketch SIMD implementations of the HOVS algorithm, for simplicity ignoring starting and ending conditions. Pseudo-code for the algorithms is in Appendix A.

The assumed machine model is like the Adaptive Solutions' CNAPS [Ham90] : there is a separate host and processor element (PE) array; values can be broadcast from either the host or a PE and received by both the host and all PEs; PEs have local memory, local memory addressing, and local conditional execution; and there is a fast parallel max where each PE emits a value and the identifier of some PE that emitted the maximum value is determined in *constant* time.

Recall equation (2.5):

$$\pi(h_t) = h_t \cdot \pi(\underset{h_{t-1}}{\text{argmax}}[p(h_t|\pi(h_{t-1}))\tilde{p}(\pi(h_{t-1}), o_1^{t-1})]).$$

In the algorithm for the case $|H| = P$, each state $h$ is associated with a particular PE, $q_h$, which $q_h$ stores the context probabilities $\{p(h'|\pi(h))|h' \in H\}$. At time $t$, $q_h$ holds the current value of $\tilde{p}(\pi(h), o_1^{t-1})$, uses it to compute $f_h(h') \doteq p(h'|\pi(h))\tilde{p}(\pi(h), o_1^{t-1})$, and stores the new value of $\tilde{p}(\pi(h), o_1^t)$.

At the $t$-th iteration, the algorithm loops over the recursion variable $h_t$. For each $h_t$, $f_{h_{t-1}}(h_t)$ is evaluated in parallel on $q_{h_{t-1}}$, and a parallel max operation over all PEs gives the desired $\text{argmax}_{h_{t-1}}$. This resides on $q_{\max} \doteq q_{\text{argmax}_{h_{t-1}}}$; point-to-point communication transfers the new $\tilde{p}(\pi(h_t), o_1^t)$ from $q_{\max}$ to $q_{h_t}$, as well as the new active context for $q_{h_t}$, for use in the next iteration. The host maintains a back-pointer from $h_t$ to $\text{argmax}_{h_{t-1}}$; after the last iteration a serial back-trace from the final state gives the sequence of maximal states. Pseudo-code for this case is in figure A.1. Time complexity of one step of the algorithm is $O(|H|) = O(|H|^2/P)$, for a speedup of $O(P)$ over the serial version.

Minor variations need to be made when $|H| > P$, $|H| \ll P$, or when the recursion is over $n$-tuples rather than single states.

When $|H| > P$, the same algorithm can be used, except that each state $h_{t-1}$ is associated with a *virtual* PE (*VPE*) [5], $v_{h_{t-1}}$. Each real PE contains $K \doteq \lceil H/P \rceil$ VPEs and an inner loop over the VPEs is added when calculating the max. Pseudo-code for this case is in figure A.2. Time complexity of the algorithm is $O(|H|K)$ for a speedup of $O(|H|/\lceil |H|/P \rceil)$.

When $|H| \ll P$, the basic algorithm uses only $|H|$ of the PEs, which is inefficient. One possibility is to simultaneously compute the recursion for $L \leq \lfloor P/|H| \rfloor$ variables $h_t$. The PE array is partitioned into $L$ sets of $|H|$ PEs and the loop body of the basic algorithm is executed on each, $\lceil |H|/L \rceil$ times. Broadcasting variables requires a loop over the $L$ sets, as does the parallel max, and the point-to-point communication. Time complexity is $\max(L, \lceil |H|/L \rceil)$. Speedup depends on $L$; ignoring integrality constraints, $L = \sqrt{H}$ gives the maximum speedup, $H^{3/2}$, using $H^{3/2}$ PEs. .

Another possibility when $|H| \ll P$ is to use the "extra" PEs to store more context nodes. This gives only $|H|$-fold parallelism, but allows more contextual information to be used, potentially increasing accuracy. Rather than all contexts ending in some particular state, each VPE holds the contexts ending in a particular state sequence. More formally, the set of contexts for a VPE is the intersection of the active fringe with some complete subtree of the context tree, where the subtree's root need not be an immediate descendent of the root of the tree. In this case, a compatibility condition is needed to determine which VPEs contribute to the $\text{argmax}_{h_{t-1}}$, since not all contexts can precede all others (for example, the context $h^{(1)}h^{(2)}$ can only precede contexts of the form $h^{(i)}h^{(1)}$). This reduces the potential parallelism to $|H|$. Pseudo-code for this case is in figure A.3. Let $S$ be the number of contexts, then $K \doteq \lceil S/P \rceil$ is the maximum number of VPEs (or context subtrees) associated with any PE. Time complexity of the algorithm

[5]Note that I am not assuming any hardware support for virtualization; it is purely an algorithmic construct.

is $O(|H|\lceil S/|H|\rceil\lceil S/P\rceil) \approx S^2/P = (|H|^2/P) * (S/|H|)^2$. Speedup is $O(P/(S/|H|)^2)$, less than $P$ due to computation of the compatibility condition.

Finally, one may vary these algorithms slightly to implement a recursion on, say, pairs $h_{t-1}^{t}$, as in equation (2.6). The problem here is that the PE for $h^{(1)}$ must contain at least the $|H|^2$ context probabilities $\{p(h|h^{(1)}h^{(2)})|h, h^{(2)} \in H\}$, which may be prohibitive in terms of storage. Pseudo-code for this case, when $|H| = P$ is in figure A.4. Time complexity of the algorithm is $O(|H|^2) = O(|H|^3/P)$, for a speedup of $P$.

### 2.4.2 Discussion

The maximum speedup possible for the recursions (2.5) and (2.6) is $|H|$, so the optimal speedup with $P$ processors is $\min(|H|, P)$. When $P$ divides $|H|$, one can get the maximal speedup of $P$. When $P$ does not divide $|H|$, one can get a speedup of $O(|H|/\lceil|H|/P\rceil)$, which is bounded below by $P/2$, the worst case being $|H| == P+1$. Even when $|H| \ll P$, one can get a speedup of $|H|^{3/2}$ by computing multiple recursions in parallel. So one can generally make good use of parallelism.

As to the space complexity, each context requires $|H|$ probabilities and $|H|$ "next context" identifiers. In a space-restricted implementation, the probabilities can likely be represented in 16 bits, as can the "next context" identifiers, for a context size of $4|H|$ bytes. Thus, for example, with $|H| = 27$, $P = 16$, and 2KB context memory per processor, one could use about 300 contexts, and achieve a speedup of $O(13.5)$. For $|H| = 53$, $P = 16$ and 4KB context memory per processor, one would have 300 contexts and $O(12.25)$ speedup. For $|H| = 53$, $P = 32$ and 4KB context memory per processor, one would have 300 contexts and $O(26.5)$ speedup. As will be seen from simulations, a few hundred contexts should provide most of the performance gain, so one can conclude that the storage requirements of the algorithm are practical.

## 2.5 Simulations

I address a number of issues via simulation, seeking some general idea of the performance of these algorithms. The specific issues are as follows. How does performance vary with the amount of context used (correctness versus size)? How much does performance degrade when context is restricted to allow a finite state machine algorithm for the SIMD implementation? How much does performance degrade due to the HOVS approximation (2.5) How well can one model a source before it is "over-trained", and does not represent well another text?

My setup for these experiments simulates an OCR system. I use output vectors from any of several ANN classifiers trained for printed character recognition on the NIST database. The system is simulated by reading some text character by character and replacing the character by a randomly chosen classifier output vector for that character. The output vectors are normalized to sum to one. The stream of classifier outputs is used in the HOVS recursion, together with a source model previously generated from another text, to generate a stream of "corrected" characters. Performance is measured as the percent correctness of the output stream.

A variety of source models were used. Straightforward $n$-gram models, $n = 1 \ldots 5$, were constructed using all the $n$-grams of that length occurring in the text used for source modeling. A "no-context" model was built, for which $p(character|context)$ is the *a priori* probability of the character. I constructed two "word-based" source models, in which a single context is a word prefix together with a fixed number (1 or 2) of characters from the preceding word. The source model consists of all such contexts found in the training text. Finally, the remaining models were constructed using variants of the Rissanen algorithm [Ris86]. This algorithm has parameters that affect its finite-sample behavior; by varying these I constructed a number of models with differing numbers of contexts. From these models more were constructed by trimming their context trees in two ways. In the basic algorithm, contexts that were not marked as active, but who have an active sibling, are made part of the active fringe. In one variant, such contexts are replaced by their parent. In the second variant, while growing the context tree, contexts were not allowed to cross word boundaries. Both these variants turn out to reduce the number of contexts, while tending to slightly improve performance on the test sets.

Unless stated otherwise, results are the average over five different classifiers and three test sets. The first 5000 lines of Moby Dick [Mel96] are used for source modeling. A "training set" consisting of lines 4900 to 5000 of Moby Dick (65044 words, 357745 characters) was used to evaluate performance when the source modeling process has "seen" the actual source. The three test sets are lines 10000 to 10100 of Moby Dick (1245 words, 7179 characters), lines 4900 to 5000 of Lord Jim [Con83] (1209 words, 6612 characters), and lines 10000 to 10100 of Lord Jim (1232 words, 6353 characters). The training and test sets have been chosen to represent "general English", one expects contextual information to be more valuable in more restricted domains, where the source is more constrained.

In stack decoding, paths are explored incrementally; at each step of the algorithm, the best path to pursue next is popped off the stack, and its one character extensions then pushed back on. For stack decoding, the total stack size allowed 32K paths; when this was exceeded, the stack was trimmed to the top scoring 3200 paths.

For exhaustive matching, the $N$ most common words from all of Moby Dick were used, where $N = 4096, 8192, 16384, 18004$ (18004 is the total number of distinct words in Moby Dick). Results are averages over all five classifiers and over the training and all three test sets.

Figure 2.2 shows the results of a number of such experiments comparing HOVS, stack decoding and exhaustive matching for a variety of source models. The figure does not include any results from using the FSM approximation to the source model, these were found to vary very little (less than 0.5% reduction in error) from their original model. The x-axis is the number of contexts used in the source model, or, in the case of exhaustive matching, the number of "context equivalents". For these experiments, only lowercase and space characters were used, so an implementation of a context minimally contains

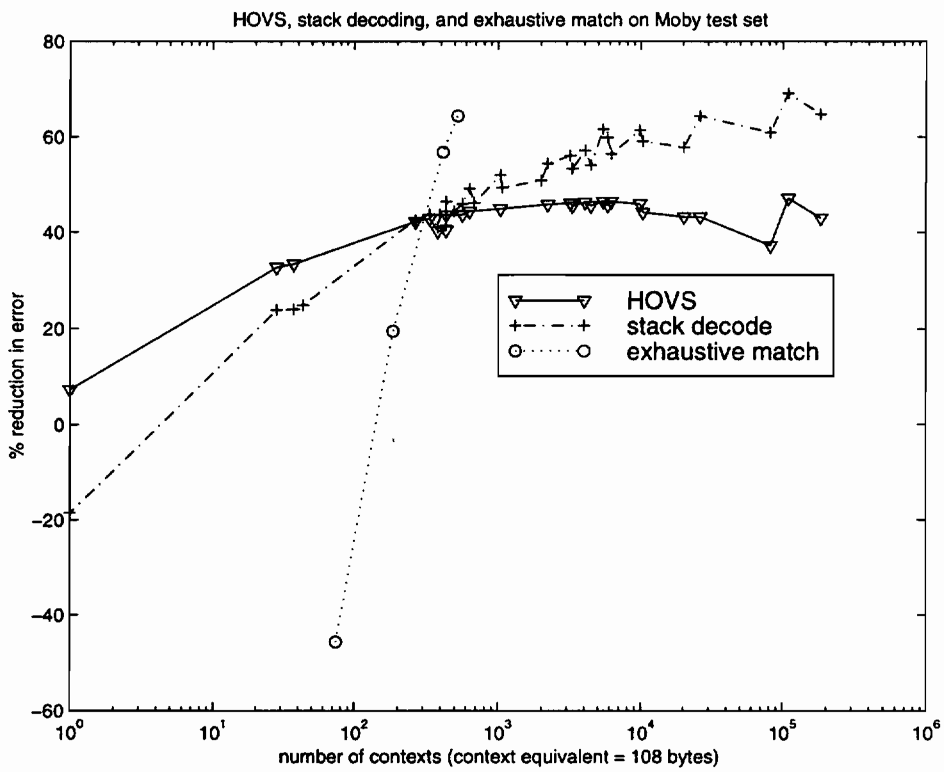


Figure 2.2: Comparison of HOVS, stack decoding and exhaustive match

27 probabilities $p(\text{char}|\text{context})$ and 27 context identifiers for the next context given the next character; allotting 4 bytes for each implies 108 bytes per context. For exhaustive matching, the number of bytes needed to store all the models was divided by 108 to give the number of context equivalents needed to store them. The y-axis is the percent reduction in error from using only the classifier output scores, with no context information, computed as $100 * (correct_{context} - correct_{raw})/(1 - correct_{raw})$, where $correct_{context}$ and $correct_{raw}$ give the fraction correct with and without the use of context. Use of this metric normalizes for the variation in $correct_{raw}$ among the different classifiers. In these experiments $correct_{\mathrm{raw}}$ was typically around 80%, so an error reduction of 40% corresponds to 88% correct; an error reduction of 70% corresponds to 94% correct.

Certain points on the graph require discussion. For no contexts and for about 30 contexts, HOVS outperforms stack decoding. In part, this is due to extensive stack trimming which takes place for these less context-restricted searches. Also, the source model for no contexts uses the *a priori* probability of the character as its prediction for $p(char|context)$. As the prior may underestimate the contextual probability, this is not an admissible heuristic for the stack decode (A*) algorithm [Nil86], further compounding its poor performance. At the other end of the graph, the source models for the two largest numbers of contexts have good performance relative to the third largest source model. The last two points correspond to the "word-based" source models described above, with one and two characters of pre-word context allowed, while the third largest corresponds to a 5-gram source model. The latter allows much inter-word context to be captured, and it appears that this does not generalize well. This is corroborated by figure 2.3 which shows that the 5-gram model has extremely good performance on the training set, and that among the largest models, better performance on the training set "mirrors" poorer performance on the test sets. In figure 2.2, the data points for exhaustive match, corresponding to 4096, 8192, 16384 and 18004 word models, or from about 80 to 500 context equivalents, show that the performance of exhaustive matching is quite sensitive to the number of words in the source that are missing from the source model. We also see that something like 10,000

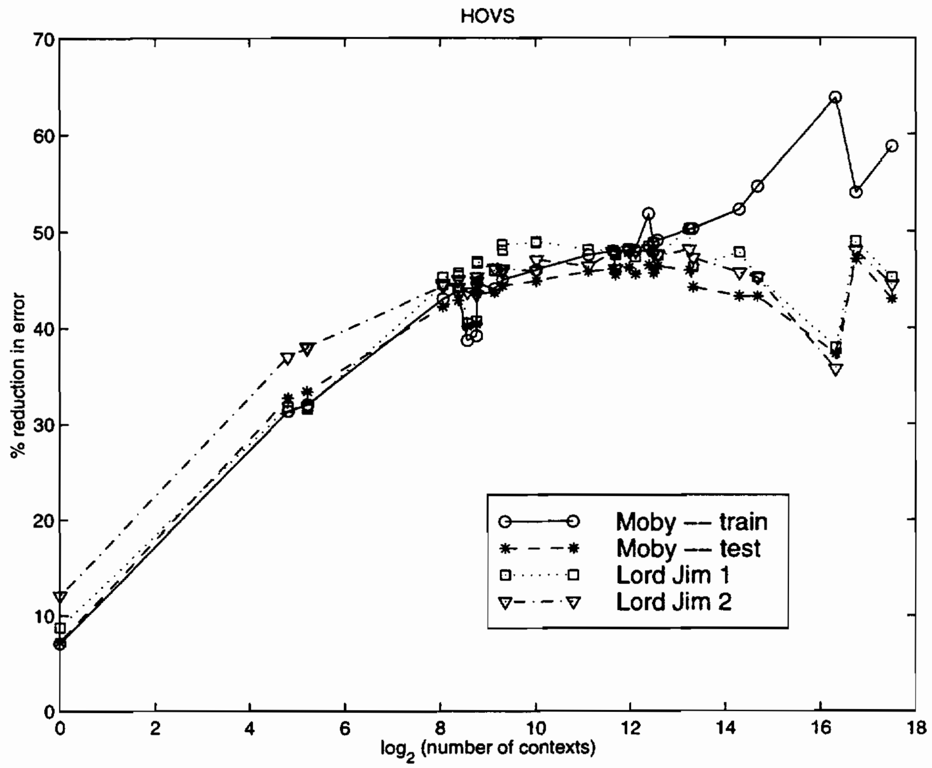


Figure 2.3: Performance of HOVS on training and test sets. Here "Lord Jim 1" and "Lord Jim 2" are test sets consisting of lines 4900-5000 and lines 10000-10100 of Lord Jim, respectively.

to 14,000 word models are needed to duplicate the performance of correspondingly sized HOVS or stack decode algorithms.

Figures 2.2, 2.3, and 2.4 show that, unlike stack decoding, the HOVS algorithm cannot make much use of larger amounts of context. Figure 2.4 compares the performance of HOVS and stack decode on the training set, where generalization is not an issue, and shows little performance gain for HOVS after a few hundred contexts, corresponding to an average context length of two to three.

This can be explained as follows. The effect of differing context probabilities is felt in the argmax operation; when the best predecessor to a given state is being determined,

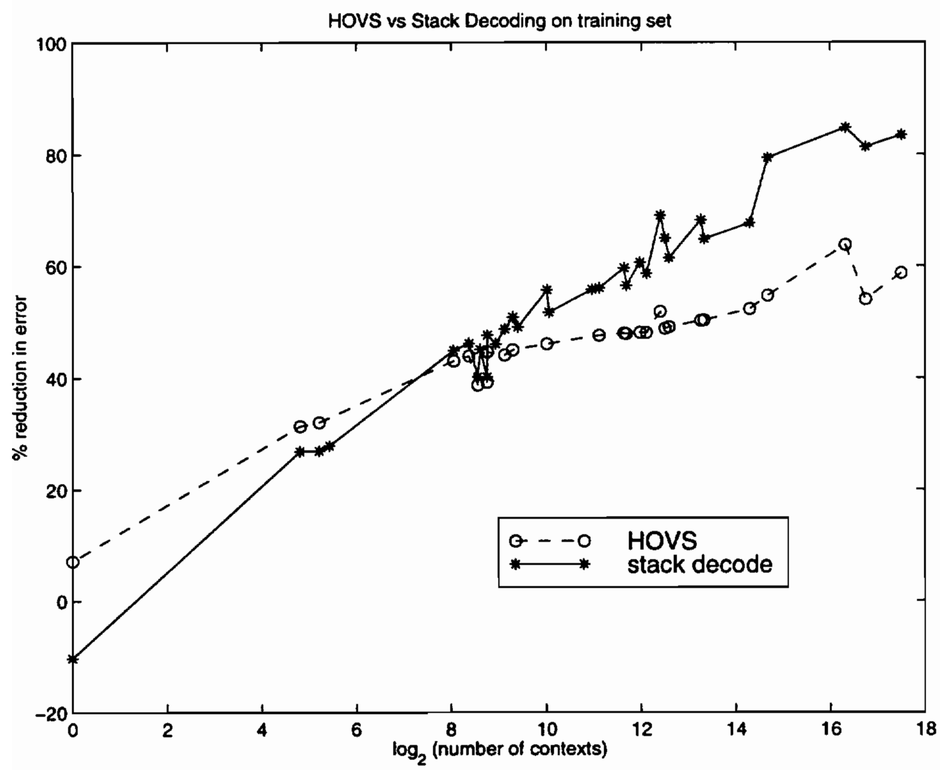


Figure 2.4: Performance of HOVS and stack decode on training set

Suppose two possible predecessor states, $h_t^1$ and $h_t^2$ of a given state, $h_{t+1}$ are being considered, each with a best path to it, and that two source models, $\mathcal{M}_{\text{short}}$ and $\mathcal{M}_{\text{long}}$ are being compared, one of which is shorter, in that its contexts are suffixes of the contexts of the other. Assume that the best path to each node is the same for both source models, when HOVS is used. The model with longer contexts will provide improved performance for this comparison only if the differences in the context probabilities for $h_t^1$ and $h_t^2$ differ sufficiently between the two models to outweigh the difference in observation probabilities between the two states.

For stack decoding, all possible previous paths to a state are "in use", so the current observation may, via longer contexts, affect which is the best path to the next state. The best path may change back in time, up to the length of the context. Conversely, for HOVS, the previous path to a state is fixed, and longer contexts can only affect outcomes better than short contexts to the extent that they give better probability estimates than the short contexts, and estimates that are sufficiently different to overcome the difference in observation probabilities for the two states.

So the inability of HOVS to take advantage of longer contexts corresponds to the notion that, for the texts and classifiers considered, contexts longer than about 3 characters provide probability estimates that differ from those of their 3 character suffixes by amounts that are small compared to the typical difference between observation probabilities at a given time. This suggests two methods of helping HOVS take advantage of longer contexts: (i) rather than just one, keep track of several "best" paths to a state, and take the argmax over all of these, and (ii) dampen the dynamic range of the classifier outputs. I discuss these further in the section on future work.

### 2.5.1 Discussion

Figure 2.2 shows that, for my simulations, HOVS, exhaustive matching, and stack decoding give approximately equivalent performance for about 300 contexts (or 10,000 to 14,000 word models). This is about the same context storage size for HOVS and exhaustive

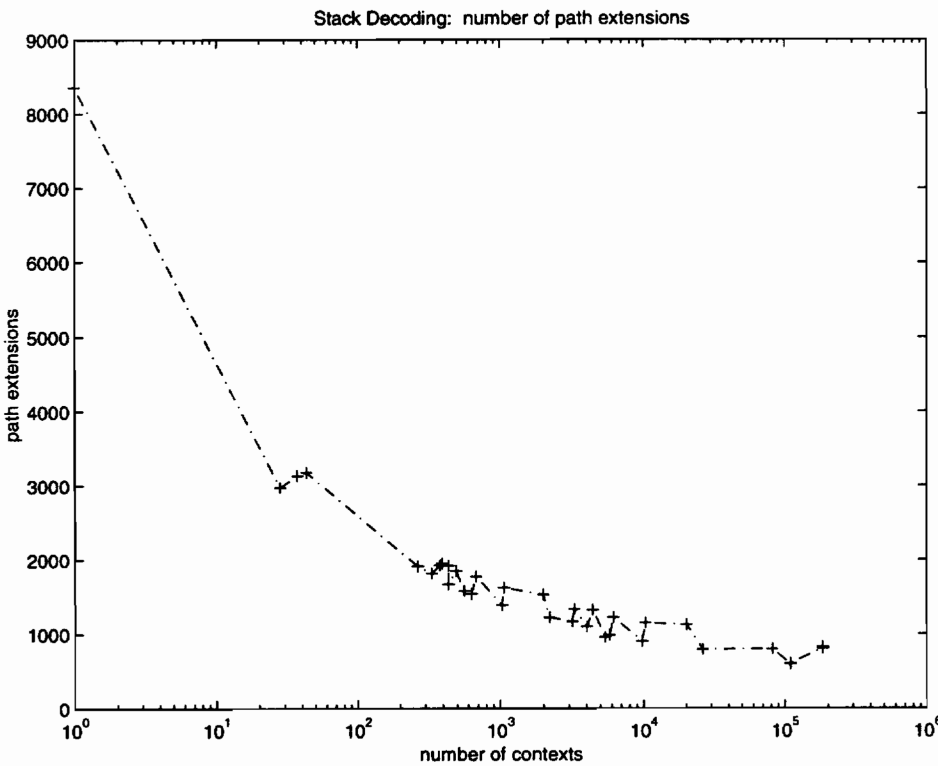


Figure 2.5: Average number of state sequence extensions per word, for stack decoding.

matching; stack decoding also requires a substantial stack (128 KB in my simulations). In terms of per-word time complexity, let $W$ be the length of the word, then HOVS is $O(W \cdot |H|^2)$, and has parallel versions that are $O(W \cdot |H|)$. Exhaustive matching is $O(W \cdot \textit{(number of words of that length)})$, and is trivially parallelizable. The complexity of stack decoding is more difficult to determine. Figure 2.5 shows the average number of state sequence extensions explored by stack decoding *per word* for the various models, averaged over classifiers and test sets. For around 300 contexts we see that an average of about 1500 extensions were explored. Other experiments with very accurate heuristics, using the Huang-Soong algorithm [SH91] obtained at best an average of about 300 extensions per word. The heuristic is constructed by a full forward pass through the Viterbi algorithm, with complexity $O(W \cdot |H|^2)$. In the following stack decode, each extension requires insertion of the new score into a priority queue, an expensive operation that is at least $O(\log |H|)$, with a substantial constant factor. I conclude that stack decoding is substantially slower that HOVS; further, it does not parallelize[6]. Although not simulated, the $n$th-order version of the Dawid algorithm is $O(W \cdot |H|^{n+1})$ and so much more expensive than HOVS for $n > 1$. I conclude that HOVS is fastest at the accuracy and storage equalized point of about 300 contexts (context equivalents). Figure 2.2 also shows HOVS has superior performance for smaller numbers of contexts. For larger numbers of contexts, stack decoding outperforms HOVS, and the tradeoff between increased accuracy and decreased speed must be made on an application-specific basis. The same is true for HOVS compared to exhaustive matching with more than 10,000 to 14,000 word models.

## 2.6 Related work

I have already mentioned the relation to the acoustic mode / language model decomposition used in speech recognition.

[6]One can parallelize the computation of the heuristic, but then the computed heuristic, which is essentially the entire Viterbi lattice, must be communicated to the stack decoder, and this wipes out any performance gain.

In their work on "connectionist speech recognition", Bourlard and Morgan [BM94] have done extensive work on using classifier outputs as HMM observation probabilities. Hovever, their formalisms have not included higher- or variable-order methods like HOVS.

Ron, Singer and Tishby [RST94] have explored the use of variable order models in classification, but their formulation assumed no classification error.

Typical work in OCR uses dictionary lookup, i.e. exhaustive matching, for contextual information. However, He [He91] has looked at HMM-based methods for handling segmentation issues.

## 2.7 Future work / questions

It would be of interest to implement the higher-order Dawid algorithm (2.7) and its HOVS version (2.6), to look at how much performance is lost by use of the simpler model. For the parallelized version, the simpler model involves recursions and storage that are $O(|H|)$, whereas for the 2nd-order one, the complexity is $O(|H|^2)$. It seems unlikely that the performance increase of using the higher-order versions would warrant the extra complexity for most applications, but some idea of the performance gain would be helpful.

The classifiers used were trained for classification performance, and hence their outputs are interpreted as probabilities. It would be of interest to determine to what extent training the classifiers to estimate the mutual likelihood, $m(x;y)$, helps. I have not done so as initial experiments indicated that this is unlikely to make much difference. Specifically, initial experiments showed that mis-estimating the class probabilities $p(h)$ used to convert classifier outputs $p(h|x)$ to $m(x;h) = p(h|x)/p(h)$ had only a small effect on the total performance.

As mentioned, the HOVS recursion (2.5) is easily extended to one where some fixed number, $n$, of best paths are used, rather than the single best. Also, the SIMD implementations can easily be extended to this case. The time complexity and number of recursion variables will increase by a factor of $n$, while the space for context memory remains the same. It would be of interest to implement this, see its performance, and, in particular, to see to what extent it allows the HOVS algorithm to make better use of longer contexts, as described above.

The HOVS recursion (2.5) combines the classifier outputs and context probabilities by multiplying them. Thus if the classifier outputs erroneously give a very small value to the correct state, it is hard for the context probabilities to counteract this; they must give equally small values to the incorrect states that the classifier prefers. There is thus an "impedance matching" problem between the classifier outputs and the context probabilities. It would be of interest to experiment with a tuning parameter that smoothed one of these, say the classifier outputs, $p(h|x)$ perhaps along the lines of

$$p_\beta(h|x) \doteq \frac{p(h|x)+\beta}{\sum_h (p(h|x)+\beta)}.$$

However, there is evidence that smoothing the classifier outputs does not help performance [Mor]. Another possibility is to train the classifier either to $p(h|x)$ or $m(h;x)$ using a smoothing regularizer [MR97]. As mentioned above, decreasing the dynamic range of the classifier outputs also might allow HOVS to better use longer contexts.

The effects of context will be stronger in a more constrained domain; it would be good to see experimentally if any of the qualitative behaviors change.

My notion of context is that of a source model, and is unrelated to error model notions such as handling insertions, deletions, and segmentation errors. Incorporating such error model information is important to any real system, but it seems completely separate from the present framework.

Another notion of "context" is a more semantic or discourse-level one, where a context is something like a probability distribution over the set of word models giving the current likelihood of seeing a model. Finding a way of computing and updating that distribution would might allow many models to be disregarded, allowing either better performance exhaustive matching, or construction (or lookup) of a more constraining source model.

## 2.8 Conclusions

HOVS provides a fast, practical, close to optimal approach to using the context provided by a single, large recursive model to constrain the interpretation of sequences of observations, as given by a probabilistic classifier. Alternative approaches based on the PIN formalism quickly become expensive as more contextual information is used. HOVS is an efficient principled approximation to PIN approaches that might otherwise be computationally intractable. For even better performance, HOVS allows an efficient SIMD implementation requiring only limited resources. The HOVS approximation framework is flexible, to accommodate differing computational resources , application characteristics and application requirements; one can "tune" the number of previous paths providing context, the (variable) length of context used, or the length of the state-tuple over which the recursion is performed. While much if not all of the underlying theory is already in literature, tying everything together into the PIN framework is novel, as is the SIMD implementation.

We thus see that, for ordered input, contextual information from a single large recursive model can usefully be used to constrain the interpretation of sequence of components. This form of contextual processing may be useful, for example, in recognizing novel words, however, when it exists, a known vocabulary will better constrain the allowable sequences. I turn next to this problem, of using the context provides by a known vocabulary of many (small) models, and to the *matching models* paradigm of contextual analysis.

# Chapter 3

# Models

In this chapter I discuss the feature grouping and model matching operations that form the stages subsequent to feature extraction, emphasizing the model matching stage. I wish to demonstrate a number of "stylized facts" about the algorithms used in these stages:

**F1** feature grouping (object hypothesis generation) involves construction and traversal of irregular data structures such as lists and trees.

**F2** model matching algorithms exhibit irregular control flow mediated by the data and/or model

**F3** most model matching and feature grouping algorithms are simple, with computational complexity coming from applying a small code "kernel" many times

**F4** model search and indexing techniques are "imperfect", in the sense that they generally restrict the number of models to be matched, but not to a single candidate.

## 3.1 Irregularity

Essentially, *irregularity* is inefficiency due to data dependent execution. Operationally, for parallel algorithms, it can be viewed as a problem of load balancing: ensuring that work is apportioned between processors so that all are kept busy. For algorithms such as branch and bound that cycle between stages of computation and scheduling (load balancing), one can formalize irregularity as the ratio of computational work to task scheduling work [GRV95].

At a more detailed level, suppose the workload consists of a set of indivisible tasks, each task consisting of applying identical code to some task-specific data. Processors may take variable amounts of time to accomplish such tasks, leading to load balancing problems, for several reasons:

1. most simply, the processors may have been allocated different numbers of tasks (*workload irregularity*);

2. a task may involve construction or traversal of data structures whose size and/or "shape" is task dependent, for example, lists and trees, leading to data dependent control flow (*data structure irregularity*);

3. processing in a task may involve a variable, data dependent, number of iterations, even for data of the same size and "shape", for example, in relaxation and search algorithms (*convergence irregularity*);

4. as opposed to irregularity due to the task, as in the preceding, there may be irregularity due to the implementation on a particular machine: variability in communication times to other processors, to memory, or to I/O devices (*communication irregularity*).

Irregularity is important insofar as it impacts cost-performance. Using a task scheduler requires writing and executing more code, as well as relocating tasks among processors, affecting implementation cost and runtime performance. For performance, tasks must not be too small, lest the scheduling and relocation overhead dominate. Hiding communication irregularity by multi-threading requires more (and more complex) code, and perhaps more hardware for fast context switching. At the processor level, hiding variable memory latencies, variable instruction execution times, and branches (or branch mispredictions) by out-of-order and speculative execution can be quite expensive in terms of chip area and complexity; I discuss this in chapter 4.

Irregularity is especially pernicious for SIMD execution, as lockstep execution generally implies that whenever one processor experiences inefficiency due to workload, convergence, or data structure irregularity, all processors do. SIMD execution disallows using multi-threading to hide communication irregularity, in fact, it generally requires that communication irregularity be avoided altogether. This forces every communication to allow for worst-case timing, and to take the worst-case time.

I will consider the subjects of irregularity and parallelism as I examining the algorithms of the feature grouping and model matching stages.

## 3.2 Feature extraction and grouping

In this section I restrict myself to vision. Feature grouping, *per se* refers to constructing sets of features hypothesized to belong to the same object. In speech, due to its one-dimensional nature, this consists of hypothesizing temporal boundary points between objects (phonemes, words, sentences, and so on). The boundaries tend to have distinctive acoustics, and hence can be modeled as different kinds of components (e.g. "silence"). Thus the boundary determination can be made part of the model matching process, and a separate grouping stage is unneeded.

In vision, however, determination of which components (may) belong to the same object is a central problem, sufficiently complex and distinct to require its own processing stage[1]. In fact, the general model of this *perceptual grouping* [Low85] stage is a hierarchical construction of increasingly complex features formed from sets of more primitive ones: pixels form "edgels", which are linked into lines, which make up parallel line pairs, which form rectangles, and so on. As this suggests, processing involves a series of stages, with changes in representation between stages [Rei91, Ger92]. Further, groups may contain features that are widely separated in the image: consider grouping edgels into lines or lines into parallel pairs. Thus when the image is distributed over multiple processors,

[1] Integrating the feature extraction, grouping and matching phases into a single (recursive) phase has been proposed, see, for example, [BCK92].

there may be significant data movement between stages as well [Rei91].

The representation of features may be complex. At the lowest, *iconic*, level of the hierarchy, features are associated with individual pixels, being properties of the pixel or of its neighborhood. Examples are pixel properties such as grayscale value, and neighborhood properties such as values of 2-D linear or nonlinear filters, or correlation with a small template. Feature extraction involves mostly neighborhood operations that can be performed in an SIMD fashion, what complexity there is coming from the treatment of pixels lying at the boundaries of the sub-images alloted to individual processors. At higher, *symbolic* levels of the hierarchy, features are complex representations of boundaries or regions, such as chain codes, Fourier descriptors, and quad-trees or are properties of these, such as curvature of a boundary or moments of a region [GW92].

The process of grouping involves finding sets of features having common or related properties, and amounts to associative lookup by attribute-value predicates [BDBH89, CEJK92, Ger92, Rei91]. The allocation of features among processors changes as more complex features are constructed, so doing associative lookup in parallel requires first redistributing the features among the processors, to balance the workload. Since the allocation of features to processors, and hence the necessary rearrangement, is input-dependent, this redistribution must be done at runtime, generally as a separate, intermediate, stage [CEJK92, GO92, Rei91] .

I study example algorithms for this stage, to understand the forms of irregularity involved. I exclude communication irregularity for now, as it is more a property of the processing architecture than of the specific algorithm.

One basic grouping operation finds the *connected components* of the image – maximal contiguous areas of pixels all having some specified property. A refinement of this is *region growing*, which finds contiguous areas of pixels that have "close" values of a property. There are a variety of algorithms for this (see [AP92] for a recent review). One class of algorithms involves only pixel-local operations, and are easily done in an SIMD fashion, however it requires multiple passes over the image, and storage of multiple copies of the image. The second major class of connected component algorithms is of the *divide and conquer* form, where images are recursively subdivided, the components of the sub-images are found, and then components intersecting a boundary of a given sub-image are merged with those on the matching boundary of the adjacent sub-image. The latter step is done by merging graphs representing the boundary connectivity information, and then finding connected components of the merged graphs. The third class involves finding boundaries explicitly, then "filling in" the enclosed regions. This technique uses traversal over boundary lists, together with local pixel operations. We see that the latter two classes involve both data structure and workload irregularity. Workload balancing by redistributing features among processors is problematic as, in any case, the labels determined from features must be propagated back to the pixels, which presumably have not been redistributed.

Another major grouping task is constructing edges from "edgels", the results of primitive edge finders such as the Canny [Can86] or Sobel [GW92] operators. Again, a large number of algorithms have been proposed; I look at some examples. Wang and Binford [WB94] use a highly accurate edgel finder, then link edges into "inner curves" based on a fixed, quadratic, curvature hypothesis, together with an error model of the edgel finder. Branches are treated by terminating inner curves at the branching, then, in a second phase, linking conjoint inner curves that have similar measured curvatures. Fischler [Fis94] deals with noisier images by a multistage procedure involving clustering, representation of hypothesized groups by minimal spanning trees, and resolution of the hypotheses by dynamic programming. Farag and Delp [FD91] use a general A* algorithm, with a specific evaluation function for linking in a new edgel. We see a spectrum of irregularity here, the Wang-Binford algorithm involving workload irregularity, the Fishler algorithm having both workload and data structure irregularity, and the Farang-Delp algorithm having convergence irregularity as well.

As mentioned above, at higher levels of the feature hierarchy, where the location of individual pixels plays no part, grouping can be viewed as associative lookup by attribute-value relationships. The example given was finding rectangular structure by first grouping into parallel lines, then into U-shaped structures, then into rectangles, using values of the attribute "angle". The associative lookup can be implemented by exhaustive search for features satisfying a given predicate, or by first building indices. In either case, workload can be balanced by reallocating features equally among processors and broadcasting the request. Data structure irregularity occurs from list and index traversal.

Finally, both for the consistency checking of hypothesized groups, and for calculation of further properties such as line curvature and moments of functions over regions, various functions must be mapped over representations such as lists and quad-trees. Workload can be balanced by reallocating features, but data structure irregularity remains.

In summary, we see mainly workload and data structure irregularity at the feature grouping stage. Time spent reallocating features between stages of the grouping process can be hidden by use of an asynchronous autonomous network, where features are redistributed for the next stage as they are computed, with computation and redistribution being overlapped [CEJK92]. Optimally, such a network should support 2D-local (nearest-neighbor), tree-structured, butterfly (permutation) and broadcast communication. [GO92, CEJK92].

Further, data structure irregularity can be hidden by balancing workload among processors based on estimated per-feature processing time rather than simply the number of features. Note that this is difficult for SIMD architectures, as task execution time depends both on the task and on which tasks are done simultaneously. All forms of irregularity can be ameliorated by averaging out variation: assigning more features per processor (and hence using fewer processors). Sub-image boundary effects require *ad hoc*, data-dependent communication; this may be hard to hide, but allocating larger sub-images per processor will tend to reduce the problem. As a final point, we see that once features are allocated, the basic algorithmic structure is that of mapping a small kernel, identical between processors, over a list or tree-like data structure.

Another point is that many feature construction algorithms are amenable to a *split and merge* technique [Web92], wherein the image is blindly split into tiles according to location, the tiles are processed independently, without communication, and then the results are merged. Communication occurs in phases: partial features are computed independently on each processor, then the partial features are recursively merged with those of other processors into larger and larger parts until the entire feature is assembled. This phasing of communication will be relevant for the SFMD computation model introduced later. Here, note that one characteristic of the split and merge technique is the need for relatively large processor memories to hold and combine final results [Web92].

So the algorithms of the feature extraction stage suggest the utility of certain hardware resources. An asynchronous autonomous network helps manage workload irregularity, by allowing overlapped computation and communication when load balancing. Fewer PEs, with large memories per PE, reduce data partitioning boundary effects, allow averaging out of variation and permit use of split-and-merge techniques. Non-lockstep (non-SIMD) execution more efficiently processes irregular data structures and allows better load-balancing and averaging out of variation.

## 3.3 Matching techniques

I examine some model matching techniques in detail. A fundamental division exists, based on the dimensionality of the input.

In what might be called "zero-dimensional" input, there is no context because there is no relation between inputs. An approximate example is olfactory perception, although even here there may be "priming" effects based on temporal ordering.

One-dimensional input is ordered, generally temporally. Examples include speech and text recognition. Ordering leads to use of a Markovian prefix-based source model, and path-discriminatory model matching based on a tree of prefixes (see below). Context is provided mainly by the immediate predecessors of the current input, possibly at several levels of granularity[2]. Although I discussed one-dimensional input in chapter 2, my concerns here are different. There, a set of models was used to construct a single source model, which was applied to improve the classification of individual components. There the issue is matching the individual models, the goal being selection of the most likely model.

Higher-dimensional input has no natural ordering, so Markovian source modeling is difficult, as determining or even defining the current state is hard: there is no distinguished set of already identified features (like immediate predecessors, for 1-D input). In vision, the basic example of higher-dimensional input, geometric, topological, and other forms of relations between features give rise to context defined by inter-component relations on a model-by-model basis.

### 3.3.1 Ordered Input (Markov models)

With ordered input, as in speech or text processing, the ordering of the input leads to an ordered match, hence to an implicit or explicit ordering on a graph model. In practice, this leads to the use of Markov models, where nodes represent states, arcs represent transition probabilities between states, and the match score can be calculated using dynamic programming (Viterbi or trellis search). The matching of separate models is synchronized by all models receiving the $t$'th input at the same time.

Markov models can be recursively combined by adding artificial source and sink nodes to the graph, and then pasting together models by identifying the source of one with the sink of the other. Frequently, at the lowest level of model, phonemes or characters, the graph structure of the model is the same for all, thus facilitating SIMD execution with individual processors matching different, but identically structured, models. Higher level models generally do not have identical structures.

[2]Occlusion may be a problem, but is usually not part of the formalism of the recognition process

### Hidden Markov Models

In the most common case, Hidden Markov Models[Rab90, He91], the models are generative, and matching finds the model most likely to have generated the observed sequence of inputs. If $a_{ij}$ is the transition probability from state $i$ to state $j$, and $b_j(O)$ is the probability of observation $O$ having been generated by state $j$, then

$$P(\mathit{observation\ sequence}|\mathit{model}) = \sum_j \alpha_T(j)$$

where

$$\alpha_{t+1}(j) = [\sum_i \alpha_t(i)a_{ij}] \cdot b_j(O_{t+1})$$

can be defined as the joint probability of the state at time $t+1$ and the observations up to that time,

$$\alpha_{t+1}(j) = Pr(\mathit{state}_j, O_1, \ldots, O_{t+1}).$$

This calculation involves computing $b_j(O_t)$ for each node at each time, and propagating values over arcs. (The formula above gives the *trellis algorithm*, replacing $\sum_j$ in the $\alpha$ recursion by $\max_j$ gives the *Viterbi* formulation.) The essential point is that the recursive nature of the equation for $\alpha_t$ means that an efficient dynamic programming implementation is possible.

### Model-discriminant versus Path-discriminant Matching

In some common cases, there is a natural way to combine a set of Markov models into a larger one. Assume that no two models contain identical paths from source(s) to sink(s), so that no two models have identical scores when some path of components occurs in the data, and hence that models are always distinguishable. Then models can be combined to form a *lexical tree* as follows: add an artificial root node to the start of each model, replace each model by the set of paths through it, and then form the tree by merging common prefixes of paths. A path from root to leaf in the tree corresponds to that same path in some model, and a Viterbi [3] match will give the same score. By labeling the leaf with the model name, finding the best matching path in the tree is equivalent to finding the best matching model. Following He [He91] I refer to matching a path in the tree as *path-discriminant search*, and to matching models individually as *model-discriminant search*, the basic vision paradigm.

### Reducing matching complexity

The construction and use of a lexical tree greatly reduces average model matching complexity, as the prefixes of many models are evaluated simultaneously. Ney [NHUTO92] reports that for a vocabulary of 10000 German words, the use of a tree reduces size by a factor of 2.5, and reduces search complexity by a factor of 7. The large latter reduction comes from the fact that most of the search complexity occurs in the prefixes of the words, which are only evaluated once in the tree implementation. By viewing the problem of finding the highest scoring model as one of search, using a lexical tree may avoid matching most models except for some small initial prefix. The tree search is typically formulated as either a beam search [Low90, NHUTO92] or as an A* search [Jel69, BJM90, SH90, KHG+91]. Beam searches maintain a "beam" of current candidates, pruning out those that fall below some threshold. A* searches operate in a "best-first" manner, using an evaluation function for choosing what path to extend next. If the evaluation function overestimates the score of the path to be extended (when the search is for the largest score), then it is said to be *admissible*, and the search is guaranteed to find the highest scoring path through the tree [Nil86]. A* searches must thus maintain a search frontier that includes some prefix of every path, to allow that path to be followed if all others prove inferior. Beam searches, by pruning paths once and for all, are not guaranteed to produce the optimal path.

These search methods are generally good at rapidly eliminating most models from consideration. For a large vocabulary problem (typically, thousands of models) the tree

[3] A trellis match will not generally give the same result, as alternate paths through the same model will support one another in trellis search, but be independent in the Viterbi and lexical tree methods.

is large, hence it is built dynamically as part of the search process. However, even with sophisticated pruning, the dynamically constructed tree is large. For example, the beam search in [NHUTO92] keeps 50000 current states (tree nodes) out of a possible 650000, while A* searches generally require an exponential amount of state.

Much work has been done to reduce matching complexity by initially screening out unlikely candidates using a coarse *fast match*. Bahl, *et al*, [BGKN89] provide a vectorizable algorithm that is admissible as an A* heuristic. In an experiment using the non-vectorized form, it provided a 2x speed improvement. Kenny, *et al*, [KLL+93] give an admissible fast match based on Viterbi search through graph representing triphone constraints implied by the lexicon, rather than the entire lexicon itself. Gillick and Roth [GR90] give an inadmissible algorithm that does hard pruning based on whether a given prefix passes a threshold; this is applied to a small ($\approx 1K$) vocabulary giving a 25x speed improvement.

Alternatively, for word-spotting tasks which typically have smaller vocabularies, and may have the words in their vocabulary chosen to be easily distinguished, thus having fewer shared prefixes, it makes sense to match models individually. This also allows the use of the somewhat more accurate trellis matching.

### Use of parallelism

In large vocabulary tasks, the various search methods rapidly eliminate models. So, attempts to parallelize the matching process by matching all models in parallel, rather than using a search technique, will probably not be cost effective. Conversely, the use of a large, dynamically constructed, global tree is difficult to implement effectively using parallelism. A static partitioning of the tree among the processors makes it likely that after a few steps, most processors will have only pruned parts of the tree, and hence have no work to do (see [Sto87] and [SB88] for interesting examples of this phenomenon in text searches). Using a "master-slave" approach by having one processor control the search process, expanding the search frontier at multiple places in parallel by distributing the expansion task to other processors, is unlikely to succeed as the amount of work in expanding a frontier node is small compared to the amount of information that must be communicated for that work to be possible. Dynamically repartitioning the tree suffers from the same problem: because of the unpredictability of the part of the tree examined by the search process, repartitioning is frequent, hence little work is done in proportion to the amount of communication necessary. So large vocabulary tasks probably cannot make cost-effective use of parallelism in the model-matching (i.e., decoding) stage[4] .

For a small vocabulary application like word-spotting, parallelizing is possible, but may not be needed as the task is small. So the actual model matching process for ordered input is an unlikely candidate for parallelism. As mentioned above, however, ancillary processes like a fast match may be parallelizable. However, in this case Amdahl's law [Amd67] implies that, as the central search part of the algorithms requires substantial computation, the speedup obtained by parallelizing ancillary processes will be slight. So, in general, I see little opportunity for useful parallelism in model matching one-dimensional input.

### 3.3.2 Unordered Input (Graph Models)

With unordered input, as in vision tasks, models are undirected (hyper)graphs, where nodes are features and arcs are relations or constraints between feature values. Models are thus *constraint networks.* Typical arcs are *geometric* and *topological constraints* relating features' adjacency, locations, angles, distances, and so forth. Geometric constraints depend on a *rigidity assumption*, that the object is rigid and the geometric relations remain the same over time. This is appropriate for a large class of machine vision tasks such as industrial part recognition. Bolle [BCKM90] gives examples of some non-geometric, non-rigid, constraints such as mutual visibility. Matching proceeds by positing a partial correspondence between input features and model components, and then checking constraints. There are typically more features in the input than components in the model, and features correctly associated with model components may be occluded or otherwise

[4] Of course, there is plenty of opportunity for parallelism in the feature extraction stage.

image $\rightarrow$ *select ROI* $\rightarrow$ *group* $\rightarrow$ *match* $\rightarrow$ *verify*

Figure 3.1: Stages of visual recognition systems

missing in the image. Thus, when the arc relations are viewed as binary, yes/no relations, the matching procedure is a double subgraph isomorphism problem, and hence, in general, of exponential complexity.

Visual recognition systems generally have the sequence of stages shown in figure 3.3.2. A *region of interest (ROI)* is selected from an image, features within the ROI are grouped to form object hypotheses, the groups are matched against models to establish correspondence between image features and model components, and to establish the pose of the object, and then the match is verified.

### Verification

Verification of a match is usually done by using the model and pose to predict the points of the corresponding object within the image (*back-projection*). The back-projected model points are then compared with the actual points of the image in some way. For example, using range images, Wheeler [WI95] matches each back-projected point to the nearest point in the image, using a *k-d tree.* Matched model and image points whose mutual distance lies within a tolerance are then used to compute some simple statistics such as the proportion of matched points to visible points, and if these statistics all are within tolerance, the model match is taken to be correct. A matched model may also be verified by predicting features and feature attributes, rather than points, see [FH86].

### Global Interpretation

A further stage in processing is to form a global interpretation of the ROI. The simplest and most common form of this is to ensure that image points are matched to only a single model, so that objects are not allowed to inter-penetrate. This is typically done in a sequential system by removing image points from further consideration as soon as they participate in a verified match. This, of course, also reduces the complexity of further matching. It does require a small amount of communication in a parallel implementation. More elaborate interpretations involve the mutual likelihoods of observing objects with certain properties in the same scene. One formalism for this is Bayesian networks; see [USA94]. Although it is intimately related to context at the level of the semantics of objects, I do not consider global interpretation formation further, both for lack of time, and, as mentioned before, as it seems unsuitable for parallelization.

For unordered input, there are five general classes of matching techniques: template matching, the (Generalized) Hough Transform, tree search, relaxation labeling, and elastic matching.

### Template Matching

In *template matching*, also called *correlation matching*, the template consists of a small image which is shifted over the image and matched pixel-by-pixel. The correlation between the template and the image at a given location forms the metric by which the match is judged. The correlation is the (squared) Euclidean distance between the template and the corresponding block of the image, so this can be viewed as a form of nearest neighbor matching. This form of matching is clearly susceptible to noise and occlusion in the image, rotation or scaling of the object, and for 2-D images of 3-D scenes, perspective distortions. It is thus mainly suitable for controlled viewpoint applications (2-D industrial). Template matching may be easily parallelized, in a SIMD fashion, and is also easily implemented in hardware (see, for example, [RV94]). The regularity of the algorithm, in fact, offers good performance on conventional machines, together with its suitability for controlled viewpoint and illumination situations as in industrial and medical applications, make it the most common form of object recognition. As its implementation is unproblematic, and as the model (template) makes no use of context, I will not discuss template matching further.

**Generalized Hough Transform (GHT)**

The Hough Transform was originally devised as a method of extracting lines from an image that is robust to noise and occlusion. As such, it forms groups of (disjoint) edge segments that potentially belong to the same line, and may be viewed as a mechanism for grouping pieces of an image that may be parts of the same object. The GHT [Bal79] is an elaboration of the Hough Transform that extracts arbitrary rigid 2D curves rather than lines, and may thus also be viewed as an object recognition, as well as grouping, technique.

Briefly, the basic Hough algorithm is as follows. Suppose we have $N$ input features, $s_i$, and $Mm$ total model components, $m_j$, where $M$ is the number of models, each with $m$ components. For each pair of features and components, $(s, m)$, there is a transformation mapping one to the other. This transformation (typically) consists of a translation, parameterized by $(a, b)$, a rotation, $R_\theta$, and possibly a scaling, $s$. The entire transform is then parameterized $(a, b, \theta, s)$. This is viewed as a point in $R^4$, and corresponds to a cell in a 4-D array constructed by quantizing the four transform dimensions, the *Hough accumulator.* In the recognition process, the transform is computed for each pair of scene and model features, and the corresponding cell of the accumulator is incremented (and the model noted). Finally, the cells are tallied. Large peaks in the histogram for the same model correspond to many scene features having (nearly) the same transform for that model, and hence to a likely occurrence of that model in the image. The location of the histogram peak gives the alignment of the model with the image, and the entries give the correspondence.

All features are matched with all components, so complexity is $O(NMm)$. Space is $q^d$, where $q$ is number of quantization levels of the parameters and $d$ is the dimension of the parameter space. Above, $d = 4$; other common transformations have $d = 2 \ldots 6$. GHT is known to be sensitive to sensor noise, scene clutter, and occlusion, generating many false positives [Gri90, BS92, SG93]. Grimson [Gri90] suggests its use as a filter, to select a smaller set of models for further verification. The 2-D Hough transform is widely used for line finding, and has an efficient SIMD implementation [FH89]; other forms, including GHT, do not seem to be much used, probably due to computation, storage and communication complexity. I will not pursue the GHT further, as it seems to be impractical.

### Relaxation Labeling

Another class of models are graphs where the nodes represent features and the arcs conditional probabilities. Christmas [CKP95] gives a recent variant. The algorithm is a somewhat more complicated version of a trellis Viterbi search, where time measures relaxation iterations rather than input ordering. Let $\theta$ be an input feature label, and $\omega$ be a model component label, then the probability at step $n$ that input feature $i$ has the same label (is matched with) model feature $j$ is given by

$$P^{(n+1)}(\theta_i = \omega_j) = (1/Z)P^{(n)}(\theta_i = \omega_j)Q^{(n)}(\theta_i = \omega_j)$$

where

$$Q^{(n)}(\theta_i = \omega_\alpha) = \prod_j \sum_\beta P^{(n)}(\theta_j = \omega_\beta)p(\mathcal{A}_{ij}|\theta_i = \omega_\alpha, \theta_j = \omega_\beta).$$

Here, $Z$ is a normalization factor, and $\mathcal{A}_{ij}$ is a binary inter-component compatibility constraint, providing "model context". The quantity $p(\mathcal{A}_{ij})$ is computed from comparison of measured values with model values, using an error model. Initial node probability values are determined by the measured values of unary attributes together with an error model. Note that in so far as the binary constraints $\mathcal{A}_{ij}$ have probability zero for features $\theta_i$,$\theta_j$ belonging to different models, the expression for $Q^{(n)}$ may be evaluated on a model-by-model basis.

The relaxation process combines evidence from measurements with prior model context. The use of measurement information throughout the relaxation process is intended to avoid the common criticism that relaxation methods are over-sensitive to the assignment of initial node probabilities (see, for example, [FH86]). In the examples given, components are line segments, the sole unary constraint is absolute orientation, and the four binary constraint types are metric relations such as angle and distance. One problem with the relaxation formalism is that all possible measurements must be made before the process starts, to relax over. Thus, the model does not direct the search process. Complexity for a single model and single iteration is thus $O(n^2m^2)$; where $n$ is the number of input features, and $m$ is the number of model features. In the implementation described, precomputing the binary constraint probabilities dominates time, requiring about five times longer than the relaxation process, *per se*. Also, the stored binary probability values use $O(n^2m^2)$ space. The relaxation process takes a variable number of iterations to converge, generally less than 5, but as many as 30, thus convergence irregularity is pronounced.

Other relaxation formalisms such as *Highest Confidence First (HCF)* relaxation over *Markov Random Fields* [CB90] have the same property that all possible model-scene pairings must be initially evaluated before relaxation takes place. HCF, however, is based on use of a priority queue. Initially, all nodes, each with all possible pairings, are entered in the priority queue, ordered by a measure of "stability". Relaxation takes place by changing the least stable pairing to be be more consistent with its neighbors, and reentering changed nodes in the priority queue. This typically leads to a consistent relaxed state by changing very few nodes, so that initially loading the queue is the dominant time.

### Tree Search

This class of model matching techniques, consists of various elaborations of *Interpretation Tree Search (ITS)*, due to Grimson [Gri90]. Interpretation Tree Search consists essentially of depth first tree search (*DFS*), where a node on level $d$ of the tree corresponds to pairings of image features with the first $d$ model features. The search is limited by a variety of unary and binary geometric constraints on the allowed pairings. Grimson shows that for a single isolated 2D object the (worst-case) search complexity is quadratic in the model size, while for a 2D object in a cluttered scene with possible occlusion, the complexity is exponential in the number of visible features:

$$o(m2^c + mn) < complexity < O(m^2 2^n + mn),$$

where $m$ is the number of model components, $n$ is the number of input features, and $c$ is the number of non-spurious input features, ones that correctly belong to some model. The complexity of determining that an object is *not* present in the scene is also exponential.

One form of elaboration on ITS is exemplified by *Local Feature Focus (LFF)* search, which adds model-based search control [BC82]. LFF is based on finding maximal cliques in a graph, and hence also relies on a constrained DFS. The name "feature focus" comes from the technique of identifying certain features as most search-constraining, and matching them first. The algorithm proceeds by first matching the focal features, then using the model to predict nearby features, and matching them using unary constraints. Thus in LFF the model has an active role in directing the search, while in ITS, it is used only to prune inconsistent interpretations, The model-image feature pairings generated are used to construct a graph of consistent pairings, and a maximal clique in this graph gives a possible model-image correspondence to be verified. Although worst-case complexity remains exponential, experimentally, the average case complexity is much reduced over ITS.

For rigid models, current versions of tree-structured matching use *alignment*. An alignment of an object model with image points is an estimate of the pose of the object. [HU90] show how to quickly compute an alignment given two oriented model points and two oriented image points, using linear, quadratic and square root functions.

Alignment may be used by itself as a matching algorithm, as follows:

- for each model do the following:
  - for each pair of model points, and each pair of image points, compute the alignment
  - verify the alignment by using it to predict image features corresponding to the aligned model

– (optionally) if verification succeeds, remove the matched image points from further consideration

Given $M$ models, each with $m$ points, and an image with $n$ points, this has complexity $O(Mm^2n^2v)$, where $v$ is the complexity of the verification phase. In [HU90], verification complexity is worst-case $v = m \log n$, giving an overall worst-case complexity of $O(Mm^3n^2 \log n)$. This a large overestimate of actual time: when an alignment is verified, no more pairs need be aligned for that model, and the image points may be removed from further consideration. [Ols93] gives simple lookup-based indexing techniques to avoid some unlikely or uninformative verifications, and gives results showing speedups of 20-150x when the model is not present in the image, and 5-10x when it is.

Alignment, or more generally, *pose estimation*, combines with tree search in an obvious way: after enough model-feature pairs are established by search, the pose of the object can be estimated, after which the location of the remaining input features can be predicted. The prediction of the feature locations is known as *back-projection* of the model onto the input; its noise sensitivity is analyzed in [TH91].

For alignment, *per se*, model size is small: for the basic form, storage for a model is just $m$ oriented points. The techniques in [Ols93] also require a 2-D index table per model, its size depends on a quantization parameter, no example is given, but it's likely small. It is necessary to identify "interesting" points (features) to use; [HU90] uses corners and inflection points as features, and report that feature extraction is as expensive as matching.

For tree-search as a whole, [BCKM90] gives further examples of constraints and their use to prune and control search. [RB93] apply ITS to match complicated models having repeated parameterized subparts, with non-uniform stretching and scaling. A real-valued constraint network is used in conjunction with ITS to determine values of model parameter values as well as satisfaction of geometric constraints. As well as pruning via constraints, tree search may be done using a branch and bound formalism [CHS91]. Table 3.3.2 shows some characteristics of several systems using tree search.

These tree-matching algorithms exhibit data structure irregularity in the (pruned) tree

Table 3.1: Some systems using tree search matching.

| | [FH86] | [FJ91] | [BH86] | [WI95] |
|---|---|---|---|---|
| features | planes (4), quadric patches (12) | plane, cylinder, sphere | three types of edges (3,4,6) | planar regions (4) |
| unary predicates | area, type, length | area, type, radius | edge properties | area, moments, dimension |
| binary predicates | relative location and orientation | parallel planes, simultaneous visibility | relative location and orientation | simultaneous visibility, relative orientation, max distance |
| matching | ITS | ITS | LFF | LFF |
| pruning | too many unmatched features | inconsistent pose | enough features for pose estimate | Markov Random Field |
| pose estimation and refinement | recursive least squares | least squares | - | gradient descent |

traversal, as well as convergence irregularity in the pose refinement phase.

### Elastic matching (deformable models)

*Elastic matching* is a technique used for non-rigid models. Matching occurs by minimization of an energy function, measuring the amount of deformation of the model required to match the object. Object classification can be done by viewing energy as a metric, and selecting the class with model "closest" to the object [Hin92]. Deformation matching builds a correspondence between model and object, so it can be used for segmentation and registration. The model does not direct search for a correspondence, as there is generally little predictability for non-rigid objects.

The definition of deformation energy varies, but is usually a function of Euclidean distance between object and model points, and sometimes also relative edge angle. These require scale, rotation and translation invariance, so pose must be estimated before measuring deformation. This is usually an iterative process, slightly deforming the model to agree with the input; estimating pose, which redefines the deformation energy potential;

re-deforming the model; and so on until some convergence criterion is reached. So we see convergence irregularity.

Matching is usually done pixel-by-pixel, and so is computationally intensive. When models describe 1-dimensional contours, dynamic programming techniques can be used to reduce the computation [Gei95]. Another technique for reducing complexity is a coarse-to-fine, multi-grid matching, where an initial rough match to subsampled input is iteratively refined [JZL96]. The techniques here tend to be regular.

The computational complexity of elastic matching may also be reduced by matching a model graph to features in the input data. Amit [AK96] defines features by local neighborhood operations on the input data, and then matches a model graph to them by a correspondence search. By requiring the model graph to be *decomposable* the search is reduced to dynamic programming, using the deformation energy as a cost function. Essentially, use of decomposable model graphs enforces an ordering on the matching of model components so that later matches have no constraining effect on previous ones[5]. In the completely non-rigid case, no prediction of feature location is possible, and each new feature is hypothesized, in turn, to be a member of all possible triples of model points, where the other two points have already been matched (the restriction to decomposable graphs makes this sufficient). The deformation energy for a particular triple is a simple function of the comparative lengths and angles of the associated triangle between the image and the model.

While the model is not active in the sense of directing search, the current partial match provides context for the matching of the next point via the deformation energy cost function. In general, we see convergence irregularity for these techniques, possibly with data structure irregularity from model traversal and feature lookup.

[5] Part of the constraining power of an arbitrary graph model can sometimes be recovered by simultaneously matching multiple decomposable subgraphs of it.

### Combining techniques

These techniques can, of course, be combined. Wheeler [WI95] uses relaxation (MRF) to "pre-prune" search, tree search to establish correspondences, robust LMS pose estimation, and 3-D template matching for verification of the back-projected model. Pre-pruning trees with relaxation reduces the number of hypotheses verified by a factor of four for realistic images.

### Reducing model matching complexity

A number of techniques have been developed to avoid having to match all models in the model base.

Basri [Bas93] gives a 2-stage approximate alignment scheme to avoid having to align all models. An image set is first (inexactly) aligned with each of a set of *prototypes* by minimization of a particular objective function. The set of possible prototypes is then reduced to a few best fitting ones, and for each of these, the previously constructed alignment is used to put into correspondence the image set and each model in the prototype's *class*. The class corresponding to a prototype is defined by clustering with respect to the objective function. Note that the prototypes may be matched in parallel.

Sengupta and Boyer [SB95] gives a hierarchical indexing scheme for libraries of graph models. Only the root of the index tree need be matched, this sets up the correspondence between scene and model features; the subsequent tree traversal requires only simple tests. The essential point is that the root model is now a somewhat reduced representation of the entire model base: it contains all primitives found in the model base, together with much weaker constraints, leading to less use of context in the search process. The match to the root is still exponential, but now on a much larger model than any found in the model base. Nonetheless, they report impressive speedups when models can be preclassified into types (for example, "chair") for which all models of this type are not too different. In this case, the root of the sub-modelbase searched does not differ too much from the individual models, so that little time is wasted matching the root model. Also, a set of subroots for the different types can be matched in parallel.

The primary search reduction technique for rigid models has, however, been the use of *indexing* techniques.

Geometric hashing [LW88] and related techniques [Rei93, Wei93, CM91], work by extracting a number of *model-based pose-invariant features* as pieces of evidence for the various models, and then choosing several best-supported models for further verification. Again, we see "imperfect" indexing, that can be followed by parallel model matching.

Invariants, $I(\vec{f})$, take a vector of features, $\vec{f}$ and produce a vector of some dimension as output. They are *invariant* to a class of transformations: the output vector does not change when the input vector is transformed by an element of the class; i.e., if $g$ is an element of the class, $I(\vec{f}) = I(g(\vec{f}))$. Suppose the invariant to be used requires $N$ features, and produces a vector of dimension $k$. For example, following Lamdan's original construction [LW88], given $N = 4$ points in $R^2$, three of them can be used to form a translation-independent coordinate system in which the $k = 2$ coordinates of the other point form an affine invariant, one that is unchanged by translations, rotations and scaling. For a $k$-dimensional invariant, a $k$-dimensional *hash table* is constructed like the GHT accumulator array by quantizing the $k$ dimensions and viewing each resulting cell in $R^k$ as a hash bin.

Suppose there are $M$ models, $M_i$, where each model consists of $m$ features, $M_i = \{f_{ij} | j = 1 \ldots n\}$. The set of models is used to create table entries $(\vec{x}, M_k, \vec{i})$, where $\vec{x} \in R^k$ is the vector of invariants computed on the (ordered) feature set $\{q_{ki} | i \in \vec{i}\}$, and $\vec{i} \in Z^{N-1}$ is a vector of integer indices. The table entry is stored in the cell in $R^k$ containing $\vec{x}$. The table is filled as follows:

1. for each model, $M_i$
2. for all combinations of $N - 1$ *basis features* chosen from the $m$ model features, $(f_{i,1}, \ldots f_{i,N-1})$
3. for all permutations of those basis features, $(f_{i,j_1}, \ldots f_{i,j_{N-1}})$

4. for all remaining $m - N + 1$ model features, $f_{i,0}$

5. enter $(I(f_{i,j_1}, ...f_{i,j_{N-1}}, f_{i,0}), M_i, (j_1, ...j_{N-1}))$ in the table.

The redundancy in the table is to allow matching when some features are occluded or incorrect due to noise. The table is constructed entirely off-line, so its computational complexity need not concern us; it contains $O(Mm^{N-1})$ entries. Each entry has 1 model index and (N-1) model-feature indices, totaling $\log M + (N-1)\log m$ bits. If weighted voting is used, entries also contain a vector of $k$ real numbers.

At runtime, the steps are:

1. given scene features $\{p_1, \ldots, p_s\}$, choose a basis set, $(p_{\mu_1}, \ldots p_{\mu_{N-1}})$ There are $O(s^{N-1})$ such bases.

2. for all the remaining scene points, $p_0$, compute the invariant $I_{\mu,0} = I(p_{\mu_1}, \ldots p_{\mu_{N-1}}, p_0)$ and record a vote for each pair $(M_k, \vec{\mu}_i)$ with an entry $(\cdot, M_k, \vec{\mu}_i)$ lying in the same bin as $I_{\mu,0}$.

3. if any pair $(M_k, \vec{\mu}_i)$ gets enough votes, verify the match

4. if the match fails, go to (1)

In the worst case, this requires $O(s^N)$ calculations of the invariant. Lamdan [LSW88] has shown that if there are a "reasonable" proportion of features in the image that belong to a single model, then the complexity is $O(s)$. Voting may consist of incrementing a counter, or may involve more complicated *weighted voting*, perhaps including interaction with nearby entries [RH93].

However, invariants are not a panacea to the problem of rigid object recognition: Grimson [Gri90] has shown geometric hashing, like GHT, tends to give many false positives in the presence of sensor noise. In the presence of sensor noise, speedup is limited, essentially to a constant (rather than being linear in the model base size, as it would be if hashing always found a single potential model) [CJ91, Gri90, Ols95]. Invariants for general 3-D models do not exist [CJ91], rather, instead of a model corresponding to a 0-dimensional manifold (point) in the "index space" generated by values of a presumed invariant, a model corresponds to a 2 or 3 dimensional manifold instead (orthographic projection giving rise to a 2-D manifold [CJ91] and perspective projection to a 3-D one [Jac96]). However, the "probabilistic peaking effect" [BA90] implies that the probability of observing points on these manifolds has a highly peaked distribution. Olson [Ols95, Ols93] shows how to use this peaking to find alignment matches with high probability of being correct, and reports speedups on real images of 20x when the model occurs in the image, and 100x when it does not (the common case, for large model bases). Olson also gives methods for eliminating from consideration model groups that are unlikely to give useful alignments; in this sense the model can then be said to direct search.

Thus, especially in the presence of sensor noise, indexing techniques are "imperfect", in the sense that they reduce the number of possible models matches that must be verified, but not to a singleton.

### Summary of matching techniques

Let us review the "stylized facts" from the beginning of the chapter.

**F1** Feature grouping (object hypothesis generation) involves construction and traversal of irregular data structures such as lists and trees.

This was discussed in section 3.2. Lists and trees serve either as representations of components (regions and boundaries) or as indexing structures for associative lookup. In the former case, traversal operations are used to compute features such as moments. The construction of the lists and trees may or may not be complex, but once constructed, the main algorithmic form is a simple data-dependent loop, with a small loop body of attribute comparisons and accumulations. A particular point is that the loop conditional, either checking for the end of the list, or for terminal tree nodes, forms a substantial part of the computation.

**F4** Model search and indexing techniques are "imperfect", in the sense that they generally restrict the number of models to be matched, but not to a single candidate.

This was discussed in the section of reducing match complexity for graph models. The point is that, as indexing is "imperfect", matching and verification of multiple candidate graph models in parallel is still of interest. For ordered models, the lexical tree provides an indexing method that seems unlikely to benefit from parallelism. For smaller vocabularies, where model-discriminant matching makes sense, indexing is not so useful, and parallel matching is again reasonable.

The sections on matching of graph models have tried to demonstrate

**F2** model matching algorithms exhibit irregular control flow mediated by the data and/or model, and

**F3** most model matching and feature grouping algorithms are simple, with computational complexity coming from applying a small code "kernel" many times

Descriptions of the algorithms have shown these trends for model-discriminant matching of HMMs; tree search, with its use of algorithms based on depth-first search; elastic matching of decomposable models; relaxation matching using either a dynamic programming type of algorithm or a priority queue based one; and model match verification, based on least squares pose refinement and location of closest points based on spatial data structures. In all these cases, we again have the phenomenon of simple kernels either iterated a data-dependent number of times, leading to convergence irregularity, or mapped over an irregular data structure, leading to data structure irregularity.

## 3.4 Parallelism and Communication in Matching

Consideration of parallelism brings up the issue of the distribution of input data, features, and models among the processors. Due to its small size, this is generally unproblematic for ordered input, text and speech, which may be broadcast and replicated at each processor. I will therefore speak in terms of visual data.

Matching a single model against a large distributed region of interest (ROI) leads to matching being handled globally. Parallelism is then used for associative lookup on the distributed data. The implication of Amdahl's law is then the need for a fast global processor. It is far from clear that significant parallelism can be achieved, compared to having the global processor also handle the associative lookup, but if, as is likely, the features have been computed in a distributed, bottom-up fashion, it may be preferable to leave them where they are, rather than moving them all to the memory of the global processor. Also, the use of an asynchronous communication network may allow the distributed features to be moved to the global processor as they are computed, in parallel with the computation of other features. So here, we see a role for parallelism in the form a special-purpose coprocessor for bottom-up processing.

In situations where the set of features in the ROI is not too large and where there are multiple models to match against, it makes sense to act as for speech and text and replicate the ROI on each processor [ND92], distributing the models among the processors. In this case, no interprocessor communication is needed during matching, except to construct the global interpretation; especially, to ensure that features are not interpreted as belonging to more than one object. However, if features have been computed in a distributed fashion, then those belonging to the ROI must be replicated on each processor. As for the previous case, an asynchronous communication network may allow this communication time to be mostly overlapped with other useful computation.

The remaining alternative is to distribute the ROI features among the processors, in "tiles", and then have each processor match one or more models on its tile in parallel. This is problematic in so far as the features belonging to a single object may be located on separate tiles. If the maximum diameter of an object is known, it may be possible to allow tiles to overlap, so that any object may be found entirely within a single tile. In the case of relaxation matching and geometric hashing where all scene features must be accessed at the start of processing, having an object lie in a single tile is the only option. If a scene object may not lie entirely in a single tile, interprocessor communication will be required during the match process. The communication pattern will be irregular, being determined by the particular models in question, the (scene-dependent) distribution of objects among tiles, and on how features are distributed and accessed. The latter two situations occur even if only a single model, possibly encoded in the algorithm, is being matched.

In the following, I will view the communication issues raised by this third alternative as outside the scope of the thesis. As a larger PE memory allows a larger replicated ROI, the communication difficulties of the third alternative are an argument for larger memories. The first alternative is unproblematic when parallelism on the coprocessor is bottom-up and SIMD, and subsumed into the second alternative when it is not. I will concentrate henceforth on the second alternative of a replicated ROI with parallel matching of multiple models.

## 3.5 Summary

In summary, examination of feature grouping and model matching algorithms leads to the conclusion that parallelism is unlikely to be valuable for feature grouping and model matching in applications with ordered input. Algorithmic characteristics for unordered input, particularly vision applications, suggest a hybrid architecture with a fast uniprocessor coupled to a parallel coprocessor. PEs of the coprocessor should have large memories and communicate via a asynchronous autonomous network. The PEs should support algorithms having small computational kernels that exhibit irregular, data-dependent control flow.

# Chapter 4

# Hardware

## 4.1 Introduction

To analyze the potential for parallelism in sensory processing we need to understand the technology available for implementation. I will therefore look at Very Large Scale Integrated (VLSI) semiconductor circuit technology, likely to be available in the near term, as it affects the performance of the algorithms used.

My philosophy of hardware analysis is that both detailed and simplified models of non-existent architectures are unconvincing with respect to ultimate performance, but that abstract, simplified models have more potential for giving insight into design and algorithm tradeoffs. I therefore do analysis at a high level – variables include: number of processors, fraction of chip area used by a given feature, frequency, power dissipation, off-chip bandwidth, granularity of computation (computation per memory access), size of model, and so on.

I am interested in relatively general-purpose, cost-effective systems for contextual sensory processing, and so will look at extending existing systems to cover a wider range of tasks, especially irregular tasks like model matching. I will focus as well on *delivery systems*, that is, systems designed for efficient delivery of specific functionality to end users, rather than, say, on general purpose workstation clusters, or research systems. I study such systems, involving small numbers of chips, both for economic reasons, and because many of the issues of large scalable systems are already well studied. A small systems focus leads us to assess the potential for on-chip parallelism, and multiple processors per chip.

Section 4.2 provides the qualitative background for the relevant architectural ideas. Section 4.3 describes current VLSI technology trends for the next 12 years. Using these VLSI trends, section 4.4 develops a quantitative model for the number of processors per chip. Section 4.5 then uses this model to examine the effect of limited off-chip bandwidth on the task of exhaustive model matching. Even for a highly parallelizable application, limited memory bandwidth and the tradeoff between the limited area for data-path and the limited area for on-chip memory constrains the useful number of PEs on a chip to be around 16-32, unless (essentially) all model parameters fit in on-chip memory. Finally, section 4.6 reviews current ideas about how VLSI trends will influence computer microarchitecture, and indicates how the SFMD architecture to be assessed in chapter 5 relates to these ideas.

## 4.2 Simple processors

Putting multiple processors on a chip requires that they use less area and hence be "simpler" than a single commodity RISC microprocessor on the same chip. Assuming the same VLSI process across comparisons, simple processors can save area by:

**limiting functionality:** One can restrict the operations done in hardware, possibly emulating them in software, or using slower, less area-intensive implementations. For "sensory" processing and context analysis, some reasonable possibilities are eliminating hardware floating point; limiting word size, perhaps to 16 bits; or eliminating integer division hardware. Later, I will show how projected VLSI process trends imply that these forms of simplification will be of lesser importance, except for the restriction on word size, which has implications for off-chip memory bandwidth.

**limiting instruction-level parallelism:** A simple processor may reduce the number of functional units (FUs) of a given type, and execute fewer instructions at a given time. Thus superscalar and speculative execution is reduced or eliminated. Even with only one FU of each type, it may still be possible to overlap loads, stores, and operations on the separate functional units. If this overlapping gives an average of 1.5 instructions per cycle (IPC), taking into account memory delays, then the degradation in performance compared to a quad-issue commodity microprocessor is at around two (figures 4.4 and 4.5). A commodity microprocessor implementing more than quad-issue will likely have small increased performance for the increased area, due to diminishing returns in trying to leverage concurrency within a single instruction stream. Even the most optimistic projections for billion transistor uniprocessors give a maximum IPC of about 12 [PPE+97]. Nonetheless, the projected VLSI trends indicate that after one or two more process generations, chip area constraints will not be a compelling reason for simplifying processors in this way. However, achieving greater ILP trades off against factors such as design complexity, manufacturing and test costs, power dissipation and chip size. A design must weigh these tradeoffs in terms of its intended application.

**sharing instruction processing hardware:** Here, shared hardware is used for producing an instruction stream used by all the simple processors. If the simple processors have no instruction memory, we have the familiar SIMD architecture. Later, I introduce the *Single Function Multiple Data (SFMD)* architecture, where each simple processor has a small instruction cache.

**sharing chip pins:** In this simplification, a given simple processor has only addressing into its local memory, with off-chip I/O and memory hardware shared among the simple processors. As for the previous methods, VLSI trends indicate that sharing resources that require only chip area will not be a significant factor unless the area required is quite large. For example, sharing large caches may be useful. However, pin (rather than area) limitations will force sharing off-chip I/O pins if there are more than about 32 processors per chip. Independent of the sharing issue, the growing disparity between computational performance and bandwidth indicates that maximizing data pin bandwidth will be important.

**sharing functional units** This is a recent idea due to Tullsen [TEE$^+$95] in which multiple threads share multiple functional units. Introduced as a way of increasing effective superscalarity, it can also be viewed as a way of sharing functional units among separate (virtual) "processors". This necessarily decreases maximum potential parallelism, but the effect on realized parallelism is unclear, due to improved resource utilization and latency hiding. The same trends indicate that the area saved by sharing functional units will not be significant in the long run. Also, the complexity of implementation, based on current superscalar design, grows quadratically with the number of processors, and may be prohibitive for larger numbers of processors (*e.g.* 16) due to design complexity and interconnect restrictions [SS95].

**reduced or shared memory:** Memory will constitute a dominant portion of chip area, and significant area may be saved by reducing the amount of cache or local memory needed by a processor. Sharing (L2) instruction cache is probably not difficult, but sharing data cache is more problematic due to coherence issues. On-chip shared or coherent cache systems may be a useful concept for multiplexing data pins among processors [NO94] but elaboration of this idea is outside the scope of the thesis.

## 4.3 VLSI measures

To construct a framework for discussing chip cost, I review some VLSI facts, and then look at projected trends over the next few years.

### 4.3.1 VLSI trends

The main driving force in the increased power of microprocessors has been, and will continue to be for some time, VLSI process improvements that allow smaller and smaller devices to be built on larger and larger chips, allowing increasingly complex designs to be fabricated cost effectively. While processes differ in many architecturally important respects [1], the basic scale measure, $\lambda$, defined as one half the gate length that can be built using the process, provides a useful way of comparing different processes[2]. A process having 1/2 the $\lambda$ of another process will, roughly, be able to put 4 times as many devices in the same area, and will also be able to be run at higher speeds due to the reduced distances and loads between points and reduced capacitances due to smaller *(charge* $\times$ *area)*.

An exception to scaling the number of devices, and increasing consequent chip "complexity", with $\lambda$ is in the I/O interface of the chip with the off-chip system. Both electrical and mechanical effects limit the degree to which the number of I/O pins scales with $\lambda$. Electrically, on-chip components forming the interface to a pin cannot shrink arbitrarily, but must be large enough, for example, to drive a signal across the off-chip wire. Mechanically, the need to bond chip pins to the package limits their density, and hence, for given chip dimensions, their number. Packaging can greatly impact the scaling of I/O pins: the more inexpensive packaging forms only allow chip pads at the periphery of the chip, forcing the number of pads to scale at most linearly with chip and feature size, while the number of chip features scales quadratically. I/O speeds are even more limited in their scaling with $\lambda$, as they are determined by the circuit board bus speeds, which typically do not exceed 100 MHz. The chip interface is thus a major barrier to the scaling of performance with process improvements.

Current trends in chip area, feature size, number and cost of pins, and general design cost, have been summarized by the Semiconductor Industry Association [Ass97]. These represent research goals and extrapolated trends, rather than forecasts, but probably represent the current best estimates of future performance. Tables 4.1 and 4.2 show some relevant trends for the next 7 generations, up to the year 2012. The table starts with the $0.25\mu$ process as the current generation, even though chips are now being manufactured

[1] For example, in the number of layers of interconnect.

[2] The use of $\lambda$ can be misleading as not all features scale at the same rate; in particular the pitch of metal layers tends to scale more slowly than lambda. None the less, it is probably the most agreed on and reasonable single number to use for comparison purposes. Whenever possible, we use actual projections rather than extrapolations based solely on scaling with $\lambda$.

using the $0.18\mu$ process, as it will be a few years before the $0.18\mu$ process is generally available.

I will refer to this table more later on. For now, the salient points are the enormous rate of increase in the functionality that may be integrated into a single chip, and the degree to which off-chip I/O bandwidth becomes increasingly limited over time, as shown by the number of pads per million transistors and by the I/O rate per million transistors. The response to this problem for general purpose machines has, and will probably continue to be, the movement of memory onto the chip [SPN96, Bur97]. The DEC 21164, with on-chip 96KB L2 cache to support its 625+ MHz clock rate, is a recent good example of this. There is thus increased pressure for "advanced" VLSI processes combining space-efficient RAM and logic on the same chip [Ass97]. Traditional processes optimized for DRAM use few metal layers and suffer a serious loss in efficiency in implementing logic, while processes optimized for logic are inefficient at implementing SRAM [Fos96]. The trend to more efficient combination of logic and memory on the same chip can be seen in table 4.5, and is discussed further in 4.7.

### 4.3.2 Chip Architecture

To understand the effects of tradeoffs at an architectural level, we need some idea of the silicon costs of the various functional components of a chip, such as registers, floating point units, and so on. Table 4.3 gives component sizes for three contemporary chips, as measured from chip micrographs [ERB+95, BBB+95, SDC94, LLNK96, GBKQ96]. These area numbers are rough, in that micrograph overlays are somewhat ambiguous as to the exact functionality contained in a given area. As indicated above, the current and future trend is toward moving more memory on-chip. Thus, memory device size is of particular importance. Table 4.4 shows sizes for a variety of current designs, from [Fos96]. Table 4.6 shows sizes for a variety of current and future "advanced" mixed ASIC-RAM designs.

For modeling, tables 4.3 and 4.5 can be distilled into table 4.6 of nominal component sizes. For now, I will look only at SRAM-based designs, and will discuss DRAM-based

Table 4.1: Description of fields for table 4.2. ‡"Mx" is a million transistors.

| *field* | *units* | *description* |
|---|---|---|
| year | - | Year of first DRAM introduction at given feature size. DRAM and logic in volume at that size available about 2 years later. |
| process size | $\mu$ | minimum feature size ($2\lambda$) |
| logic/SRAM density | Mx/cm$^2$ | packed transistors per unit area for logic and embedded SRAM process |
| chip size | mm$^2$ | total chip area |
| chip pads | - | number of chip pads; this exceeds number of package pins under the assumption that the package will be used to distribute power and ground to chip, or will contain multiple chips |
| package pins/balls (BGA) | - | number of package pins |
| I/O bus width | bits | bus width to system memory and peripherals (not cache) |
| chip speed | MHz | on-chip clock rate |
| off-chip speed | MHz | chip-to-board clock rate for peripheral busses |
| logic chip capacity | Mx/chip | packed logic or embedded SRAM transistors per chip |
| SRAM chip capacity | MB/chip | maximum embedded SRAM per chip (logic process) |
| I/O ratio | pins/Mx | possible off-chip signals per million transistors of logic |
| I/O bus bandwidth | GB/s | maximum bandwidth over system bus |
| I/O bandwidth ratio | (MB/s)/Mx | maximum bandwidth per million transistors |

Table 4.2: VLSI Technology Trends, from [Ass97]. Columns give process generations, with minimum feature size ($\lambda$) in row 2. Where applicable, values given are for the "cost/performance" market, targeting < \$3000 desktop machines and laptops. †: Goal is problematic, but solutions are currently being pursued. ‡: There is no way yet known to reach this goal.

| year | 1997 | 1999 | 2001 | 2003 | 2006 | 2009 | 2012 |
|---|---|---|---|---|---|---|---|
| process size ($2\lambda$) | $0.25\mu$ | $0.18\mu$ | $0.15\mu$ | $0.13\mu$ | $0.10\mu$ | $0.07\mu$ | $0.05\mu$ |
| logic/SRAM density ($Mx/cm^2$) | 3.7 | 6.2 | 10 | 18 | 39 | 84 | 180 |
| chip size ($mm^2$) | 300 | 340 | 385† | 430† | 520‡ | 620‡ | 750‡ |
| chip pads | 256-800 | 300-976 | 352-1193 | 413-1458† | 524-1968† | 666-2656† | 846-3587† |
| package pins/balls | 256-600 | 300-732 | 352-895 | 413-1093† | 524-1476† | 666-1992† | 846-2690† |
| I/O bus width (bits) | 64 | 64 | 128 | 128 | 128 | 256 | 256 |
| chip speed (MHz) | 350 | 526† | 727† | 928† | 1108† | 1468† | 1827† |
| off-chip speed (MHz) | 75/175 | 100/263 | 100/362 | 125/464 | 125/554 | 150/734 | 150/913 |
| package cost (cents/pin) | 1.4-2.8 | 1.25-2.5 | 1.15-2.30 | 1.05-2.05 | 0.90-1.75 | 0.75-1.50 | 0.65-1.30 |
| chip size ($10^9\lambda^2$) | 19 | 42 | 68 | 101 | 208 | 506 | 1200 |
| logic chip capacity (Mx/chip) | 11.1 | 21.1 | 38.5 | 77.4 | 202.8 | 520.8 | 1350 |
| SRAM chip capacity (Mb/chip) | 1.8 | 3.5 | 6.4 | 12.9 | 33.8 | 86.8 | 225 |
| I/O ratio (pins/Mx) | 23-54 | 14-35 | 9-23 | 5-14 | 2.6-7 | 1.3-4 | 0.6-2 |
| I/O bus bandwidth (GB/s) | 0.6/1.4 | 0.8/2.1 | 1.6/5.8 | 2.0/7.4 | 2.0/8.9 | 4.8/23.5 | 4.8/29.2 |
| I/O bandwidth ratio ((MB/s)/Mx) | 54/126 | 38/100 | 42/151 | 26/96 | 10/44 | 9/45 | 4/22 |

Table 4.3: Areas of various architectural components, in units of $10^6\lambda^2$,measured from micrographs with overlays. 'ALU' includes shifter and multiplier; the number in parentheses is the number of adders. The PA8000 has no on-chip caches, its load/store area includes 28 entry address reorder buffer and 2 address adders and it has 32 general purpose + 56 rename registers, each 13-ported (4 write + 9 read ports). The 21164 has a 96KB on-chip L2 cache, leading to the large number of transistors, the bus interface area includes L2 cache controller. The 21164 D cache is larger than its I-cache since it is dual read-ported. The 21164 GP registers are 6-ported (4 read, 2 write), its FP registers are 9-ported (5 read, 4 write), and each functional unit has 3 9-ported registers, 2 read + 1 write. The 21164 area also includes $52 \times 10^6\lambda^2$ for clock drivers. For the PPC604, registers have 8 read ports and there are 20 rename registers.

<table>
<tr><th></th><th>PA8000</th><th>PPC 604</th><th>21164</th><th>CNAPS 1064</th></tr>
<tr><td>I decode/fetch</td><td>66</td><td rowspan="2">46</td><td rowspan="3">131</td><td>-</td></tr>
<tr><td>I branch</td><td>77</td><td>-</td></tr>
<tr><td>I reorder</td><td>210</td><td>92</td><td></td></tr>
<tr><td>I cache</td><td>-</td><td>73 (16KB)</td><td>31 (8KB)</td><td>-</td></tr>
<tr><td>D cache</td><td>-</td><td>83 (16KB)</td><td>51 (8KB)</td><td>21 (4KB)</td></tr>
<tr><td>L2 cache</td><td>-</td><td>-</td><td>320 (96KB)</td><td>-</td></tr>
<tr><td>cache controllers</td><td>170</td><td>56</td><td rowspan="2">170</td><td>-</td></tr>
<tr><td>bus interface</td><td>134</td><td>46</td><td>-</td></tr>
<tr><td>load/store</td><td>90</td><td>53</td><td>52</td><td>-</td></tr>
<tr><td>GP registers</td><td>62</td><td>48</td><td rowspan="2">67</td><td rowspan="2">17 (16b)</td></tr>
<tr><td>ALU</td><td>77 (2)</td><td>82 (3)</td></tr>
<tr><td>FP registers</td><td>-</td><td>33</td><td rowspan="2">98</td><td>-</td></tr>
<tr><td>FP FU</td><td>225 (2)</td><td>83</td><td>-</td></tr>
<tr><td>pad area</td><td>230</td><td>120</td><td>220</td><td>-</td></tr>
<tr><td>die</td><td>1350</td><td>784</td><td>1200</td><td>4063</td></tr>
<tr><td>signals</td><td>701</td><td>171</td><td>?</td><td>75</td></tr>
<tr><td>transistors</td><td>3.8M</td><td>3.6M</td><td>9.3M</td><td>14M</td></tr>
</table>

Table 4.4: Memory device sizes for various designs, using current (1995) processes optimized for either SRAM or logic, but not both, from [Fos96]. The 'logic' process used is a 3-metal $0.5\mu$ ASIC process, the "standard" SRAM process is 2-metal $0.5\mu$, the "CNAPS" process is 2-metal, $0.4\mu$, and the "standard" DRAM process is 2-metal, stacked cell, $0.5\mu$. "Actual density" takes into account reduced space utilization due to number of metal layers and peripheral process rules. Area for sense amps and decoder logic is not included. "Logic scaling" gives the factor by which the area of logic devices increases for an SRAM or DRAM process, compared to the 3-metal ASIC process; this not known for the CNAPS process.

| type | process | device size $\lambda^2/bit$ | density (maximum) $KB/(10^6\lambda^2)$ | density (actual) $KB/(10^6\lambda^2)$ | logic scaling |
|---|---|---|---|---|---|
| embedded 6T SRAM | logic | 1072 | 0.12 | 0.08 | 1 |
| standard 6T SRAM | SRAM | 688 | 0.18 | 0.12 | 2.3 |
| embedded DRAM | logic | 368 | 0.34 | 0.20 | 1 |
| CNAPS 4T SRAM | SRAM | 294 | 0.43 | 0.35 | 1 |
| standard DRAM | DRAM | 51 | 2.45 | 1.22 | 3.9 |

Table 4.5: Various reported, measured and forecast SRAM densities, for $0.35 - 0.4\mu$ processes. The area reported includes ancillary circuitry such as decoders, cache tags and write buffers, except for values marked with †. Except for "SIA", processes are mixed SRAM and logic. "SIA" is from [Ass94]; "micrographs" from table 4.3; "NO" from [NO94]; and "KD" from [KD92]. For comparison, the CNAPS density (commercial 2-metal $0.4\mu$ SRAM process) is 0.19, showing the increasing density for "advanced" mixed logic and memory, small $\lambda$ processes.

| source | ports | density $KB/(10^6\lambda^2)$ |
|---|---|---|
| SIA | 1 | 0.31† |
| micrographs | 1 | 0.30†, 0.22, 0.26 |
| micrographs | 2 | 0.16, 0.19 |
| NO | 1 | 0.19 |
| NO | 3 | 0.08 |
| KD | 1 | 0.25 |

Table 4.6: Nominal component sizes, in units of $10^6\lambda^2$. The SRAM and DRAM numbers include ancillary circuitry such as sense amps and decoder logic. The smaller SRAM number is for single-ported memory, the larger is for triple-ported. The DRAM numbers are for single-ported memory. All sizes are for a logic process, except as noted.

| | |
|---|---|
| IU (32b) | 30 |
| FPU (32b) | 40 |
| SRAM (1 KB) | 5-10 |
| DRAM (1 KB) | 3 |
| DRAM (DRAM process) (1 KB) | 0.25 |

designs later. I include a nominal size for DRAM in a merged logic / DRAM process, as that giving a 20-fold decrease in density (including ancillary circuitry such as sense amps) over SRAM [PAC+97, FPC+97]. I also include a nominal size for DRAM embedded in a purely logic process, from table 4.5, assuming that for a single-ported DRAM, the sense amps and decoders take up 1/2 the area and are the same size as for SRAM, and that the cell size for embedded DRAM is 1/3 that of embedded SRAM, as given in the table.

## 4.4 Processors per chip

Based on the nominal sizes from table 4.6 and the data from table 4.2, one can make some statements about the number of processors it will be possible to put on a chip, assuming each processor has its own path to external memory.

First, consider pin limitations. Suppose each processor requires $d \in \{32, 64\}$ pins for data, $a \in \{0, 32\}$ pins for data addresses, $i \in \{0, 32\}$ pins for instructions, and $r \in \{0, 32\}$ pins for instruction addresses. Here $a = 0$ represents multiplexing the data pins with the address pins. Similarly, $r = 0$ represents multiplexing the instruction pins with the instruction address pins. Finally, $i = 0$ represents the situation where processors do not have their own instruction pins, as for an SIMD architecture where the pins are shared between the PEs. Pin counts are multiples of 32 as (i) this allows the use of Rambus technology [Cri97], which requires 31 pins and multiplexes address and data

Table 4.7: Pin limitations on number of processors per chip.

| $d$ | $a$ | $i$ | $r$ | $p$ | $pins$ |
|---|---|---|---|---|---|
| $pins \leq 600$ (1997) | | | | | |
| 32 | | | | 16 | 512 |
| $895 < pins \leq 1093$ (2003) | | | | | |
| 32 | | | | 32 | 1024 |
| 32 | | 32 | | 16 | 1024 |
| 32 | 32 | | | 16 | 1024 |
| 32 | 32 | | | 16 | 1024 |
| 64 | | | | 16 | 1024 |
| $1476 < pins \leq 1992$ (2009) | | | | | |
| 32 | 32 | 32 | | 16 | 1536 |
| 64 | | 32 | | 16 | 1536 |
| 64 | 32 | | | 16 | 1536 |
| $1992 < pins \leq 2690$ (2012) | | | | | |
| 32 | | | | 64 | 2048 |
| 32 | | 32 | | 32 | 2048 |
| 32 | 32 | | | 32 | 2048 |
| 64 | | | | 32 | 2048 |
| 64 | 32 | 32 | | 16 | 2048 |

signals, and (ii) when addresses are not multiplexed, either 24 or 16 pins are needed (for data and instructions, respectively) to provide sufficient address space, and more pins will be required for various control purposes.

Table 4.7 gives solutions for $(d, a, i, r, p)$ that satisfy the high-end pin count goals for the various process generations in table 4.2, assuming $p \in \{16, 32, 64\}$. The low-end pin count goals given allow no solutions for the $(d, a, i, r, p)$ within our limits, except for $(d, a, i, r, p) = (32, 0, 0, 0, 16)$ in 2006.

Based on the numbers from table 4.3, and on numbers from the current CNAPS, one can reasonably take the amount of chip area, both inter- and intra-processor, not including SRAM, IU, and FPU, to be about 20%. Together with table 4.6, this gives the number of processors that can be put on a chip as

$$processors = 0.8 \frac{size}{logic + cell \cdot 2^{memory}} \tag{4.1}$$

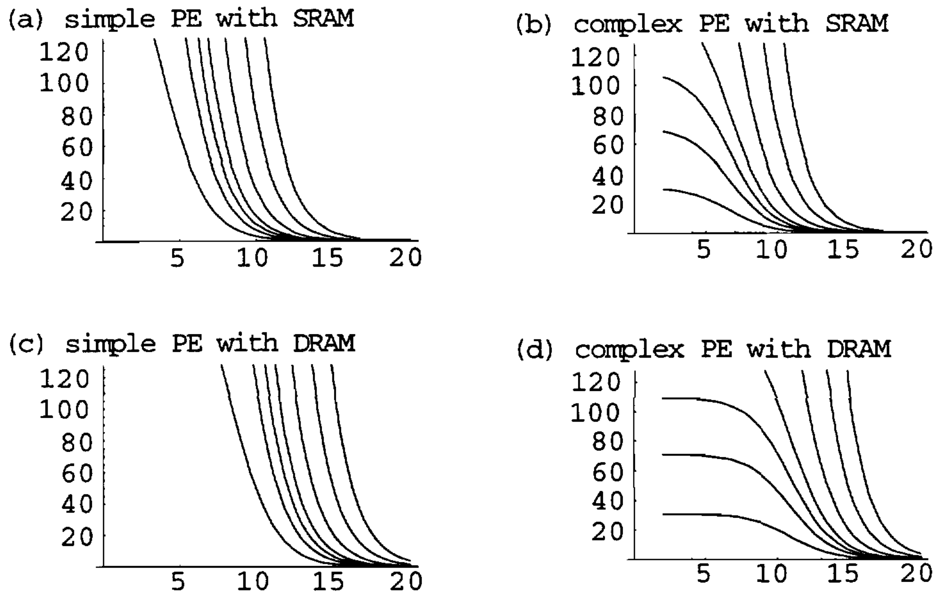


Figure 4.1: Plots for equation 4.1, the maximum number of processors per chip, as a function of the amount of memory per processor, for the next 7 process generations. The abscissa is the $\log_2$ of the amount of memory in KB. The ordinate is the maximum number of processors per chip, each having the given amount of memory. In the labels for the graphs, "simple PE" corresponds to *logic* = 70, "complex PE" corresponds to *logic* = 500, "SRAM" corresponds to *cell* = 5, and "DRAM" corresponds to *cell* = 0.25.

where *size* is the chip area in $10^6\lambda^2$, *memory* is the number of kilobytes of memory, *logic* is the area devoted to logic, and *cell* is the size of 1 KB of memory, from table 4.6[3] For a simple processor having only SRAM, a single IU and a single FPU, from table 4.6, *logic* = 70. For comparison, if we simplify processors only by sharing system bus interface and chip pads we can estimate from the numbers for the PPC604 that a reasonable "complex" processor size, not including on-chip caches, might be *logic* = 500. Figure 4.1 plots equation 4.1 for SRAM and DRAM processes and for simple and complex processors.

Table 4.7 shows that a 16 processor chip, with 32 I/O pins per processor, 32 pins (total) for instructions, and power and ground pins, is a viable option in the near term.

[3]This equation does not include a logic scaling factor, which will be 1 for the cases we consider. I assume that the logic scaling factor will be 1 for future merged logic-DRAM processes, tailored to implementing logic in a DRAM process.

Having 32 pins per processor would allow the use of Rambus memory technology, which can deliver 16 bits per 450MHz clock using 31 pins [Cri97]. Note that such a design uses much power: in general, to achieve a specified bandwidth, one trades off frequency against number of pins, and higher frequency translates into higher power.

An interesting option to take advantage of the increasing feature density is to look at the capabilities of the smallest chip in a given generation having this many pads, or the smallest package having enough pins. From table 49 of [Ass97] one can compute the minimum size flip chip having 700 pads, allowing 100 pads for power and ground in addition to the 600 for signals[4]. If $w$ is the chip width in $mm$ and $p$ is the pad pitch in $\mu m$, then $A = 1000(w/p)$ leads may be placed along a side of the chip. If there are $R$ rows of leads around the perimeter of the chip in which $A$ then the total number of leads is $4R(A - R) = 4000R(w/p - R)$. Solving $700 = 4R(1000w/p - R)$ for $w$ gives the values shown in table 4.8. The values for $R$ and $p$ are from table 49 of [Ass97]. The last column gives the number of transistors in such a chip of the given generation, using the logic/SRAM density values from table 4.2. The table clearly shows the trend of pin and pad restrictions limiting the possible on-chip parallelism, and hence leading to increased processor complexity ($Mx/PE$).

Table 4.7 and figure 4.1 show that pin limitations, rather than area limitations, will define the possible solutions for situations with 16 or more processors per chip, assuming data pins are not shared. In table 4.7, for the first two generations, only instruction pin sharing allows non-shared data pins at all, and for the next two generations, it allows a doubling of the maximum possible number of processors per chip. In the final two generations, the possibilities are more complex, but the general point holds that freeing pins by instruction pin sharing may be very valuable. Both table 4.8 and 4.1 coupled to the limits of about 32 processors (with individual paths to off-chip memory) per chip given by table 4.7, show that simplifying processors by reducing the complexity or number of

[4]The assumption that 100 extra leads for power and ground is consonant with the numbers given in [Ass97], which, for instance, gives a 1997 "cost-performance" target of 600 signal (non-power / ground) pins together with a target of 704 total leads.

Table 4.8: Minimum chip size for 16 processors with individual I/O

| year | generation ($2\lambda$) | pitch $p$ ($\mu m$) | chip width $w$ ($mm$) | | chip size ($10^9\lambda^2$) | | transistors ($Mx$) | |
|---|---|---|---|---|---|---|---|---|
| | | | ($R=3$) | ($R=4$) | ($R=3$) | ($R=4$) | ($R=3$) | ($R=4$) |
| 1997 | $0.25\mu$ | 250 | 15.3 | 11.9 | 14.8 | 9.0 | 8.7 | 5.3 |
| 1999 | $0.18\mu$ | 180 | 11.0 | 8.6 | 14.9 | 9.1 | 7.6 | 4.6 |
| 2001 | $0.15\mu$ | 150 | 9.2 | 7.2 | 14.9 | 9.2 | 8.5 | 5.1 |
| 2003 | $0.13\mu$ | 130 | 8.0 | 6.2 | 15.0 | 9.0 | 11.4 | 6.9 |
| 2006 | $0.10\mu$ | 100 | 6.2 | 4.8 | 15.4 | 9.2 | 14.7 | 8.9 |
| 2009 | $0.07\mu$ | 70 | 4.3 | 3.3 | 15.1 | 8.9 | 15.5 | 9.4 |
| 2012 | $0.05\mu$ | 50 | 3.1 | 2.4 | 15.4 | 9.2 | 16.9 | 10.3 |

Table 4.9: Minimum chip size for 32 processors with individual I/O. Computed as for 16 processors (table 4.8), but assuming 1200 pins needed rather than 700.

| year | generation ($2\lambda$) | pitch $p$ ($\mu m$) | chip width $w$ ($mm$) | | chip size ($10^9\lambda^2$) | | transistors ($Mx$) | |
|---|---|---|---|---|---|---|---|---|
| | | | ($R=3$) | ($R=4$) | ($R=3$) | ($R=4$) | ($R=3$) | ($R=4$) |
| 1997 | $0.25\mu$ | 250 | 25.8 | 19.8 | 42.2 | 24.8 | 24.6 | 14.2 |
| 1999 | $0.18\mu$ | 180 | 18.5 | 14.2 | 42.3 | 24.9 | 21.2 | 12.5 |
| 2001 | $0.15\mu$ | 150 | 15.5 | 11.9 | 42.4 | 25.0 | 24.0 | 14.2 |
| 2003 | $0.13\mu$ | 130 | 13.4 | 10.3 | 42.2 | 24.8 | 32.3 | 19.1 |
| 2006 | $0.10\mu$ | 100 | 10.3 | 7.9 | 42.4 | 25.0 | 41.4 | 24.3 |
| 2009 | $0.07\mu$ | 70 | 7.2 | 5.5 | 42.3 | 24.8 | 43.5 | 25.4 |
| 2012 | $0.05\mu$ | 50 | 5.2 | 4.0 | 43.3 | 25.6 | 48.7 | 28.8 |

their functional units (other than cache), will diminish in importance.

The case examined here is when data pins are not shared (each PE has its own path to external memory), intimating some form of distributed or NUMA memory model. The next section addresses architectures that share data pins.

## 4.5 Processor-memory tradeoffs: exhaustive model matching

Appendix B examines the effect of off-chip data bandwidth limitations on achievable parallelism. For the task of exhaustive matching of a set of models, it looks at the effects on potential parallel speedup of factors such as data-path / chip-memory area tradeoffs, model base size, amount of computation per model, comparative speed of simple PEs versus complex microprocessors, and preloading or caching of models most likely to match. These factors are studied under the assumption that off-chip data bandwidth is the same for both the sequential and the parallel architectures, using the technology parameter values from this chapter, and under assumptions favoring on-chip parallelism.

The dominant condition turns out to be whether all models can be stored on-chip; more precisely, whether essentially all the algorithm's working set of models fits on-chip. If so, linear speedup is possible, if not, then parallel speedup is limited to $k$, the ratio of the computation time spent matching a model to the time taken to load the model. Figure 4.2 plots, for various chip size and processor complexity assumptions, the largest number of PEs for which the entire model-base fits on-chip, as a function of the model-base size. It shows, for 16 or 32 simple PEs per chip, assuming an SRAM size of $5 \times 10^6\lambda^2$ per KB, that model-bases in the 2-3 megabyte range will fit on-chip in the current generation, and model-bases of size 5-7 MB in the next generation.

In comparison, consider a merged logic-DRAM process with DRAM memory of size $0.25 \times 10^6\lambda^2$ per KB, and allowing for the same amounts of SRAM (0, 2, 4, 8, or 16 KB at $5 \times 10^6\lambda^2$ per KB). The SRAM may be used for instruction memory, cache, register file or a second "fast random access" memory. Here, the 20-fold increase in density translates directly into a 20-fold increase in allowable model-base size. The increased density of DRAM has shifted the balance, so that area for data-path and SRAM is now probably more limiting for most designs than area for the model-base.

As the number of processors increases, memory available for storing the models decreases as silicon area previously devoted to memory is used for data-path. Figure 4.3 shows how, in a typical case, as the number of processors increases past the point where all models fit on-chip, the speedup of the parallel implementation over the sequential on rapidly decays to near $k$, the ratio of compute time per model to the time to load a model from off-chip.

Thus there are basically two regimes, one where the working set of models fits on-chip, and one where parallel speedup is limited to $k$ independently of the number of processors, $P > k$. The important conclusion is that, for the applications I consider, *there are only two viable architectural alternatives*: if the range of target applications allows each model set to fit entirely on-chip, then an architecture of many small processors may be preferred, leading to a vector, SIMD or SFMD architecture. In all other cases, a few (slightly more than $k$, depending on distributional assumptions) complex PEs will be preferred, leading to an MIMD architecture.

## 4.6 Microarchitectural trends

Another way to see the impact of current VLSI trends on microarchitecture is to look at current ideas about what architecture(s) will be appropriate for later process generations. After reviewing how a current state of the art superscalar uniprocessor works, and briefly discussing *Very Long Instruction Word (VLIW)* and vector processing architectures, I will look at a number of extrapolations on how architectures could make use of the huge number of transistors available in later generations.

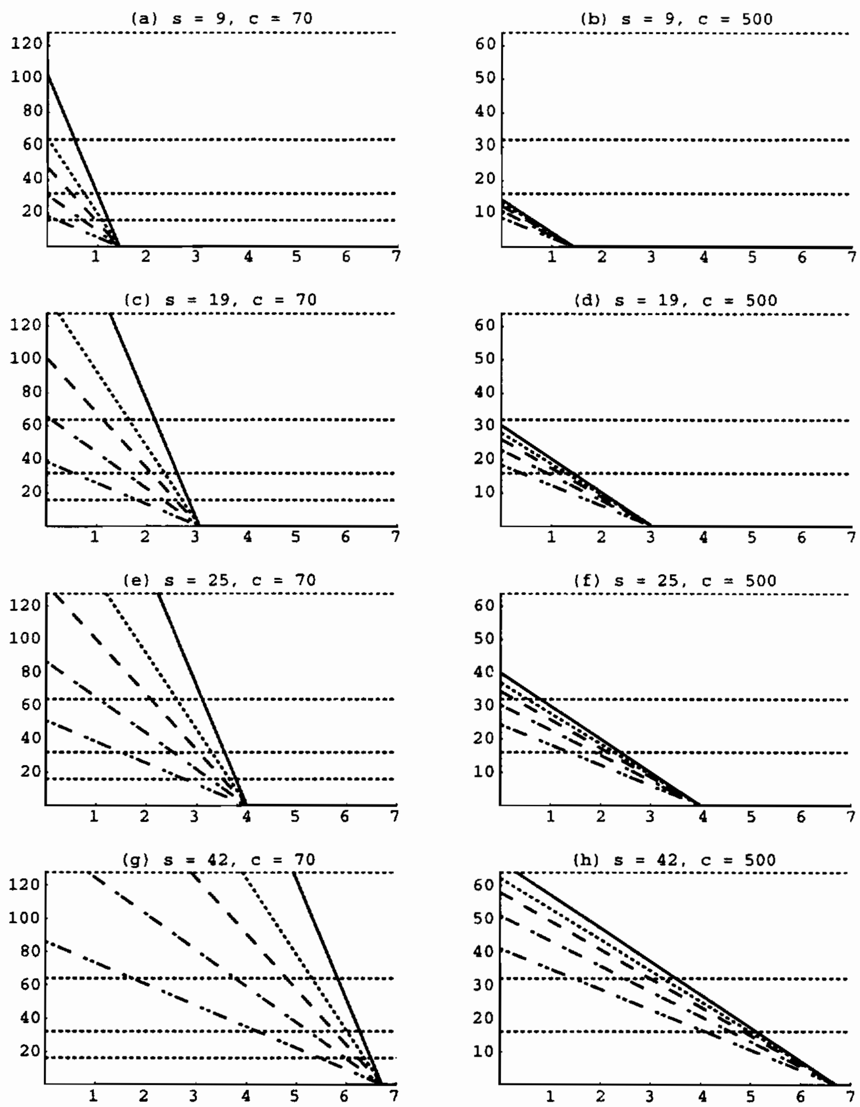


Figure 4.2: The maximum number of processors for which all models fit on-chip, as a function of the total model-base size in megabytes, assuming an SRAM size of $5 \times 10^6\lambda^2$ per KB. The different lines on each set of axes correspond to different amounts of per-PE instruction memory; 0, 4, 8, 16 and 32 kB. Horizontal lines are drawn at 16, 32, 64 and 128 processors, for comparison. The different axes show the graphs for differing chip size and processor complexity assumptions. The parameter $s$ (see table 4.1 and 4.2) gives chip size in units of $10^9\lambda^2$ and to a certain extent indicates the process generation: $s = 19$ corresponds to a "commodity" chip of the $0.25\mu$ generation, $s = 42$ to a commodity chip of the $0.18\mu$ generation, and $s = 9$ and $s = 25$ to minimal chips of the same two generations having sufficient pins to allow per-processor external memory for 16 and 32, respectively. The parameter $c$ gives a measure of processor complexity and area requirements, $c = 70$ corresponds to a simple floating point processor, while $c = 500$ corresponds to a complex superscalar processor.

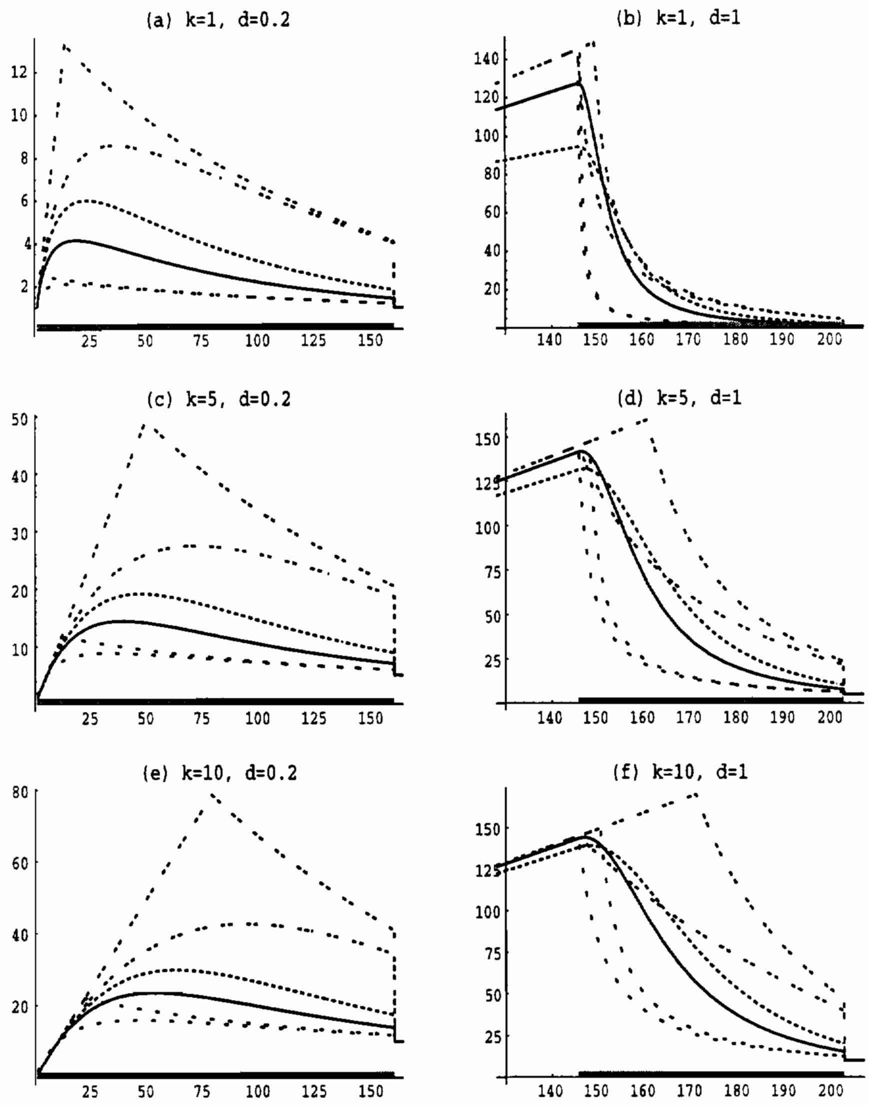


Figure 4.3: Speedup as a function of the number of processors, for varying $(k, d)$, where $k$ is the ratio of computation time per model to the time to load a model from off-chip memory, and $d = 1$ corresponds to a model base size of 1 MB, $d = 0.2$ to a size of 5 MB (both sizes of model bases are assumed to have 1000 models). Different curves on the same graph give speedups for different task distribution and modeling assumptions, as described in the appendix. The gray line on the x-axis marks the interval for which some, but not all, models fit in on-chip memory. For numbers of processors to the left of the interval, all models fit on-chip. For numbers to the right of the interval, the per-PE memory is too small to hold a single model.

### 4.6.1 Superscalar Architectures

Superscalar architectures attempt to extract as much parallelism as possible from a single instruction stream. The goal is higher performance while maintaining binary compatibility with previous software. The 1998 state of the art is a maximum issue rate of five instructions per cycle. There seems to be no generally acknowledged limit to what is possible, with estimates from an issue width of eight [TEE+95] to sixteen [LS97] and even thirty two [PPE+97].

Superscalar execution has the following features [SS95]. First, a set of instructions is fetched. With branches, there is uncertainty about what instructions will be needed; because of pipelining, it substantially degrades performance to not fetch instructions before it is known which branch will be taken. *Branch prediction* guesses the most likely branch, so that its instructions can be fetched while the branch condition is still uncertain. Based on static or dynamically acquired information, branch prediction can be quite accurate, but mispredictions still occur. To provide a larger window of instructions in which to discover ILP, branch prediction may be elaborated into *speculative execution*, where instructions in the predicted branch are not only fetched, but may be executed before the branch direction is finally known. Misprediction may then require undoing changes to registers and other state. Speculative computation is wasted when the branch is mispredicted, which is a particular problem when branching is data-dependent, as it is for most of the algorithms of chapter 3. A secondary point is that branch prediction based on runtime information requires a substantial amount of area to store statistics.

The ability to undo changes to state is provided by having instructions affect temporary *rename registers*, which can be copied to real "logical" registers once the branch direction is known to be correct. In addition to the area for the extra registers, this also requires hardware and state for mapping rename registers to logical ones. Besides speculative execution, a superscalar architecture usually supports *out of order execution*, in which parallelism is increased by allowing instructions to be executed out of sequential order, typically as soon as the necessary data and functional units are available. In order to preserve sequential semantics, this generally means that instructions are *retired* in sequential order, at which time their associated rename registers are written to the corresponding logical ones.

With out-of-order execution, the task of determining which instructions may execute when is complicated, requiring that an instruction's data dependencies be determined, and that the needed data be found. If, as is typically the case, the needed data is the result of a recent computation, it may be in a rename register rather than a logical one. The data may or may not be available yet, depending on the execution status of the instruction producing it. The determination and tracking of data dependencies and availability is usually done by means of a *reorder buffer* which maintains a window of instructions, removing them as they are retired. A reorder buffer consumes a lot of area, both for the buffer itself and for interconnect between it and the functional units and registers (cf. table 4.3).

Up to this point, I have ignored the issue of accessing memory. This is state that may have to be restored after a failed speculation, and something like rename registers are used, however the resolution of data dependencies requires that address comparisons be used to determine the mapping to memory locations. For speed, an associative lookup table of active memory locations is needed. From the point of view of implementing the reorder buffer, accessing memory is more complicated than other functional units due to the unpredictable latencies involved. Executing multiple instructions per cycle means that state, both registers and memory, must be multi-ported to allow multiple accesses per cycle, with concomitant expense – for $n$-fold multi-porting, the area increase is approximately $n$-fold, and the access speed is reduced by a factor of $\log(n)$ [ZA92].

With this background, we can see why superscalar architectures can provide only a small amount of parallelism cost-effectively, especially for the applications I am interested in. First, superscalar execution is expensive in terms of area, both for multi-porting and for control. In fact, the area required for monitoring dependencies and resource availability is quadratic in the issue width [SS95]. Superscalar designs are also extremely complex,

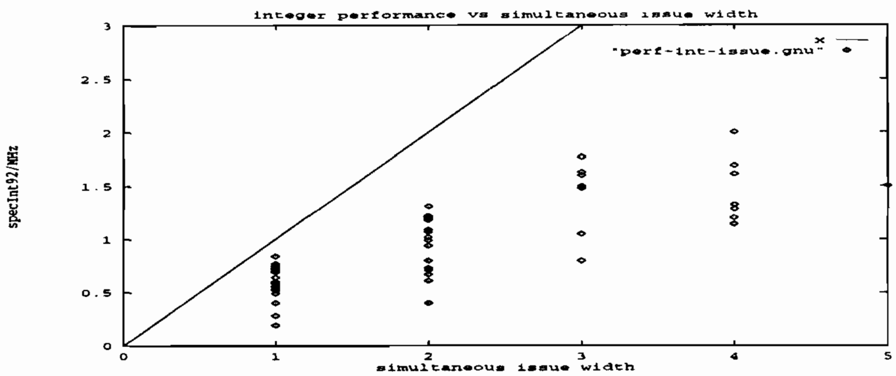


Figure 4.4: Scaling of integer performance with simultaneous instruction issue width.

and have a high design and test cost. Second, a large set of studies essentially conclude, for common codes such as the Spec benchmarks, that imperfect branch prediction and resultant wasted speculation severely limit the achievable parallelism to small factors (2-4) [SJH89, JW89, BYP+91, LW92, TGH92, Wal91, ME93, EG95]. These estimates are borne out by the performance of contemporary superscalar processors, as shown in figures 4.4 and 4.5, where performance normalized by processor speed is seen to be sublinear in the issue width. One estimate for maximal achievable parallelism for billion transistor designs issuing 16 or 32 instructions per cycle puts the achievable instructions per cycle at about 12 in both cases [PPE+97]. Another estimate for a 16-wide issue design gives an IPC of about 9 [LS97].

### 4.6.2 VLIW and Vector Processing architectures

*Very Long Instruction Word (VLIW)* architectures attempt to exploit instruction level parallelism without all of the hardware complexity of a superscalar design. Rather than using hardware to find ILP, in a VLIW design the compiler determines parallelism and issues a long instruction word that directly controls multiple functional units on the chip. This avoids much of the design complexity and various hardware difficulties of a superscalar design, for example, highly multiported registers, bypass interconnect area cost and speed

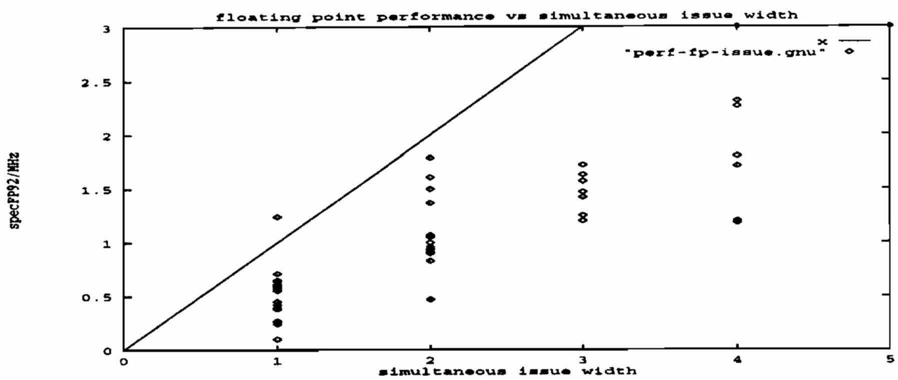


Figure 4.5: Scaling of floating point performance with simultaneous instruction issue width.

limitations, and design complexity. We will see that superscalar designs have problems with data-dependent control flow, due to increased branch misprediction, that limit the amount of parallelism they can exploit. While a VLIW design has the advantage that the compiler can search for ILP over a larger window of instructions, that should help little when control flow is data-dependent. So a VLIW design should have little advantage over a superscalar one for the problems we consider, and VLIW architectures will not be discussed further.

As mentioned earlier, vector (micro)processors (VP) embed parallelism at the assembly language level as operations on vectors of elements. Such a design has a number of good characteristics (see [Asa98]). Some of the advantages are: different implementations of the same design can provide different numbers of PEs, hidden from the programmer by the vector abstraction; computation is naturally partitioned in a way allowing a fast scalar processor to work together with the array of PEs; compilation for such designs is well understood; and the regularity of the parallel computation allows for a variety of hardware optimizations. The major disadvantage is a lack of flexibility, stemming from the use of the single vector datatype for all parallel constructions.

However, from the point of view of this thesis, vector processing is a type of SIMD processing. The memory bandwidth argument of 4.5 is the same as for SIMD designs, with the same implications: either few complex PEs or many simple ones with onchip memory. Also, the performance comparisons of SIMD and SFMD in chapter 5 apply to vector processing as well. The point is that, in this thesis the analysis is at broad enough level that VP and SIMD are hard to distinguish. However, it is *not* clear how to extend a VP design to include SFMD functionality, whereas extending a more general SIMD design like CNAPS to SFMD is simple.

### 4.6.3 Implications of trends

Implications of the VLSI trends I have outlined were elaborated by a number of architecture teams in a discussion of design options for a chip containing a billion transistors, as will probably be buildable[5] within the next 10 years [BG97]. This discussion brought out a number of common themes and issues (with differing emphases by different authors): generality of use, design complexity and design cost; cost and complexity of instruction delivery; interconnect and timing effects; hiding memory and functional unit latency; and memory bandwidth issues.

For complicated designs, generality of use is critical to offset design time and cost. The alternative to a complicated design is to reduce design time and cost by reusing a single (simpler) design, by replicating a part many times.

As we have seen in the description of superscalar architectures, much if not most design complication comes from issues of instruction delivery: handling branching and discovering instruction level parallelism (ILP). Designs focus on discovering ILP for reasons of both compatibility with existing code, and because of the importance of speeding up (apparently) sequential code, as expressed by Amdahl's law. Discovering ILP is the motivation for superscalar designs, and we have seen above that the cost, in terms of reorder buffers and the like, is high, while relative increases in performance are slowing down, leading to a point of diminishing returns. The major difficulty in finding additional ILP is due to the unpredictability of branches and hence of what code will actually be

[5]However, the question of whether it will be cost-effective to *design* such a chip is still open.

executed. Without speculative execution, the latter limits the region of code within which non-dependent instructions must be found, and hence the available ILP. With speculative execution, short cycle times lead to deep pipelines, implying large penalty for mispredicted branches when the pipeline must be flushed. There are two complementary solutions to the branch prediction problem: One is speculative execution of the most probable branch, or even redundant computation of both branches, the former of which occasionally wastes computation, and the latter of which always does. The second solution is better branch prediction, which requires substantial area for tables to accumulate branch prediction statistics. Also, as ILP is found, multiple instructions must be issued in a single cycle, which becomes expensive for wide (4-8) issue designs, due to the need for a multi-ported instruction cache and complicated instruction merge/align logic for simultaneous issue of non-contiguous instructions.

A number of unobvious issues about interconnect and timing arise in later VLSI generations. Smaller feature sizes lead to higher clock frequencies; in later generations, with larger chips as well, an across-chip signal may take up to 20 clock cycles [Mat97]. Even without the effect of larger chips, wire delays scale linearly, rather than quadratically. For small $\lambda$ wire delays soon dominate gate delays [SV97]. One analysis of superscalar designs [PJS97] shows that delays in the bypass wires that move results from one execution unit to another become a major limiting factor for wide-issue superscalar designs. One implication of these effects is that chip architectures will be partitioned into modules with most communication local to a module. A second implication is that very wide-issue superscalar may be difficult to achieve; this is also implied by the quadratic growth with issue width of the area used for instruction delivery [SS95], and the possible unavailability in any case of sufficient usable ILP.

Memory accesses (and some operations such as division) require multiple cycles to execute, introducing a latency problem. As the disparity between memory and CPU speeds grows, it will become increasingly difficult to keep the processor busy while a memory request is serviced. Latency is *tolerated* by executing other instructions while the request is serviced, requiring out-of-order execution and substantial hardware support, as described above for superscalar architectures. Tolerating the larger delays to be expected with upcoming generations will require the discovery of even more ILP, making further hardware demands. For tasks with much data-dependent execution, it is unclear whether sufficient ILP can be discovered. Rather than tolerating it, latency can be *reduced* by various techniques such as prefetching, use of larger cache blocks, and other forms of speculative loads; by multi-threading; and by intelligent cache management of lockup-free caches. Unfortunately, these all tend to increase traffic to memory [BGK96], exacerbating memory bandwidth requirements, and all substantially increase design complexity.

Perhaps the largest problem in future generations will be in providing adequate bandwidth to off-chip memory. Current microprocessors already spend a large fraction of time idle, waiting for memory. This could be due either to memory latency or to raw memory bandwidth limitations due to pin count. Simulations of aggressive latency-tolerating designs for a number of common benchmarks show the processor idle 10 to 30% of the time purely because of memory bandwidth constraints [BGK96]. The key point is that any improvements in finding ILP make this worse, since such improvements imply more instructions, and hence more data, processed each cycle. The same is true for other forms of on-chip multiprocessing: more instructions processed means more data is needed. One current estimate is that a wide superscalar architecture issuing 16 instructions per cycle will require a bandwidth of 8 loads or stores per cycle [PPE+97]. Coupling this with the fact that off-chip accesses will be serviced using a system clock that is at least 4 times slower than the on-chip clock, we see the reason for the current trend of moving as much memory on-chip as possible, as large on-chip level one and level two caches. For modularity reasons, this trend is likely to lead to multiple multi-ported on-chip caches. Another possibility is the use of on-chip DRAM for greater density (see the discussion of IRAM technology below, see also [SPN96]). The problem is exacerbated in chips designed for a low price-point, due to the high cost for packages with large numbers of pins. I examine the memory bandwidth problems specifically for model-matching type tasks in Appendix

B.

### 4.6.4 Competing design plans

The teams in [BG97], mentioned at the beginning of the previous section, presented six basic architectures, dealing with the implications of these issues in various ways. I will omit one of the plans [WTS+97] as tangential to this discussion, and add the simultaneous multi-threading design, which was omitted from the issue due to lack of space and availability elsewhere. The first three designs presented were for uniprocessors, emphasizing that general utility and code compatibility are paramount for recovering the design, test, and fabrication costs of complicated designs and large chips.

The first plan [PPE+97] is, essentially, for more of the same: a wide-issue (e.g. 16) superscalar design with out-of-order execution to tolerate latency, out-of-order fetch for both instructions and data, speculative loads, a large on-chip cache to provide adequate memory bandwidth, and much area devoted to better branch prediction. Simulations of this design on contemporary (SPEC 95) benchmarks show an execution rate of over 12 instructions per cycle (IPC). The designers point out the following advantages: this design uses only known techniques; simultaneous multi-threading (SMT) (see below) and multiple processors on a chip (CMP) designs are too memory bandwidth bound to achieve better performance even if they theoretically have a better IPC rate – the designers believing that, for parallelism, it is better to use multiple chips and chip hardware to tolerate latency. The other teams point out the corresponding disadvantages: the design does not deal with interconnect issues, and modularity of the architecture is unclear; the design is extremely complex; it has lower computational density than an SMT or CMP design; and data-dependent branching will be a problem.

The second uniprocessor plan [SV97] emphasizes the need for modularity and signal locality. A *trace processor* design consists of multiple superscalar modules together with higher-level control and instruction dispatch. The individual modules are complete, but simple, superscalar CPUs except that instruction delivery is handled globally. A *trace* is a sequence of (8-32) instructions giving the code to be executed assuming certain (zero or more) branch conditions hold. Global instruction dispatch predicts traces to be executed (rather than single branches) and sends them to the CPU modules for speculative execution. The design target is an IPC rate of 16. The advantages of a trace processor are modularity of design, locality of (most) signaling, and sharing of all the complexity of instruction handling, in a uniprocessor framework, with its advantage of being general purpose. The design again has problems with data-dependent branching, and wasted speculation, also the interface between the several CPU modules and memory requires a complex distributed system of multi-ported caches with a coherence protocol: different modules are executing code from the same thread, so there is no notion of distributed memory with some memory "belonging" to a given processor.

The third uniprocessor plan [LS97] emphasizes the problem of data-dependent branching, arguing that current techniques are approaching a limit, and that further progress along this line requires speculation of data *values*. Simulations show that speculating data values is possible for current benchmark codes. The advantages of this design are in improving branch prediction and IPC in a general purpose uniprocessor design. The problem is that it depends on value predictability; other work by the same authors [LWS96] shows that sensory processing tasks, for example, MPEG coding, may have little such predictability.

A *simultaneous multi-threading (SMT)* design is somewhere in between a uniprocessor and a chip multiprocessor. The goal of the design is for better resource usage with only small changes to conventional superscalar design. SMT uses multiple program counters to execute instructions from multiple threads of control on a single chip, using the same mechanisms a superscalar architecture does to support out-of-order execution and ILP. Although the goal is better performance on multi-threaded code, SMT performs well on more standard parallel codes well, too [LEE$^+$97]. The advantages of the design are better use of hardware resources because of the ability to utilize both instruction-level and thread-level parallelism, and the use of known technology. Its disadvantages are the same as that of a wide superscalar design: lack of modularity, interconnect issues, especially for bypass circuitry, complexity of the instruction logic, difficulties with data-dependent execution, and smaller computational density than CMP due to substantial instruction handling logic. In addition, the unified data cache may be problematic for multiple threads [HNO97, LEE+97].

The final design considered is a *Chip Multiprocessor (CMP)*, which puts a small number (8) of shared-memory MIMD processors on a single chip [HNO97]. The processors are relatively simple: dual-issue superscalar with in-order execution, each having its own L1 instruction and data caches. In simulations that compared such a CMP with a 12-wide issue SMT design, performance was similar, with the SMT design somewhat more flexible. An important exception was in the simulation of codes with high data bandwidth requirements, where the increased data cache bandwidth of the CMP design gave substantially better performance. It was also argued that the SMT design will not scale with decreasing process feature size due to interconnect effects and the lack of locality in wide-issue superscalar designs. The advantages of CMP are thus argued to be its design modularity and locality of interconnection, the computational density of multiple processors, the design simplicity of simple processors with (mostly) local interconnections, and the data bandwidth of individual L1 caches.

Conversely, other design teams found several possible problems with a CMP design. In chapter 5 I propose the SFMD architecture, designed for model-matching tasks, which has multiple processors per chip; here I indicate my responses to the objections.

For use as a general-purpose chip, a CMP requires either that a single application be multi-threaded or that separate applications be run on the individual processors (multiprogramming). It was argued that coding multi-threaded applications is hard, and therefore a CMP will be used mainly for multiprogramming, the point then being that, as a general-purpose chip, the chip will have no "killer app" and no significant market, and so will not be built. In our domain, admittedly not general-purpose, the primary form of parallelism is data or knowledge parallelism. This suggests (and my architecture supports) an SIMD style of programming that is not difficult.

It was argued that, even for multi-threaded applications, limited inter-PE interconnect and synchronization overhead imply that partitioning a single application into multiple threads (1 per PE) with significant inter-thread communication will not scale [LS97]. My point of view: we have already seen that an autonomous asynchronous inter-PE network is needed. Synchronization in SFMD is global and fast. As it is global, it introduces delays as some PEs wait for others; I analyze this in the next chapter.

Amdahl's law [Amd67] implies that performance of single, multi-threaded applications will suffer disproportionately from the poorer sequential performance of the simpler PEs of a CMP. My point of view: processors should not be too simple. We have seen that in many cases, memory bandwidth limitations argue for fewer, more complex processors. Suppose we are comparing the execution of a code on two architectures, one parallel and one sequential, and that the individual PEs of the parallel design are slower than the sequential design, say by a factor of $q$, Amdahl's law notes that if a fraction, say $1/n$, of a code cannot be parallelized (is "sequential"), then the maximum speedup factor from parallelizing the code is at most $n$. If, when parallelized, the sequential section is executed on a single PE, then the maximum speedup is $n/q$. The implication is that $q$ should be made as small as possible for at least one PE of the parallel design. The typical solution is to add a fast sequential processor to the design, separate from the array of parallel PEs [WAK+96, KPP+97]. This processor can use an off-the-shelf design optimized for sequential processing, and execute in parallel with the PE array. Due to diminishing returns from increasing issue width in superscalar designs, a simpler, say dual-issue, superscalar processor can be used, which should not perform too much worse than wider-issue ones. While this is probably the best solution, since the processor is optimized for sequential tasks, note that some implementations of SFMD allow for one PE to be made more complex than the rest, and to be used for sequential parts of the code, but also execute as part of the PE array.

The use of simple processors implies that tolerating memory latency will be a problem.

My point of view: I have not analyzed the issue of tolerating latency, but as relative memory latency gets worse it becomes harder to hide with the available ILP, especially when data-dependent execution makes branch prediction difficult. Conventional designs also avoid memory latency by using on-chip caches. But caching depends on locality, and may not perform well with some kinds of data-dependent tasks, such as indexing using pointer-based structures such as trees. Some of these problems may be ameliorated by software techniques like compiler-directed prefetching, but these apply equally to simple and complex processors. So simple processors may not be at such a disadvantage compared to complex ones for data-dependent accesses.

The increased computational density of CMP exacerbates the off-chip memory bandwidth problem. My point of view: in fact, as we have seen, for exhaustive model-matching type tasks, off-chip memory bandwidth limits speedup to $k$, the ratio of the model computation time to model load time. With SFMD, instruction pins are shared, which may allow more pins to be devoted to data bandwidth. However, consider a CMP where the PEs have individual L1 I-caches and share an on-chip L2 I-cache. Running an SPMD program, such a system may have virtually all instruction requests satisfied on-chip. The aggregate off-chip instruction bandwidth may then be only slightly more than for a single processor design, and may be satisfied using the same number of instruction pins. Reducing the instruction bandwidth by use of SIMD or SFMD techniques then has little benefit.

For example, for 32 PEs each with 16KB L1 I-cache and shared L2 I-cache, the per-PE L1 miss rate is around 0.003% to 1.3% for a variety of benchmarks [HP96, FPC+97], with the benchmarks similar to sensory processing algorithms having smaller miss rates, 0.003% to 0.02%. With these numbers, the L2 cache is accessed only 0.1% to 0.6% of the time, for the tasks similar to sensory processing. Running an SPMD program, all processors access the same code set, so, many L2 accesses may be expected to be hits, finding code previously accessed by another processor (see [Lun87, MM93] for related work). Thus multiple on-chip PEs may not require a substantially higher aggregate off-chip instruction bandwidth than a single PE.

Note that use of SIMD/SFMD techniques may reduce the needed *size* of the L2 I-cache, as they guarantee that different PEs will be accessing (nearly) the same set of instructions.

## 4.7 IRAM and Embedded DRAM

Clearly, the "all models on chip" design benefits from as much on-chip memory as possible. The basic idea of an *Intelligent RAM (IRAM)* design [KPP+97] is to avoid memory bandwidth limitations by moving the system memory onto the chip, or, more precisely, by moving the data-path logic onto the memory chip. This is accomplished by using DRAM rather than SRAM to implement the memory, and implementing the chip in a DRAM process. Mostly for speed reasons, I have been assuming the use of SRAM in the designs, rather than DRAM. SRAM allows single cycle memory access, and is several times faster than embedded DRAM. Also, in a logic process, embedded DRAM is not much denser, perhaps a factor of 2-4, based on table 4.5. However, merged logic and DRAM processes are becoming available [TMH+98]. For future versions of these, the density of embedded DRAM may be around 16-32 times that of SRAM, taking into account interconnect and associated circuitry as well as just the memory cells themselves [FPC+97]. DRAM then becomes an attractive alternative or complement to SRAM for on-chip memory.

As on-chip memory, DRAM has significant latency, but very high bandwidth. When a row address is decoded, and the corresponding word line driven, the entire row of data is delivered to the sense amplifiers. Along the lines of fast page mode DRAM [Sha97], the data in the sense amplifiers may be moved in parallel to a cache, where individual data may be accessed in a single cycle. Thus in a few clock cycles, maybe 4, a large number of bits become available to the PE. The exact number of bits is a design decision; in a commercial DRAM chip, a row might contain 256 or 512 bytes. In a commercial design, where processing the data occurs on another chip, the data in the sense amplifier cache must be selected to be sent off-chip through an interface that is much smaller than 256 bytes wide. Thus most of the potential bandwidth is lost. However, if the processor is on the same chip, this need not occur, rather the processor can access the data in the sense amplifier cache directly, and the access latency may be amortized over all data accessed from a single row decode. Latency may also be hidden along the lines of an SDRAM design [Sha97]. In this case *two* sense amplifier caches are used: after delivery to the sense amplifiers, the data is moved *en masse* to one of the caches where it can be accessed while another row is decoded and delivered to the other cache, thus hiding the latency of accessing the second row. For data access with high sequential locality, such as vector processing, most or all of a row may be accessed before fetching the next row, amortizing the latency; further, if the next row may be predictably prefetched, the latency may be hidden entirely.

For non-sequentially accessed data, the latency of DRAM exacts a significant performance penalty. To alleviate this, a likely scenario is to use some of the allotted per-PE memory area for DRAM, and some for SRAM. The SRAM might be used as a separate memory, a large register file, or as a cache. In either case, if, say, half the area is used for DRAM, and half for SRAM, used of embedded DRAM still improves on-chip memory density over SRAM by an order of magnitude, depending on factors such as size of the arrays and the relative overhead of ancillary circuitry such as cache management hardware.

There are penalties for implementing logic in a current DRAM process, due to fewer metal layers and the need to tailor a DRAM process for density and minimal current leakage rather than for switching quickly [PAC+97]. However, recent opinion is that a DRAM process allowing fast logic is possible for a 20-30% increase in cost per wafer[FPC+97]. Another issue is the sensitivity of DRAM to heat: due to increased leakage, a small increase in operating temperature necessitates a greatly increased refresh rate [PAC+97]. Thus embedded DRAM chips may only be suitable for low power applications. With present trends, putting system memory on chip in DRAM limits the memory to around 128 MB in the next (gigabit DRAM) generation [KPP+97]. This number results when the entire chip consists of DRAM; since we want some logic, the actual limit is less. The Berkeley IRAM chip has about one fourth the chip area devoted to logic, giving 96 MB of DRAM. This may be limiting in image understanding where a great deal of derived information, such as extracted features, is kept [BDBH89] or where algorithms require a large amount of memory per PE [Web92]. However, this is a substantial amount of working space for, say, a 3 megabyte image (1K by 1K pixels in 24 bit color). As well as increasing the amount of on-chip memory, IRAM also increases memory bandwidth, both from having the PE(s) and memory communicate at the chip clock rate, and from having the PE data-paths access the entire sense amp array in parallel. A single CPU cannot utilize the large IRAM bandwidth [YHO97, BCK$^+$97], so the sense amp cache must be accessed in parallel by some number of processors. A vector processing model has been suggested [KPP$^+$97], which can utilize a relatively wide array per vector processor, say 64 bytes. However, as I have argued, contextual processing requires data-dependent execution, and, as I will show later, a vector or other SIMD model suffers a large performance loss on such tasks. Instead of a vector model, with a few wide per-PE memories, for contextual processing tasks it may be better to have more PEs, each with a narrower memory. Such a design may require more area for replicated decode circuitry, but can decrease power requirements because of its shorter word line lengths

Consider the following "back of the envelope" example. For a 64 PE system with 1 MB of DRAM per PE, a 16 byte-wide per-PE interface puts 4 32-bit words of data in the sense amp cache, amortizing latency by 4, assuming a 32-bit PE architecture accessing one 32-bit word at a time. The per-PE memory is then 128 bits wide by 512K bits deep. At 1 GHz, and assuming latency is entirely hidden by the various techniques I have mentioned, such a system delivers an aggregate bandwidth of $64 * 4 = 256$ GB/sec. For comparison, an off-chip interface to 16 Concurrent Rambus memories will require $16 * 31 = 496$ pins, perhaps two orders of magnitude more power, and deliver $16 * 1.6 = 25.6$ GB/sec. If we allow for improvement in Rambus technology by arbitrarily doubling the bandwidth, the aggregate bandwidth is still a factor of 10 less than the embedded DRAM design. As another comparison, if we imagine a future 10 GHz sequential processor with a 64-bit architecture, its bandwidth to on-chip L1 cache is $(64 bits \times 10 GHz) = 80 GB/s$ , a factor of 6 less. The last two designs will also use much more power.

Embedded DRAM is a promising approach for a low power "all models on chip" design, both for its density and bandwidth. Of course, much of this is speculation, and depends on the actual processes available in the future. In particular, real implementations are likely to use pre-designed DRAM "macro-cells" (e.g., [DBKK98]), and will be constrained by the characteristics of the designs available.

## 4.8 Power and size issues

I have not looked at power or die size issues in any depth, although they are clearly important in the economics of a chip targeted at embedded applications. Here are some observations and speculations.

The largest effect on power dissipation from an SIMD or SFMD design is likely to come from sharing of instruction processing hardware. For a desired hit ratio, the area requirements of a single shared global I-cache for an SIMD or SFMD design may be substantially less than the combined area of the per-PE caches in an MIMD CMP. On-chip cache uses a significant percentage of the total power used by a microprocessor [PAC+97]. The same advantage from sharing applies to other instruction fetch and dispatch hardware. Also, to the extent that sharing instruction pins or use of large on-chip memories decrease the number of pins needed, they allow a smaller die size, and reduce the power requirements for driving off-chip lines. Driving off-chip lines has a high power cost, two orders of magnitude greater than on-chip memory accesses to either DRAM or SRAM [FPC+97].

[Asa98] discusses other ways a vector processing or SIMD design may have an advantage in power dissipation, compared to a non-vector microprocessor. These include less expensive register files, reduced datapath switching energy, less switching of datapath control lines, and more regular access to the memory system. Possible disadvantages of a vector or SIMD design come from broadcasting control data and interprocessor communication. All these points apply directly to an SFMD design when operating in SIMD mode, but *not* when in SFMD mode, due to desynchonization of instruction execution between separate threads when in SFMD mode [AGWFH94, WHAG+92].

Finally, the improved computational density of a CMP may allow adequate performance with slower clocking. If so, tolerating memory latency also becomes less of a problem, assuming (off-chip) memory is not also clocked more slowly. The improved computational density of CMP may also allow a smaller die size for equivalent performance.

## 4.9 Conclusions

In section 4.4 we saw that for a design that does not share data pins, giving each PE its own path to external memory, simple pin restrictions limit parallelism. Section 4.5 showed, for a large class of highly parallelizable applications like model matching, that when data pins are shared, area tradeoffs imply two cases. Either (essentially) all models (task parameters) fit on-chip, or parallelism limited to $k$, the ratio of the time to match a model (execute a task) to the time to load the parameters for that model (task). In the latter case, it is not cost-effective to have many more than $k$ PEs. Due to bandwidth limitations and the difference between on- and off-chip clocks, the load time for a single parameter may be several on-chip clocks, so that to achieve a ratio $k$ will require a multiple of $k$ number of operations per parameter. This implies that a small number of PEs, around 16, is probably maximal for all but the most special-purpose architectures. So designs that rely on off-chip memory are limited to at most a small degree of parallelism. There are thus two *viable architectural possibilities.* One uses many simple processors and puts as much memory on-chip as possible, so that all models can be kept on-chip. The processors could be architected as a vector processor, or as an SIMD or SFMD array, perhaps as part of a hybrid architecture with a fast scalar processor in addition to the parallel array. The other possibility is to use a few complex processors and external memory. In this case, chip resources should be sufficiently abundant that the individual processors need not be substantially simplified from a commercial microprocessor, and an MIMD architecture seems most likely.

In section 4.6: I responded to objections to a general-purpose CMP as follows:

*"Multi-threaded programming is hard":* Perhaps, but an SIMD programming model is not,

and is appropriate for tasks with data/knowledge parallelism, such as model matching.

*"Computational density aggravates memory bandwidth problem ":* The results from Appendix B bear this out: for model matching tasks, either all models must fit in on-chip memory, or parallel speedup is strictly limited by the ratio of computation time to load time.

*"Synchronization is expensive":* The SFMD model introduced in the next chapter uses fast (global) synchronization. I will analyze the expense of this technique in the next chapter.

*"Amdahl's Law tends to favor a single fast processor over several slower processors acting in parallel":* The typical, and probably best, solution to this is to use part of the chip area for a conventional microprocessor core, to execute non-parallelizable parts of code. The microprocessor is programmed essentially independently of the parallel PE array. However, at least conceptually, SFMD allows PEs in a CMP to be of different speeds, so that a design could have a single fast PE and numerous simpler, slower ones. Programming of all PEs would be identical, with the fast PE(s) being distinguished by a testable identifier.

In conclusion, for our application domain, we see that if (essentially) all models do not fit on-chip, parallel speedup is strictly limited, and one is left with a design using a few fast processors with external memory. Such a design requires much power, for fast access to the external memory, and has the added system expense of the external memory. I feel that such a design will not outperform conventional designs sufficiently to economically justify its construction. For example, a 16 PE design, with each PE being a somewhat aggressive quad-issue superscalar architecture with an IPC of 3, has a maximum IPC of 48. This is only 4 times that projected for an advanced 16-issue superscalar. Assuming that something of equivalent performance to the wide-issue design will be built as a general purpose chip, and given the design and test cost for these chips, I expect a four-fold performance increase (at most) to be insufficient justification for building the design with a few fast processors.

Instead I will concentrate on the "all models on chip" niche, using many simple processors. The discussion of IRAM suggests that model matching and image understanding algorithms may be possible with the amount of on-chip memory available in the future. Simple processors are then desirable to leave as much area for memory as possible. Use of an SIMD/SFMD architecture simplifies processors by avoiding (most) per-PE instruction hardware, it offers a simple programming model, and SFMD improves performance over SIMD for data-dependent tasks.

# Chapter 5

# SFMD

In this chapter, I introduce the *Single Function Multiple Data (SFMD)* architecture, as an extension of SIMD, and discuss its implementation and cost. In an SFMD design, each simple processor has a small instruction buffer, so that repetitive, data-driven computations such as model-matching can be done in parallel with less load imbalance than if done in SIMD mode. The silicon cost of SFMD execution is essentially that of the instruction buffer; from the VLSI trend numbers reported in chapter 4, I conclude that this cost is not too great, perhaps requiring reducing the on-chip memory size by 3-6%. I discuss the relation to existing SIMD compilers and programming environments and illustrate that SFMD functionality can be made use of transparently to the programmer, as a compiler optimization, and that debugging SFMD is as easy as debugging SIMD. Thus, there is no additional programming cost to using SFMD compared to SIMD. I compare the performance of SFMD to that of an SIMD system, and show for a wide range of model-matching type tasks with data-dependent execution that SFMD should provide a 1.5 to 2-fold improvement in performance. In SFMD, processor synchronization is global, and I discuss the potential penalty incurred compared to an SPMD multiprocessor with point-to-point synchronization. I show that if message sending is not too infrequent and if the variance in the computation time between processors is not too large, that SFMD can be competitive with an SPMD design.

## 5.1 Overview of SIMD Architecture and Programming

As background, I first review SIMD parallelism. In SIMD, multiple processing elements, or *PE*s, simultaneously execute identical instruction sequences. Typically, each processes different data. For example, an image may be partitioned among the PEs, and each PE convolve the pixels of its part of the image with a specified mask. Alternatively, depending on how we wish to interpret the term "data", each PE may process the same data using different sets of parameters, making use of knowledge parallelism as discussed in chapter 3. For example, in vector quantization, codebook vectors may be partitioned among the PEs, and then data may be quantized by each PE finding the nearest codebook vector in its set, and then comparing among the best vectors found by each individual PE. Here I am viewing the codebook vectors as parameters of the algorithm.

Architecturally, the shared instruction stream is produced by a controller, or *sequencer*. Generally, each PE has a certain amount of *local memory*, which only it can access directly. All PEs execute a given instruction in the stream at the same time, so are synchronized at each instruction[1]. This means synchronization is implicit, the hardware need not support it, and the programmer need (can) not manage it. It also means that instruction management hardware and bandwidth is shared among the PEs, leading to simpler processors and reduced off-chip bandwidth.

SIMD architectures differ in the functionality of their PEs. If PEs can independently address local memory at differing locations, rather than all having to access the same address at a given step, the architecture is said to have *local addressing*. If PEs can independently determine whether to execute a given instruction, rather than having this determined by the sequencer, the architecture has *local conditional execution*. Branching, *per se*, is done only on the sequencer, since PEs have no control over the instruction stream they read. All PEs see the same instruction stream, yet a given PE executes the code in

[1]Such synchronization of many processors occupying more than one chip is becoming more difficult as feature size diminishes, chip size increases, and clock rates increase, as discussed in chapter 4. For this reason, SIMD architectures which tolerate slight de-synchronization are being introduced [Wee97]. These architectures behave in a strictly SIMD fashion, and do not add SFMD extensions.

only one branch of any if-then-else, and so must idle while other PEs execute the code in the other branch. This is the cost of synchronizing at each instruction, in lock step. It implies that when execution is data-dependent, some PEs may idle while others complete their work. Minimizing this idle time by balancing the work done by the individual PEs, *load balancing*, is the major issue in programming SIMD machines for applications with data-dependent execution.

## 5.2 The SPMD and SFMD Computation Models

The notion, introduced above, of partitioning the data and operating on the different parts in parallel, is called *data parallelism.* Data parallelism has been described as a parallel programming model "with much to commend it" *vis a vis* a variety of other models for practical portable parallel programming, where the dimensions of comparison include architectural independence, reduction of descriptive complexity, and the ability of a programmer to form some estimate of the performance of the executing computation [Ski91]. Data parallelism allows programmers to think in terms of familiar matrix and vector abstractions, which have a simple mapping to the architecture and execution model. One early large study of applications in science and engineering found that data parallelism was nearly always the source of parallelism in execution [Fox88, Fox89]. Since the advent of superscalar processors, instruction level parallelism would have to be added as a major source; as well, the use of special purpose hardware for handling messages or for DMA provides another. None the less, at least anecdotally, data parallelism is still the dominant form of explicitly programmed parallelism.

For SIMD execution of data parallelism, the parallel operations use identical instruction streams. Instead, one can require only that the parallel operations use the same *program*, with potentially different instruction streams due to data-dependent branching. This is called *Single Program Multiple Data (SPMD)* computation. SPMD is the natural way to implement data parallelism on a machine built from multiple general-purpose processors, and is the dominant model for programming such machines [HQ90]. In SPMD programs, processors are free to communicate with each other at any time, to the extent that this is permitted by the architecture and programming language. This introduces semantic complexities in the form of explicit synchronization and the possibility of deadlock and race conditions. It also introduces performance issues in dealing with the expense of synchronization using either interrupt code and handlers, or spin locks, and with contention for shared variables. Debugging becomes extremely complex, due to uncertainties about the order in which events occur and difficulty in reproducing events dependent on race conditions.

I introduce the *Single Function Multiple Data (SFMD)* computation model as intermediate between the SIMD and SPMD models. As an extension of SIMD, SFMD allows different processors to be executing different instructions (of the same program) at different times. A program is divided into sections of SIMD code interspersed with *SFMD blocks.* SFMD blocks, which generally are small nested loops, are executed independently on each processor. As a restriction of SPMD, SFMD does not allow processors to communicate at arbitrary times. Rather, for processors to communicate, *all* processors must synchronize in a *barrier synchronization* before any communication takes place. Thus no communication between processors occurs within an SFMD block. Also, processors cannot communicate with the sequencer during SFMD blocks. In particular, the "no-communication" rule implies that multiple processors cannot access the same variable in shared memory from within an SFMD block, as this is a form of communication. Thus, at least for memory references within an SFMD block, a distributed memory architecture must used.

As intermediate between SIMD and SPMD, two immediate questions are the comparative efficiency of SFMD versus SIMD, and of SFMD versus SPMD. I examine these in detail later on.

An essential point is that from a programmer's perspective, SFMD and SIMD are both very similar to conventional, non-parallel programming in that there is effectively a single thread of control [2]. This, in turn, implies that debugging, a major problem for more general parallelism models, is for SIMD and SFMD equivalent to debugging a non-parallel program. To see that SFMD and SIMD are equivalent in this regard, note that an SFMD program can be debugged as an SIMD program as follows: each SFMD block is executed on a single processor at a time, leaving the other processors disabled, and iterating through all processors in some order. As there is no communication between processors during the SFMD block, the sequentialized execution is equivalent to the parallel one.

A second essential point is that programming for a SFMD architecture is virtually the same as programming for a SIMD architecture with local conditional execution. From a high-level language, e.g., 'C' with SIMD extensions, the compiler can make the needed changes. Even for assembly language programming, only slight additions are needed to SIMD code to take advantage of SFMD operation.

Support for SFMD functionality can be added to an existing SIMD machine to increase its flexibility, scope, and power; in particular, as an optimization for highly data-dependent code such as indexing, model matching, and other forms of knowledge parallelism. An SFMD architecture retains two key advantages of an SIMD one: simplicity of programming and debugging, and the use of a single instruction stream. The latter means that a VLSI implementation of an SFMD architecture can share instruction processing hardware and cache among the PEs, and need only have pins for a single instruction interface. As discussed in chapter 4 smaller instruction bandwidth can be used to reduce the number of package pins, reduce cost, or can be used to increase data bandwidth by freeing up pins for data I/O. Shared instruction hardware allows PEs to be simpler, smaller and easier to design.

It remains for us to show the viability of the SFMD concept: what is the cost of implementation and what is the comparative performance of SFMD versus SIMD, and versus SFMD?

[2]Deadlock, race conditions and lack of fairness are still possible for both SFMD and SIMD, so some care must be taken in handling interprocessor communication. But the programmer's problem is much simpler due to the explicit order in which events are known to occur.

## 5.3 Implementing the SFMD Programming Model

Given a SIMD architecture with the local addressing and local conditional execution, SFMD programming can be made available at the assembly language level by adding three constructs:

`distribute n start` tells the sequencer that the next $n$ instructions are to be distributed for independent execution on the PEs. Call these next $n$ instructions an *SFMD block*. There is a similar instruction that tells the PEs to store the next $n$ instructions for independent execution; I will use the same name for both. As the $n$'th instruction is distributed, control passes to the individual PEs, with the program counter starting at `start`.

**sync** tells the individual PEs to suspend execution and signal the controller (barrier synchronization). Control passes from the individual PEs back to the sequencer for further SIMD execution. It is not necessary, but may be convenient, to allow **sync** to return a value, indicating the exit state of the PE and allowing the sequencer to take action immediately in case of certain exceptions. **sync** is ignored if not within an SFMD block.

`branch-local` encompasses one or more local branch instruction(s), including a loop construct; the branch target must lie within the enclosing SFMD block. These are ignored if not within an SFMD block.

Most importantly, I further require that *code within an SFMD block contain only references to PE-local memory*; none to global (sequencer) variables, to external memory or to the local memory of another PE. It must also contain no inter-PE communication. Note that a message may be *sent* from within an SFMD block, and routed through any interconnection network, as long as they are not *acted upon* within an SFMD block[3] . This is important

[3] It should be noted that race conditions, unfairness and starvation are possible if the the behavior of the program is dependent on the order in which messages are received. However, this is as true for SIMD execution as it is for SFMD.

and allows overlapping computation with communication.

When the PEs are independently executing an SFMD block, the system is in *SFMD mode*, and normal execution is then referred to as *SIMD mode.*

## 5.4 The SFMD Programming Environment

When programming in a data-parallel 'C'-like language designed for a SIMD architecture, the use of SFMD functionality may be an optimization performed by the compiler, completely hidden from the user. This optimization is possible when the architecture's instruction set distinguishes references to PE-local memory from references to non-local memory[4]. In this case, variable type and usage analysis can determine for any given block of code whether the constraints on non-local references are met, and emit code for SFMD execution if so.

Reads of global variables may be allowed, by copying the value into local memory before starting distributed execution. No new problems are introduced for debugging, as SFMD execution is semantically equivalent to executing on each PE sequentially, and can be executed this way during debugging.

To the programmer, SFMD ameliorates two annoyances of SIMD programming: (i) in conditionals, a PE need not be idle while other PEs execute the branch it did not take, and thus, (ii) loops and recursions may execute a processor-dependent number of times. Of course, due to the barrier synchronization, the latter is only useful for nested loops, where the inner is data dependent: for a single loop, the time is in any case the maximum of the times of the loop executing on each processor.

### 5.4.1 Translating SIMD to SFMD

In this section I give examples of how to translate SIMD code to SFMD code. Formalizing the translation would require specifying an SIMD instruction set architecture in detail,

[4]For example, in a shared memory architecture the hardware may completely hide the difference between local and non-local references, providing a transparent non-uniform memory access (NUMA) model. In such an architecture, automatic conversion of SIMD to SFMD code cannot be done.

which would take us too far afield. The examples are intended to clarify what is involved, and to further clarify the SFMD idea.

In a conventional SIMD machine, there are two instruction streams, one for the sequencer, and one for the array of PEs. Each PE sees all instructions in the instruction stream going to the PE array. There are generally a number of *reduction* operations for computing functions of results produced on the individual PEs. Let $v(p)$ be some value output by each PE $p$ as part of the computation. Reduction operations include AND-ing, OR-ing, summing, or taking the maximum or minimum of the $v(p)$. For our purposes, we can ignore reduction operations other than OR and AND as they are done in SIMD mode.

Local conditional execution is implemented by *predicating* instructions on various logical conditions occurring locally to the PE, for example, the contents of a condition code register, or the output of the adder being zero. A given PE ignores instructions whose predicates are false for it. In a minimalist implementation, the predicate can simply test a condition code register for being zero. This allows predicating the instruction to be encoded in a single bit of the instruction word. Denote an instruction, $I_k$ predicated on a condition, $C$, by $I_k^{(C)}$.

Program control flow constructs such as `for` and `while` loops are done on the sequencer. SFMD allows moving such loops from the sequencer to the PEs (if space allows). Loop termination criteria may or may not depend on results from the computation done on the PEs in the body of the loop. When the criterion does not depend on PE results, for example, in a `for` loop of fixed length, the loop may be moved entirely to the PEs. This leads us to consider `OR` and `AND` reductions which derive the termination condition on the sequencer from the termination conditions on the PEs, `done(p)`.

Consider a loop on the sequencer with body *B(p)* computed on the PE array,

```
while not(AND done(p)) B(p) endwhile.
```

Such a loop corresponds to looping until all PEs finish some set of tasks. When control is moved to the PEs, each PE executes

```
while not(done(p)) B(p) endwhile; sync;
```

and the sequencer waits for all PEs to `sync`. A loop,

```
while not(OR done(p)) B(p) endwhile,
```

corresponds to looping until the first PE finishes its set of tasks, as might be done in searching or model matching until an adequate example is found. When control is moved to the PEs, each PE executes

```
while not(done(p)) B(p) endwhile; sync;
```

and the sequencer waits for any PE to `sync`. After the first PE `sync`'s, the sequencer may interrupt the others.

The above examples illustrate how to map control flow constructs from the sequencer to the PE array. This involves the addition of `branch` and `sync` instructions. There is also the issue of transforming a stream of possibly conditionally executing PE instructions into a stream that makes use of local branching.

Consider a stream of PE instructions containing a block of conditionally executing code:

$$I_1; \ldots I_n; I_{n+1}^{(C)}; \ldots I_{n+k}^{(C)}.$$

Suppose this code corresponds to `A; if C then B` which is a SIMD idiom for `if C then B else A`, when code blocks `A` and `B` side-effect the same set of variables[5]. Then $I_1; \ldots I_n$ computes `A` and `C` and $I_{n+1}^{(C)}; \ldots I_{n+k}^{(C)}$ computes `B`.

By reachability and usage analysis, [Wol96] and by code movement to separate `A` from the computation of `C`,

$$I_1; \ldots I_n; I_{n+1}^{(C)}; \ldots I_{n+k}^{(C)}$$

becomes

$$I_{i_1}; \ldots I_{i_k}; I_{i_{k+1}}^{(\neg C)}; \ldots; I_{i_n}^{(\neg C)}; I_{n+1}^{(C)}; \ldots I_{n+k}^{(C)},$$

[5]This is a sensible idiom in the SIMD case: since all PEs must see the instructions in block `A` anyway, it does no harm to execute them and then overwrite the results. Also, with a 1-bit conditional execution predicate as described above, the `A; if C then B` formulation avoids having to compute $\neg$ `C`.

where instructions $I^{(\neg C)}_{i_{k+1}} \ldots I^{(\neg C)}_{i_n}$ compute A and the instructions $I_{i_1}; \ldots I_{i_k}$ compute the condition $C$. Insertion of branch and sync instructions then gives

$$I_{i_1}; \ldots I_{i_k}; \texttt{branch(C,L0)}; I^{(\neg C)}_{i_{k+1}}; \ldots; I^{(\neg C)}_{i_n}; \texttt{branch(L1)}; \texttt{L0:} I^{(C)}_{n+1}; \ldots I^{(C)}_{n+k}; \texttt{L1:} \quad \texttt{sync}$$

We now have a block of code containing branches. Break this code into maximal segments each containing no non-local communication. Any segment all of whose branches have targets lying in the same segment can be executed in SFMD mode. We can place the `distribute` instructions for each such segment by finding the minimal sub-segment containing all branches and branch targets of the full segment.

These examples should illustrate how both high level and assembly code may be translated to make use of SFMD functionality. Of course, this is intended simply to illustrate, there will be many details pertaining to a particular SIMD instruction set architecture.

## 5.5 Hardware Implementation and Hardware Cost

I am interested in high volume, low cost, low power, embedded, "delivery system" applications. Such systems must have few chips; scalability to 100's or 1000's of chips is not an issue. Parallelism is thus achieved with multiple PEs per chip. As we saw in chapter 4, current VLSI trends lead us to consider an architecture with many (e.g., 64 or 128) small PEs and on-chip system memory. While chips can contain many transistors, area for PE logic is at a premium since most area will be used for memory, and since use of area translates into higher power requirements. Nonetheless, as we saw, there will be area available for making PEs substantially complex; in particular there will be area available for implementing SFMD functionality.

Adding SFMD functionality to an architecture whose PEs have local addressing and local conditional execution is straightforward. Here I outline an example implementation. Hardware for branch tests and decoding sequencer instructions in the instruction register (IR) already exists. Local memory is suitable for local addressing.

A "micro-sequencer" must be added, consisting of an auto-incrementing program

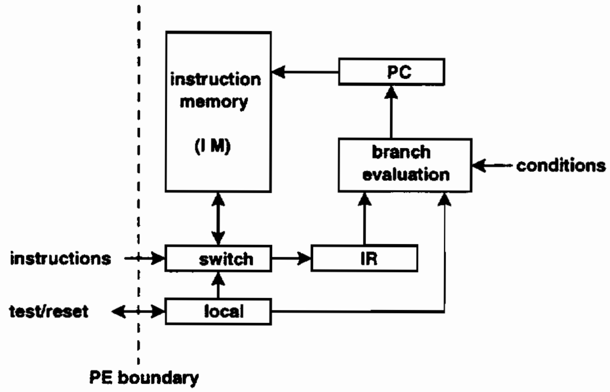


Figure 5.1: Example SFMD implementation

counter (PC), an instruction buffer (IM), decode logic for branch instructions, and state for determining mode and disabling inter-PE communication in SFMD mode. The instruction buffer can be loaded from the existing instruction bus (IB); this requires a 1-2 mux on the IB that allows instructions on the IB to be sent to the IR, to both IM and IR, or to be ignored. SFMD blocks can be stored at known locations within (small) IM, so addresses are both small, requiring few bits, and absolute, so that no branch address computation is needed.

By sending instructions to the IM and IR simultaneously, it is possible to execute the instructions (in SIMD mode) while they are being loaded, hiding the overhead of loading IM. The existing PE output path can be used for the barrier synchronization. A 1-bit path from the sequencer to each PE is added for interrupting local execution.

Execution of a `distribute n` instruction on a PE causes the next $n$ instructions to be both executed in normal mode and stored sequentially in IM, starting at the current address in the PC. The $(n + 1)$'st instruction is executed in SFMD mode, it is typically either a `branch-local` to start execution, or possibly a `sync` if the instructions are just being executed once and cached[6].

Almost the entire cost of providing SFMD functionality is silicon area used by the IM.

[6]For example, if the distributed code is a subroutine that will be encountered again.

The IM contains inner loop code, or model-driven conditional code, which is likely to be small. For example, consider the cost of adding an IM of 512 4-byte instructions to each PE of a current generation ($0.18\mu$) 64 PE chip. Taking into account a 20-fold increase in density of DRAM over SRAM (table 4.6), panel (g) of figure 4.2 shows that without IMs, such a chip could have over 100MB of DRAM memory. Adding all the IMs to all the PEs then uses $(64 \times 20 \times 2KB)/(100MB) = 2.56\%$ of the DRAM memory. So adding 2 KB of IM to each PE of a 64 PE chip reduces system memory for data and models by less than 3%. This percentage grows linearly with the size of IM and decreases linearly with the number of PEs. If dual-ported SRAM is used, as in the following section, the number increases by about 50%, to 4% of memory. In terms of chip area, using the numbers from chapter 4, for a 64 PE chip, each KB of (single ported) IM uses 1.7% of chip area in the $0.25\mu$ generation, 0.76% in the $0.18\mu$ generation, and 0.47% in the 0.13 generation.

These numbers seem substantial, but not prohibitive. We will see that SFMD execution can provide around a 1.5 to 2-fold increase in performance over SIMD execution on tasks with data-dependent control flow. For our applications, this increase seems well worth the chip area cost of a small (1-4 KB) buffer.

### 5.5.1 An alternative implementation of SFMD

An alternate implementation of the SFMD model allows us to have SFMD blocks that are larger than would fit in a given amount of local instruction memory (IM). This implementation operates on an stream of predicated instructions, including `sync`'s, but not including `distribute` and `branch`. The PE has a condition code register, and instructions can have one of two predicates, $C$ or $\neg C$, allowing the instructions to execute when the condition code register is non-zero or zero, respectively. This requires two bits in each instruction word. The techniques mentioned previously may be used to translate SIMD code to this form of SFMD code.

The idea is to replace the IM by a FIFO instruction buffer, filled at 2 instructions per clock by a double-wide bus, and emptied at 1 instruction per clock. The appropriate instructions are flushed from the FIFO without being executed when it is known that their predicate will fail. This requires two ancillary "branch" FIFOs of addresses within the instruction FIFO, pointing to the end of sequences of predicated instructions of the two types. More precisely, the branch FIFOs point to the instruction immediately following the last predicated instruction in the sequence. Call such a point a *branch target*, as that is what it effectively is, even though there are no actual branch instructions. When a predicated instruction reaches the end of the FIFO and the predicate fails, instructions up to next branch target are flushed without executing, effectively performing a branch. Effectively, the hardware "discovers" branches on the fly.

For each PE, this design needs the instruction and branch FIFOs as well as a small finite state machine for handling full and empty FIFO conditions, and for determining "flush" conditions based on predicate and PE state. As the branch FIFO can be small, containing a small number of addresses, each of at most probably 16 bits, the instruction FIFO has the dominant area cost. As well as `sync`, this design requires simple global communication, much the same as barrier synchronization, for handling "full FIFO" conditions.

On the downside, this implementation is somewhat complicated and there will be inefficiency when a full buffer condition on one PE stalls the instruction distribution and perhaps stalls other PEs. There will probably be a 1 to 2 cycle branch delay for flushing the FIFO, although this might sometimes be masked by an analog of a branch delay slot, possibly with multiple instructions, where the first 1 or 2 instructions preceding predicated instructions are guaranteed to not affect the condition code register, by use of no-ops, if necessary. For very small branch bodies, it may be preferable to use predicated instructions in the normal SIMD fashion. For example, two instruction word bits are used for predicates: `00` indicates a normal SIMD unpredicated instruction, `01` indicates a normal SIMD predicated instruction that is executed if the condition code register is nonzero, but is ignored by the branching mechanism, `10` and `01` indicate the two SFMD predicates.

Detection of full FIFO conditions can be done some number of instructions early, so that delays in communicating with the central instruction distribution mechanism do not require killing and resending instructions sent before the "full FIFO" message is received and acted on. It is possible, if unlikely, that a branch may extend past the end of the FIFO, that is, all instructions in the FIFO have the same predicate. If this branch is taken, the entire FIFO is emptied, and further instructions with that predicate will be processed (flushed) more slowly, as they are delivered.

Finally, with this design, speedup over SIMD is limited to at most twofold, as that is the instruction delivery rate, however, we will see that SFMD generally can be expected to give at most a 1.5 to 2-fold improvement over SIMD.

The advantage of this design is that there is no limitation on block size, as there is when the entire block must fit in the local IM. Also, due to flushing of untaken branches, and continuous emptying of the FIFO, an instruction FIFO holding a given number, $n$, of instructions should generally support, without too much inefficiency due to empty or full FIFO conditions, SFMD blocks which are considerably longer than $n$. With the implementation described in the previous section, an IM holding $n$ instructions is limited to SFMD blocks of length $n$. With respect to area, this advantage of the FIFO implementation is reduced by the fact that the FIFO uses dual-ported memory, and thus requires about 1.5 times as much area as an IM to implement the same number of instructions.

As the hardware hides the FIFO management tasks, the programming complexity of both implementations of SFMD is the same, except that the FIFO implementation may allow for more sophisticated performance tuning, to avoid "full FIFO" conditions. However, even this will be similar to programming the non-FIFO implementation to fit SFMD blocks within the available instruction memory.

## 5.6 Performance Improvement of SFMD versus SIMD

What performance improvement may be expected by adding SFMD to SIMD? There are two basic components, improvement on branches, and improvement on nested loops, where the inner loop count varies locally.

Unnested (equiprobable) branches speed up most when the branch bodies have the same size, with a factor of 2 improvement. For nested branches of depth $d$, the factor is $2^d$, but these are probably unusual for $d > 3$. An exception would be applying a decision tree classifier in a data-parallel way, as in [BD94].

To examine improvement on nested loops, suppose we have a set of $N$ models (or any independent tasks) to be evaluated on an architecture with $P$ processors. On an SFMD architecture, partition the set into $P$ groups, assign each group to a processor, and have each processor evaluate all the models in its group. If evaluating the $j$'th model of the $i$'th group takes time $t_{ij}^{(sfmd)}$, then the total time is

$$T_{sfmd} = \max_{i=1}^{P} \sum_{j=1}^{N_i} t_{ij}^{(sfmd)} \tag{5.1}$$

where $N_i$ is the size of the $i$'th group, $\sum_{i=1}^{P} N_i = N$, and the *(sfmd)* superscript notes that individual models may take different times to execute on SIMD and SFMD machines due to data-dependent branching.

On an SIMD architecture, partition the set into $\lceil N/P \rceil$ groups of size $P$ and sequentially evaluate each group in parallel. Each group has a model that takes the most time to evaluate; SIMD execution forces the whole group to have this time complexity. So, evaluating a single group, $G_i$, takes time $t_{ij}^{(simd)}$, where $j$ indexes over the elements of the group, $1 \leq j \leq P$. The total time for SIMD execution is then

$$T_{simd} = \sum_{i=1}^{\lceil N/P \rceil} \max_{j=1}^{P} t_{ij}^{(simd)} \tag{5.2}$$

Ignoring data-dependent branching and taking $t_{ij}^{(sfmd)} = t_{ij}^{(simd)} \doteq t_{ij}$, we see that optimal $(i, j)$-indexing of the $N$ models for either case is a bin packing problem. As such, $(i, j)$-indexing will be heuristic, and I examine $T_{simd}/T_{sfmd}$ by simulation. It should be clear that the expected improvement of SFMD over SIMD cannot be large unless the outer loop count is large or the variance in $t_{ij}$ is large. Relaxation-based algorithms may have large variance (see, for example [CKP95]). When indexing is possible, one would not expect the outer loop count to be large, but it may be for non-indexable models such as elastic ones.

To examine the possible magnitude of the effect in general, I look instead at multiplication of an input vector by a large sparse matrix. Rows are partitioned among the PEs, and each PE computes all the row-vector inner products for its set of rows[7]. $T_{sfmd}$ is given by equation 5.1, with $\{t_{ij} | 1 \leq j \leq N_i\}$ the set of all rows for processor $i$. $T_{simd}$ is given by equation 5.2, with $\{t_{ij} | 1 \leq j \leq P\}$ the set of rows executed by all processors at time $i$. Here $t_{ij}$ is the time to perform a row-vector inner product. Note that, to examine specifically the relationship of max-of-sum and sum-of-max, I am assuming $t_{ij}$ is the same for both SIMD and SFMD. In fact, $t_{ij}$ will differ between SIMD and SFMD execution, due to conditional execution depending on how many non-zero row elements a given PE has. Later analysis and simulation will examine the effect of such data-dependent execution on $T_{simd}/T_{sfmd}$.

Under a variety of choices of matrix size ($256 \times 256$ to $2048 \times 2048$), number of processors (16,32,64), distribution of elements (uniform, clustered around the diagonal), and sparsity (fraction of nonzero elements from 0.001 to 0.4) we get that the ratio $T_{simd}/T_{sfmd}$ decreases from around 2.2-2.7 for sparsities near 0.001, to 1.2 for sparsities near 0.06, and to 1.1 or less for more dense matrices (figure 5.2). The effect is thus significant but not dramatic. Note however that handling sparse matrices is difficult for vector and superscalar designs, as pipelines are rendered useless. Thus even a small improvement may be significant.

### 5.6.1 Analysis: sum-of-max vs max-of-sum

To compare equations 5.2 and 5.1 analytically, I will take $t_{ij}^{(sfmd)} = t_{ij}^{(sfmd)} \doteq t_{ij}$ to be independent realizations of a random variable, $T$. For a random variable, $X$, denote its mean and variance by $\mu_X$ and $\sigma_X^2$, respectively. Assume $\mu_T < \infty$ and $\sigma_T^2 < \infty$, and define

[7] I assume the assignment of rows to PEs is independent of the number of nonzero elements in the rows. If not, then for $N \gg P$, simply sorting rows by number of elements and then assigning row $i$ to processor $i \bmod P$ can be a good enough packing heuristic to make $T_{simd} \approx T_{sfmd}$. Permuting the rows this way may require the outputs to be reverse permuted, which will be too costly for some applications.

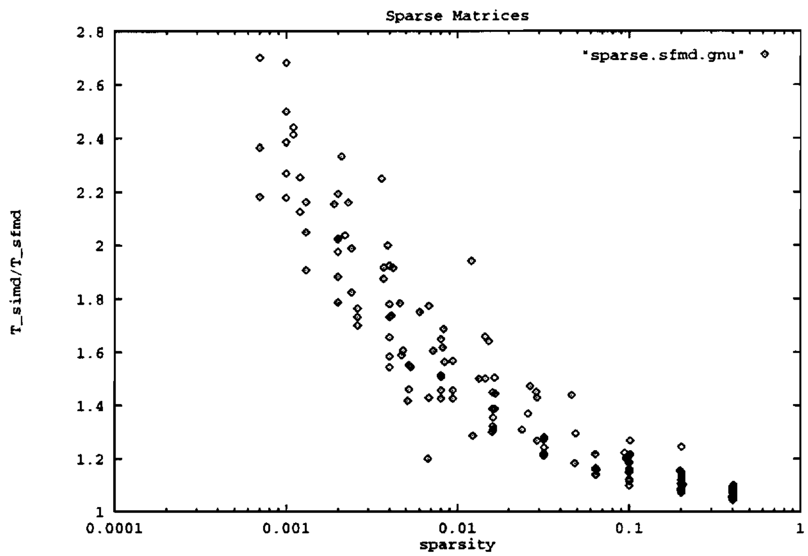


Figure 5.2: Sparse matrices: speedup vs. sparsity

$S = (T - \mu)/\sigma$ to be the normalized version of T. For a distribution, $X$, and fixed sample size, $n$, one may consider the distribution, $X_{(n)}$, of the largest value in a random sample of size $n$. I assume that $T_{(n)}$ is well defined. For simplicity, assume $N$ is a multiple of $P$, $N = MP$ in equations 5.2 and 5.1. Let $MAXSUM$ be the random variable $\max_{i=1}^{P} \sum_{j=1}^{M} T_{ij}$ and $SUMMAX$ be the random variable $\sum_{j=1}^{M} \max_{i=1}^{P} T_{ij}$.

For large $M$, by the central limit theorem,

$$\begin{aligned} MAXSUM &\doteq \max_{i=1}^{P} \sum_{j=1}^{M} T_{ij} \\ &\asymp M \max_{i=1}^{P} G_i \\ &= M\mu_T + \sigma_T \sqrt{M} \max_{i=1}^{P} Z_i \\ &= M\mu_T + \sigma_T \sqrt{M} Z_{(P)} \end{aligned}$$

where $G_i \sim \mathcal{N}(\mu_T, \sigma_T/\sqrt(M))$ and $Z_i \sim \mathcal{N}(0,1)$ are normally distributed.

Similarly,

$$SUMMAX \doteq \sum_{j=1}^{M} \max_{i=1}^{P} T_{ij}$$

Table 5.1: Representative values of $\mu_{Z_{(P)}}$, the expected value of the maximum of a random sample of size $P$ from a standard normal distribution. Values marked with (†) have been approximated using equation 5.8.

| $P$ | $\mu_{Z_{(P)}}$ | $P$ | $\mu_{Z_{(P)}}$ | $P$ | $\mu_{Z_{(P)}}$ |
|---|---|---|---|---|---|
| 8 | 1.424 | 50 | 2.249 | 128† | 2.598 |
| 16 | 1.766 | 64† | 2.335 | 500 | 3.037 |
| 32 | 2.070 | 100 | 2.508 | | |

$$\begin{aligned} &\asymp MY \\ &= M\mu_{T_{(P)}} + \sqrt{M}\sigma_{T_{(P)}} Z \end{aligned}$$

where $Y \sim \mathcal{N}(\mu_{T_{(P)}}, \sigma_{T_{(P)}}/\sqrt{M})$ and $Z \sim \mathcal{N}(0,1)$ are normally distributed.

Evaluating the expected value $\left\langle \frac{SUMMAX}{MAXSUM} \right\rangle$ seems difficult. One approximation is

$$\left\langle \frac{SUMMAX}{MAXSUM} \right\rangle \approx \frac{\langle SUMMAX \rangle}{\langle MAXSUM \rangle} \tag{5.3}$$

$$= \frac{\mu_{T_{(P)}}}{\mu_T + \sigma_T \mu_{Z_{(P)}}/\sqrt{M}} \tag{5.4}$$

This gives the obvious approximation

$$\left\langle \frac{SUMMAX}{MAXSUM} \right\rangle \approx \frac{\textit{expected maximum value}}{\textit{expected (average) value}}, \tag{5.5}$$

asymptotically for large $M$. If $T$ is normally distributed, $T \sim \mathcal{N}(\mu, \sigma)$, then $\mu_{T_{(P)}} = \mu + \sigma\mu_{Z_{(P)}}$, and

$$\frac{\langle SUMMAX \rangle}{\langle MAXSUM \rangle} = \frac{\mu + \mu_{Z_{(P)}}\sigma}{\mu + \mu_{Z_{(P)}}\sigma/\sqrt{M}} \tag{5.6}$$

If $T$ is uniformly distributed, $T \sim \mathit{Unif}[m - s/2, m + s/2]$, then $\mu_T = m$ and $\sigma_T = \sqrt{s^3/12}$. Let $V \sim \mathit{Unif}[0,1]$; it is known that $\mu_{V_{(P)}} = P/(P+1)$[Dav70]. Using $\max T = (m - s/2) + s \max V$, one obtains

$$\frac{\langle SUMMAX \rangle}{\langle MAXSUM \rangle} = \frac{m + \frac{s}{2}\left(\frac{P-1}{P+1}\right)}{m + \mu_{Z_{(P)}}\sqrt{s^3/(12M)}}. \tag{5.7}$$

There is no known closed form for $\mu_{Z_{(P)}}$; Table 5.1 gives some representative values. Although closed forms are not known for most distributions, there are a variety of approximations and bounds. I use the approximation

$$\mu_{Z_{(P)}} \approx \Phi^{-1}\left(\frac{P-0.4}{P+0.2}\right) \tag{5.8}$$

([Dav70] p. 67), where $\Phi$ is the cumulative distribution function (*cdf*) of the standard normal distribution, in the construction of table 5.6.1.

As the $t_{ij}$ represent times, they should be positive, hence I wish to consider distributions with positive support, such as the exponential, or, more generally, the gamma distribution. Let $\Gamma_\alpha(x)$ denote the cdf of the gamma distribution with parameter $\alpha$, so $\Gamma'_\alpha(x) = (1/\Gamma(\alpha))x^{1-\alpha}e^{-x}$. If $T \sim \Gamma_\alpha$, then $\mu_T = \alpha$, $\sigma_T = \sqrt{\alpha}$, and it is known ([Dav70]) that

$$\mu_{T_{(P)}} \geq \begin{cases} \Gamma_\alpha^{-1}(\frac{P}{P+1}) & \text{when } \alpha \leq 1 \\ \Gamma_\alpha^{-1}(\frac{P-1}{P}) & \text{when } \alpha > 1. \end{cases} \tag{5.9}$$

Thus

$$\frac{\langle SUMMAX\rangle}{\langle MAXSUM\rangle} \geq \frac{\Gamma_\alpha^{-1}(1-\beta_\alpha)}{\alpha + \mu_{Z_{(P)}}\sqrt{\alpha/M}} \tag{5.10}$$

where

$$\beta_\alpha \doteq \begin{cases} \frac{1}{P+1} & \text{when } \alpha \leq 1 \\ \frac{1}{P} & \text{when } \alpha > 1. \end{cases} \tag{5.11}$$

Figures 5.3 and 5.4 show some representative graphs of equations 5.6, 5.7, and 5.10. Examination of these equations show that they are relatively insensitive to the number of processors varying from $P = 16$ to $P = 128$. The most sensitive is for the normal distribution where the ratio varies from about 1.7 for $P = 16$ to 2.0 for $P = 128$. So use of $P = 16$ provides a reasonable approximation in this range.

Since the $T_{ij}$ should be positive, $\sigma_T$ has been restricted so that this is true for the uniform case ($m > s/2$), and almost always true for the normal case ($\mu_T > 2\sigma_T$). Simulations indicate that, for the normal and uniform distributions and these parameter values, the approximation 5.3 is very close. For the gamma distribution, I graph the lower bound on $\langle SUMMAX\rangle/\langle MAXSUM\rangle$ given by equation 5.10, and the difference between it and $\langle SUMMAX/MAXSUM\rangle$, estimated over 30 simulation runs.

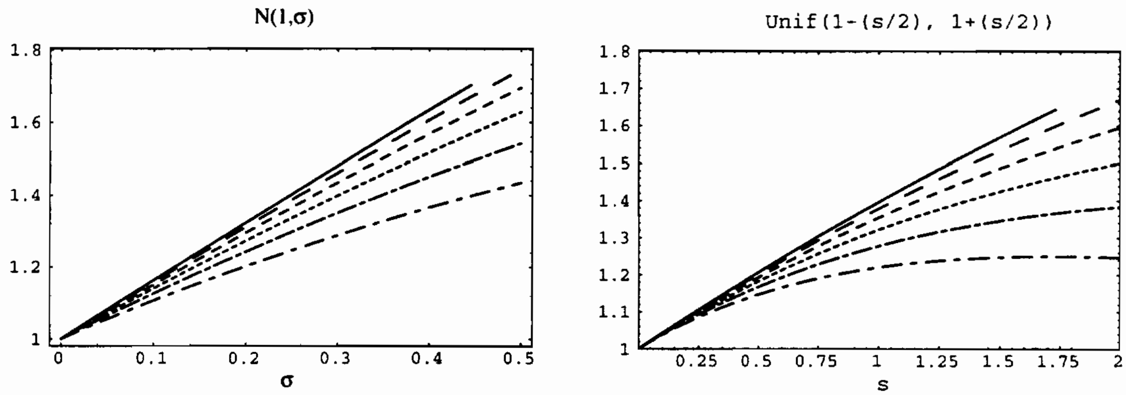


Figure 5.3: Representative graphs of $\langle SUMMAX\rangle/\langle MAXSUM\rangle$ for the normal distribution, $\mathcal{N}(1,\sigma)$, and the uniform distribution, Unif$(1-s/2, 1+s/2)$. Here, the number of processors, $P$, is 16, and $\sigma$ (respectively, $s$) varies. For the normal distribution, $\sigma$ has been chosen so that the probability of a negative $t_{ij}$ is small. Different curves are for different values of $M$, the number of loop bodies per processor; $M \in \{8, 16, 32, 64, 128, 256\}$, in order from bottom curve to top curve. Simulation results show these are good approximations for $\langle SUMMAX/MAXSUM\rangle$; the simulation results are not shown.

We see that the ratio is not too large; generally $1.3 - 2$ for not too small $\sigma$ and $M$, except for the $\Gamma_1$ (exponential) distribution, where it is substantially larger.

### 5.6.2 Code Transformations for sum-of-max versus max-of-sum

There is a code transformation due to van Hanxleden [vHK92] for dealing with the "max-sum versus. sum-max" problem. The idea is that a pair of nested loops can be converted into a single loop, so that the processors execute the body of the code together in lockstep, but using different values for the loop variables, effectively executing different steps of the iteration. A simple version of the transformation is shown in figure 5.5.

This transformation applies generally to nested loops, being of interest to us when one or both loops execute a data dependent number of iterations. Some examples are sparse matrix computations, and matching a set of structured models by having the outer loop be over the set of models, while the inner loop traverses the model. Another alternative, for graph or tree-structured models matched by some version of depth-first search, would

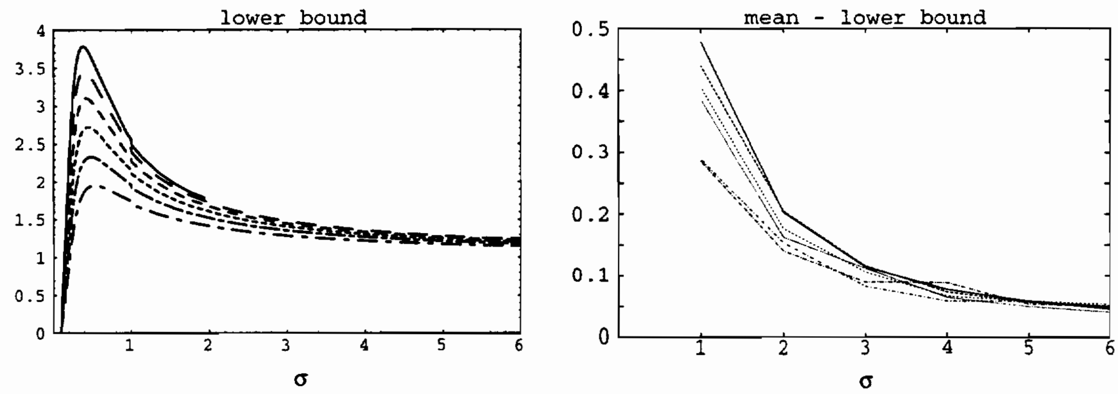


Figure 5.4: Left panel shows graph of lower bound of $\langle SUMMAX\rangle / \langle MAXSUM\rangle$ for gamma distribution, $\Gamma_\alpha$. Here $\mu_T = \alpha$, $P = 16$, and $\sigma = \sqrt{\alpha}$ varies. Right panel gives difference *(mean - lower bound)*, where *mean* is the mean of *SUMMAX/MAXSUM* over 30 trials. Different curves are for different values of $M$, the number of loop bodies per processor, $M \in \{8, 16, 32, 64, 128, 256\}$, in order from bottom to top..

```
(original)                        (flattened)
-------------------               ------------------------

init_1                            init_1
                                  init_2
while test_1                      while test_1
  {                                 {
    init_2                            <BODY>
    while test_2                      increment_2
      {                               if !test_2
        <BODY>                          {
        increment_2                       increment_1    #
      }                                   init_2         #
    increment_1                         }
  }                                 }
```

Figure 5.5: van Hanxleden's *loop flattening* transformation, from figure 11 of [vHK92]. The original code is on the left, and the flattened code is on the right. This is a simplified version of the transformation that assumes `test_1`, `test_2`, and `init_2` have no side effects, and, for each outer loop iteration, the inner loop is executed at least once. The code on the right marked with "#", is the conditional block that, in SIMD execution will be traversed by all PEs, but executed only by some, while in SFMD execution will be traversed only if executed.

be to have the outer loop iterate over possible root to leaf paths, and have the inner loop build and traverse the individual paths.

Clearly, from the point of view of SIMD execution, the transform converts the variation between inner loop counts into conditional execution of parts of the new loop body. For such transformed code, the $T_{simd}/T_{sfmd}$ is determined by the time spent in the conditional code, marked with a "#" in the figure. If $T_{cond}$ is the number of instructions in the conditional code, `increment_1` and `init_2`, $T_{common}$ is the number of instructions in common, and $p_{cond}$ is the probability of executing the conditional code (`test_2` fails), then

$$\frac{T_{simd}}{T_{sfmd}} = \frac{T_{common} + T_{cond}}{T_{common} + p_{cond} \cdot T_{cond}}$$

In situations where the expected time spent executing `<BODY>` is the same for both SIMD and SFMD and also dominates the expected time for executing the conditional block, the transform should give results as in section 5.6.1. Indeed, experimental results from the original paper [vHK92], where the conditional code is a simple loop variable increment and initialize, and the body is the call to a comparatively large subroutine, show performance improvements in the 1.2 – 2.4x range.

On the other hand, as we saw in chapter 3 (as stylized fact **F3**), model matching and many other irregular computations require very simple and compact `<BODY>`s. Examples of this were bodies consisting of probability lookup and accumulate, mapped over a graph traversal (HMMs and relaxation networks), nearest neighbor calculations using a *k-d* tree (back-projection and verification), and attribute comparisons mapped over a list or tree of features or over an object model (interpretation tree search, grouping). Also, tree and graph searches, where the `<BODY>` performs the per-node computation, must have `<BODY>`'s that include data-conditional computation, based on the type of node, for example, interior or leaf node. Here one expects SFMD to significantly outperform SIMD as the time saved by ignoring the conditional code is significant.

To make these ideas about the effectiveness of the transformation in more irregular situations more concrete, I look in detail at two versions of sparse matrix – vector multiplication. Figure 5.6 shows code for multiplication of a dense vector by a sparse matrix, both already stored in the individual PEs local memory. Parallelism is obtained by partitioning the matrix rows among the PEs and computing row-vector products simultaneously. Analysis of the corresponding assembly language code produced by an optimizing compiler for the flattened version gives $T_{common} = 12$ and $T_{cond} = 5$, where the value given for the time is the number of assembly instructions comprising the appropriate part of the code. The conditional part of the code is entered once per row, so $p_{cond}$ is the number of rows per PE divided by the number of nonzero elements per PE; under assumptions of uniform distribution of elements and rows among the PEs, $p_{cond}$ is thus one over the average number of nonzero elements per row, $\langle \textit{row length} \rangle$, and

$$\frac{T_{simd}}{T_{sfmd}} = \frac{17}{12 + 5/\langle \textit{row length} \rangle}.$$

The expected speedup is about 1.5 for all but the sparsest matrices.

Figure 5.7 shows code for multiplication of a sparse vector by a sparse matrix, where the matrix rows are stored in the individual PEs local memory, and the vector elements are being broadcast one by one. Here, a more elaborate version of the flattening transformation is required, and there are two separate $p_{cond}$'s, for the `while(mi > vi)` and `if(mi == vi)` conditions. The first of this succeeds exactly once per row, giving a $p_{cond}$ of $1/\langle \textit{row length} \rangle$, while the second succeeds whenever a row contains a nonzero element at the desired column. Under uniform distribution assumptions, this gives a $p_{cond}$ of $1/(\textit{number of columns})$. Counting assembly language instructions as before gives

$$\frac{T_{simd}}{T_{sfmd}} = \frac{28}{13 + \frac{7}{\langle \textit{row length} \rangle} + \frac{8}{(\textit{number of columns})}}$$

So the expected speedup is about 2 for all but the smallest, sparsest matrices. So, together with the results shown in figure 5.2, we see that either with or without loop flattening, SFMD gives about a 1.5 – 2 -fold speedup.

As an example of the potential utility of SFMD functionality for model matching, I consider *interpretation tree search (ITS)*, a technique used in vision, described in chapter

```
void unflattenedDenseV()
{
  int r, i, sum, imax, nr;

  /* init_1 */
  nr = NumberOfRows;
  r=0;



  /* test_1 */
  while(r < nr)
    {
      /* init_2 */
      i = RowStart[r];
      imax = RowStart[r+1];
      sum = 0;

      /* test_2 */
      while(i<imax)
        {
          /* <BODY> */
          sum += MatrixValue[i]
                 * Vector[Column[i]];
          /* increment_2 */
          i++;
        }
      /* increment_1 */
      RowSum[r] = sum;
      r++;
    }
}
```

```
void flattenedDenseV()
  {
    int r, sum, i, imax, ms;


    nr = NumberOfRows;
    r = 0;
    imax = RowStart[r+1];
    sum = 0;
    i = 0;

    while(r < nr)
      {

        sum += MatrixValue[i]
               * Vector[Column[i]];
        i++;


        if (i >= imax)
          {

            RowSum[r] = sum;
            sum = 0;

            r++;
            imax = RowStart[r+1];

          }

      }
  }
```

Figure 5.6: Loop flattening transformation applied to multiplication of a dense vector by a sparse matrix. Assembly listings for `unflattenedDenseV` have 29 instructions comprising the nested `while` loops, not including one-time setup and cleanup code, while `flattenedDenseV` has 17 (including loop overhead). For the latter, $T_{common} = 12$ and $T_{cond} = 5$.

```
void unflattenedSparseV(int vi, int vv)
{
  int r, ri, mi, nr;
  nr = NumberOfRows;
  r = 0;
  while(r < nr)
    {
      ri = RowIndex[r];
      mi = MatrixIndex[ri];
      while(mi <= vi && ri < RIdx[r+1])
        {
          if (mi == vi)
            RowSum[ri] += MatrixValue[ri] * vv;
          ri++;
          mi = MatrixIndex[ri];
        }
      r++;
    }
}

void flattenedSparseVOpt(int vi, int vv)
{
  int r, ri, mi, nr;
  nr = NumberOfRows;
  r = 0;
  mi = MatrixIndex[ri = RowIndex[r]];
  while(r < nr)
    {
      while(mi > vi)
        {
          r++;
          if (r == nr)
            return;
          mi = MatrixIndex[ri = RowIndex[r]];
        }
      if (mi == vi)
        RowSum[r] += MatrixValue[ri] * vv;
      mi = MatrixIndex[++ri];
    }
}
```

Figure 5.7: Loop flattening transformation applied to multiplication of an element of a sparse vector by appropriate elements of a sparse matrix. The flattened version requires a more complicated version of the transformation, and has been slightly hand optimized. Assembly listings for `unflattenedSparseV` have 30 instructions comprising the nested `while` loops, not including one-time setup and cleanup code, while `flattenedSparseVOpt` has 28 (including loop overhead). For the latter, $T_{common} = 13$ and $T_{cond} = 15$.

3. [8] To review, ITS is a technique for establishing a correspondence between image and model features. It consists essentially of depth-first search (*DFS*), where a node on level $d$ of the tree corresponds to a pairing of image features with the first $d$ model features. The search is limited by a variety of unary and binary geometric constraints on the allowed pairings. Search complexity implies small models are matched to small numbers of data features, so distributing models and data to local memories is practical.

To examine the effect of SFMD on this form of model matching, I performed some simple simulations. To match a model with $D$ features to a set of $B$ data points, we attempt to match the first model feature with each data point in order, with some probability of success, $p_{match}$. If we succeed, we attempt to match the second model feature with one of the remaining $B-1$ data points, and so on. If we match all $D$ features, we then check for global consistency of the correspondence, with some probability of success, $p_{check}$. This procedure is equivalent to DFS in a tree with branching factor $B-d$ at level $d$ of the tree, $1 \leq d \leq D$, where the probability of expanding any given node is $p_{match}$, and the probability of stopping the search at any given leaf is $1-p_{check}$.

By writing the search as an iteration managing an explicit stack, one obtains a loop with some common code and some code conditional on whether the current node has any child nodes left to be expanded. The bulk of the "no-child" code deals with leaf nodes, consisting of testing for global consistency and recording solutions. The relative performance of SIMD and SFMD thus depends mainly on the probability, $p_{leaf}$, that the node being traversed is a leaf. If, for each iteration, the time for the leaf code is taken to be 1, that for common code is $t$, and that for the non-leaf code is $k$, then

$$T_{simd}/T_{sfmd} = \frac{t+k+1}{t+(1-p_{leaf})k+p_{leaf}\cdot 1}. \tag{5.12}$$

Panel 1 of figure 5.8 shows values of $p$ from a variety of simulations of ITS, with $B, D \in \{8, 10, 12, 14, 16\}, p_{match} \in \{0.1, 0.2, 1/B\}, p_{check} \in \{0, 1\}$. Grimson [Gri90] reports searches on realistic data of around 5000-10000 expansions; this corresponds to $p_{leaf} \approx$

[8] See [Gri90] for a complete description of ITS and for the complexity results alluded to here.

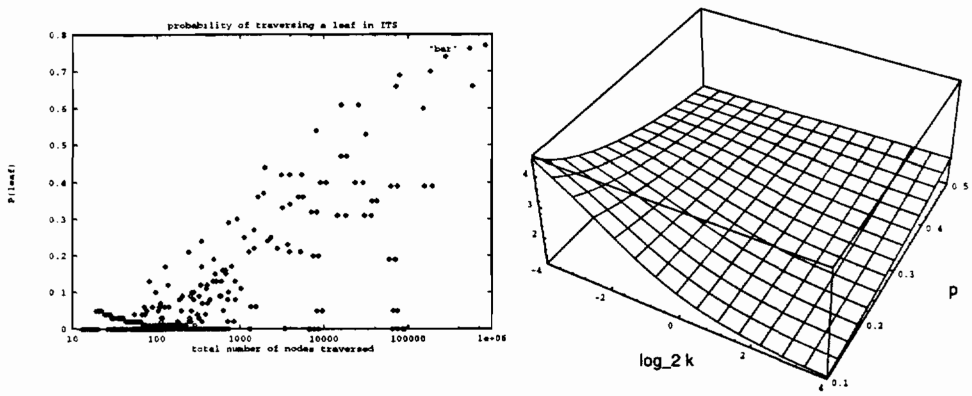


Figure 5.8: Interpretation Tree Search speedup. Panel 1 shows the probability, $p_{leaf}$, of traversing a leaf. Panel 2 plots equation 5.12 for realistic values of $p_{leaf}$ and $k$, with $t = 0.1$.

$0.2 - 0.4$. Panel 2 of figure 5.8 shows how equation 5.12 behaves for $p_{leaf}$ in this regime and for realistic values of $k$. We see speedups in the range 2–4 unless the leaf code is very small. In fact, the code for global consistency checking is typically larger than that for local consistency, corresponding to $\log_2 k < 0$.

### 5.6.3 Summary

We see that SFMD can provide substantial, about 1.5 to 2 -fold, performance improvements over SIMD for model-matching type tasks. For large nested data-dependent loops, this can be given by the reduction from sum-of-max to max-of-sum. When the inter-loop variability is suppressed, as by the van Hanxleden flattening transformation, intra-loop body conditionality appears in order to handle end-of-loop (or leaf node) situations. In this case, for small loop bodies, or, more generally, for loop bodies where the code for handling the end-of-loop case is substantial compared to normal case, SFMD gives a performance improvement over SIMD by bypassing the conditional code. As we saw in chapter 3, this latter case, of pieces of code which are both executed conditionally on the data and relatively large compared to the non-conditional part of the loop body, is typical of both model-matching and ILV tasks.

## 5.7 The Price of "No Communication"

SFMD extends the SIMD model by allowing independent conditional execution. Conversely, it restricts the usual SPMD model by allowing communication to occur only after a barrier synchronization. It makes the latter restriction in order to preserve SIMD semantics and ease of programming and debugging.

SFMD is a natural approach to model-matching tasks, which tend to have restricted and predictable communication patterns, suitable for this "bulk-synchronous" execution style. Exhaustive model matching, for example word-spotting or vector quantization, only requires communication after each model is matched, to determine the maximum or minimum. Branch-and-bound scenarios, such as matching of deformable models, only require communication after each evaluation, to propagate the new bound. Both of these can be done with a simple broadcast or "global max" communication, and do not need sophisticated networks or message-passing systems. However, many "tiled" ILV algorithms have a permutation communication pattern, where each PE simultaneously talks to its neighbor in a particular direction, known in advance as part of the algorithm. Unlike SIMD, in the SFMD model the *times* of these communications may be unpredictable. Other ILV algorithms, in particular feature grouping, involve even more random communication patterns: many-to-many at unpredictable times with unpredictable targets for the messages from a given source.

Even with unpredictable communication, there is a performance tradeoff between doing pairwise synchronization at arbitrary points and delaying synchronization to the end of a block and then communicating "in bulk" . There are a number of effects involved in this tradeoff. With communication at predictable times, and with predictable network latencies, the compiler can sometimes reduce the number of synchronizations that would be needed by a pure MIMD implementation [DZO92, BCJ90]. With predictable communications the compiler may also be able to move code so as to overlap more computation with communication. The main effect, however, is that synchronization at unpredictable points has an additional cost, for either polling or handling interrupts. The interrupt handling involves some mechanism to detect interrupts, branching to code for handling the interrupt, as well as saving and restoring the state of the computation. In our realm of small, tightly coded loops, and limited per-PE code memory, this is a lot of overhead, and it makes more sense to use a polling scheme incorporated into the code. The next section describes a simple model, analyzed and simulated to compare the tradeoff between bulk synchronization and synchronization at random points done with polling.

### 5.7.1 Communication Simulation

I examine the effect of limiting communication to be "bulk-synchronous" by use of the following model. Suppose a number of processors are iteratively executing the same block of code. At some point in each block, the processor executing the block conditionally sends a message to some processor chosen at random according to some probability distribution. The chosen receiver may be the same as the sender, in which case a message is not sent, but the computation requested by the message is still performed. This model is reasonable for model-matching and ILV-type tasks where the data (image) needed to evaluate a model is distributed among the processors in a way not known in advance, for example, feature grouping. Refer to this as the *random* case.

Another reasonable case is where the communication pattern is such that no processor receives more than one message originating from a given iteration, including itself. This pattern occurs, for example, when processors partition two-dimensional data and, in a given block, all request data from a neighboring processor in the same direction. Although all processors need not participate, refer to this as the *permutation* case.

In both these cases, the requesting processor may proceed after submitting the message, the *asynchronous* case, or it may block, waiting for the reply, the *synchronous* case. I will only look at the asynchronous case, as it allows more overlap of computation with communication, and is the less favorable to SFMD. Processors may send messages at most once per block, at a time given by a random variable. In the SPMD case, things proceed as follows (see left panel of figure 5.9):

- to send a message, the message is put on the processor's *send queue* after which an *autonomous asynchronous network* routes the message to another processor's *receive queue*
- processors *poll* with some frequency, looking for messages on their receive queue
- as soon a message is received, current processing is interrupted and a reply is computed and sent
- no polling or interruption occurs while the reply is being calculated

I will assume for simplicity that the processor sending a message can perform other useful work until either the reply is received or the end of the current block is reached. Of course, it continues polling in any case, and can be interrupted at any time to reply to a message. This model is appropriate, for example, when, as for two-dimensionally tiled data, the message and receiver are known statically, and a message can be sent in advance, requesting the data that will be needed for a future iteration.

In the SFMD case, things proceed similarly (right panel of figure 5.9), except that barrier synchronizations must be introduced before any messages are *read*. In particular, the SFMD model will also *send* the message before the barrier synchronization; the "no communication" restriction only requires that all processors must wait at the barrier before *receiving* and acting on the message. If multiple messages are sent to the same processor in a given block, all replies may be sent before synchronizing at the barrier. Assume that, immediately after the barrier, the processors check whether any messages have been posted, and if not, the computation proceeds without a reply phase. This can be determined quickly using the barrier synchronization hardware. If the target of a message is the same processor that sent it, the reply is computed after the barrier, in the reply phase. This would be the probable coding, especially for the permutation case, unless messages were very infrequent, as the reply phase usually occurs.

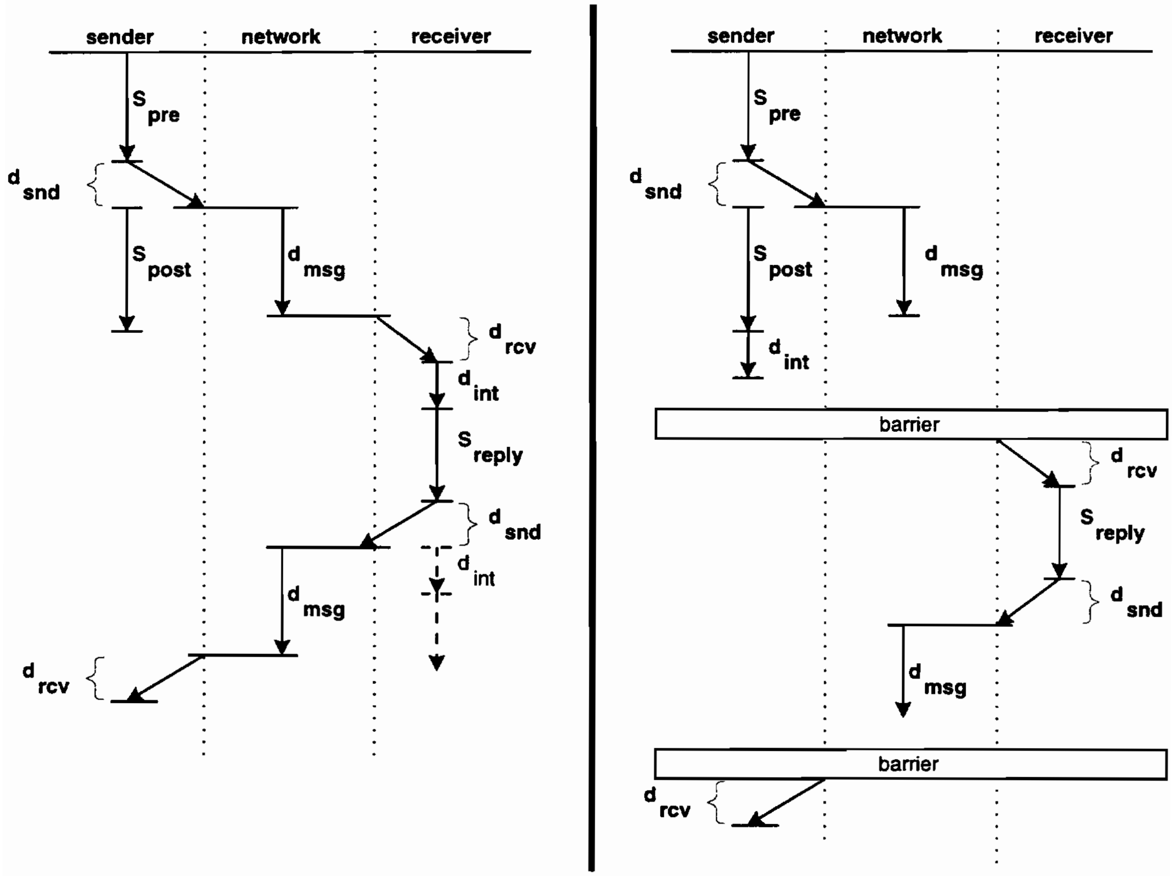


Figure 5.9: Message-passing model for SPMD (left) and SFMD (right)

As mentioned above, messages are dequeued, routed, and enqueued by a DMA-like "autonomous asynchronous network". Discussion of this network is outside the scope of the thesis. I treat the network as a black box, assuming only that there are no network congestion problems, and that messages of the same length take the same time to deliver. Also, the network must participate in the barrier, signaling that all outstanding messages have been delivered, to ensure correct semantics.

Using the notation in table 5.2, suppressing the $b$ and $p$ parameters for clarity, the time for a single block on a single processor under the SPMD model (left panel of figure

Table 5.2: Definitions for "no-communication" simulation

| | |
|---|---|
| | *constants* |
| $P$ | number of processors |
| $N$ | number of iterations |
| $p_{snd}$ | probability of sending a message in a given block |
| $d_{snd}$ | time to construct a message and put in send queue |
| $d_{rcv}$ | time to remove message from receive queue and interpret it |
| $d_{int}$ | time spent saving/restoring registers and so on, before/after computing reply |
| $d_{msg}$ | time to deliver a message (assumed constant) |
| $d_{bsynch}$ | time to perform barrier synchronization |
| $d_{bsnd}$ | time for send-transfer-receive when doing bulk-synchronous transfer $d_{bsnd} \doteq d_{snd} + \max(d_{int}, d_{nmsg}) + d_{rcv}$ |
| | *variables* |
| $S_{pre}(b,p)$ | time in block $b$ executed by processor $p$ *before* point where message may be sent |
| $S_{post}(b,p)$ | time in block $b$ executed by processor $p$ *after* point where message may be sent |
| $S_{rep}(b,r,p)$ | time to compute the reply to the $r$'th message received by processor $p$ that was sent during block $b$ |
| $\delta_s(b,p)$ | 0 or 1 according to whether processor $p$ sent a message during iteration $b$. |
| $\delta_r(b,p)$ | 0 or 1 according to whether processor $p$ received a message during iteration $b$. |
| $\delta_{self}(b,p)$ | 0 or 1 according to whether processor $p$ sent a message during iteration $b$ whose target was itself |
| $\delta_s^*(b)$ | 0 or 1 according to whether any processor received a message during iteration $b$, $\delta_s^* \doteq (1 - \prod_{p=1}^{P}(1 - \delta_s))$ |
| $targ(b,p)$ | destination (if any) of message sent by processor $p$ during block $b$ |
| $n_{rcv}(b,p)$ | number of messages received by processor $p$ during iteration $b$ |
| $\Delta_p$ | delay between when a message is received by processor $p$ and when it is processed, due to processing of previously received messages |

5.9) can be written

$$
\begin{aligned}
b^{spmd}(b,p) \quad \doteq \quad & S_{pre} + (1-\delta_s)(S_{post} + \sum_{r \in R}(d_{rcv} + d_{int} + S_{rep}(r) + d_{snd} + d_{int})) \\
& + \delta_s \max[(S_{post} + \sum_{r \in R}(d_{rcv} + d_{int} + S_{rep}(r) + d_{snd} + d_{int})), \\
& (1-\delta_{self})(d_{msg} + \Delta_{targ} + S_{rep}(targ) + d_{snd} + d_{msg} + d_{rcv})]. \qquad (5.13)
\end{aligned}
$$

where the summation is over the set $R(p)$ of all message replies performed by the processor in the block. The time for a single block under the SFMD model is

$$
\begin{aligned}
b^{sfmd}(b) \quad \doteq \quad & \max_p [S_{pre} + d_{snd} + \max(S_{post} + d_{int}, \delta_{self} d_{nmsg})] + d_{bsynch} \\
& + \delta_s^* \max_p \max_{k=1}^{n_{rcv}} \sum_{r=1}^{k} (d_{rcv} + S_{rep}(r) + d_{snd} + (n_{rcv} - k + 1) d_{nmsg}) \\
& + d_{bsynch} + d_{rcv}. \qquad (5.14)
\end{aligned}
$$

The $\max_k$ following $\delta_s^*$ is over the time to compute the first $k$ replies and then deliver the $k$'th and following replies sequentially. Note that assuming the network delivers the messages sequentially is a worst case scenario for SFMD.

### 5.7.2 Analysis

I give a rough analysis of the permutation case. The *random* case has two natural execution models, one where each processor replies to all the messages it has received before it synchronizes with the other processors, and one where each processor replies only to a single message (or some fixed number of messages) before synchronizing. Under the first model, the random case has the same qualitative behavior as the permutation case, with the possible multiple replies in the random case acting like a larger, more variable, reply in the permutation case. I have not yet addressed the second case. For either case, strong assumptions are needed about the network's performance delivering multiple messages for reasonable modeling.

In general, the constants $d_{snd}$, $d_{rcv}$ and $d_{int}$ will be small, and I ignore them for purposes of analysis. Assume that the network has sufficient bandwidth that $d_{msg} = d_{nmsg}$. The overhead of polling may be quite small, especially with hardware assistance from the autonomous network (it could be just checking a register periodically). However, even a single-cycle poll, if done at every instruction, doubles the time. So polling would be done at some slower frequency, which then has the effect of delaying when a message is processed. I will ignore polling in the analysis (which penalizes the SFMD case) and give some simulation results.

Take all $S_{pre}$, $S_{post}$, and $S_{rep}$ for all processors to be mutually independent. This is reasonable for the scenario where execution of the code after the message depends on the message result, and the time to compute a reply, e.g. by indexing into a data structure, is independent of the content of the reply. Let $\mu^{pre} = \langle S_{pre} \rangle$, $\mu^{post} = \langle S_{post} \rangle$, and $\mu^{blk} = \langle S_{pre} + S_{post} \rangle = \mu^{pre} + \mu^{post}$. Assume that message destinations are equiprobable and independent, so $\langle \delta_s \rangle = \langle \delta_r \rangle \doteq q$, and $\langle \delta_{self} \rangle = q/P$.

Even for the *permutation* case, the delay $\Delta_{targ}$ for SPMD may be nonzero. The reason is that the threads on the separate processors rapidly become desynchronized [AGWFH94, WHAG+92], so that when the message is received, the target processor may be processing a message from another processor sent during a previous block. Processing the new message is delayed until all previously received messages have been processed. To estimate $\langle \Delta_{targ} \rangle$, note that the probability that a message is being processed at a given point is the average time spent processing messages, $q\mu^{reply}$, divided by the average total time per block, $\mu^{blk} + q\mu^{reply}$. The expected time to finish processing the message is $\mu^{reply}/2$. We get a lower bound by ignoring the case where there is one or more previous messages to be processed after the current one and before the just-received one:

$$\langle \Delta_{targ} \rangle \approx \frac{q\mu^{reply}}{\mu^{blk} + q\mu^{reply}} \frac{\mu^{reply}}{2}.$$

Taking expectations over *max* operators is difficult; instead I examine the equations in regimes where one term sufficiently dominates the other that one can substitute its value for the appropriate *max*. Denote this *dominates* relation by "$\gg$".

Under these assumptions, for the SPMD case, I get

$$\langle b^{spmd}\rangle \approx \begin{cases} \mu^{blk} + q \cdot \mu^{reply} & \text{if } C_{\gg} \\ \mu^{blk} + q(\frac{2P-1}{P}(\mu^{reply} + 2d_{msg}) - \mu^{post}) & \\ \quad -q^2\mu^{reply}(1 - \frac{P-1}{2P}\frac{\mu^{reply}}{\mu^{blk}+q\mu^{reply}}) & \text{if } C_{\ll} \end{cases} \tag{5.15}$$

where $C_{\ll}$denotes the condition

$$S_{post} + \delta_r S_{rep} \ll (1 - \delta_{self})(S_{rep}(targ) + \Delta_{targ} + 2d_{msg})$$

and $C_{\gg}$ denotes the reverse. The total time for all $N$ iterations is $\max_p \sum_1^N b^{spmd}$, and, as in section 5.6.1, the central limit theorem implies

$$\langle t^{spmd}\rangle \approx N\langle b^{spmd}\rangle + \sqrt{N}\sigma_{b^{spmd}} Z_{(P)} \tag{5.16}$$

for large $N$, where $Z_{(P)}$ is distributed as the maximum of $P$ standard normal variates.

For the SFMD case, and the permutation communication pattern, we have

$$\langle b^{sfmd}\rangle = \begin{cases} \mu^{pre}_{(P)} + (1 - \delta_{self})d_{msg} + \rho(d_{msg} + \mu^{reply}_{(P)}) & \text{if } D_{\gg} \\ \mu^{blk}_{(P)} + \rho(d_{msg} + \mu^{reply}_{(P)}) \text{if } D_{\ll} \end{cases} \tag{5.17}$$

where $\rho = \rho_P(q) \doteq \langle\delta^s_*\rangle = (1 - (1-q)^P)$, $D_{\gg}$ denotes the condition

$$(1 - \delta_{self})d_{msg} \gg S_{post}$$

and $D_{\ll}$ its reverse.

We have $\langle t^{sfmd}\rangle = N\langle b^{sfmd}\rangle$, so the speedup is $\frac{\langle t^{spmd}\rangle}{\langle t^{sfmd}\rangle} \approx \frac{\langle b^{spmd}\rangle}{\langle b^{sfmd}\rangle}$ asymptotically in $N$.

As $C_{\gg}$implies $D_{\ll}$, there are three regimes. In regime **R1**, defined by $C_{\gg}$, the time to complete the block dominates the times to compute and send the reply. In this case, for large $N$,

$$\frac{\langle t^{spmd}\rangle}{\langle t^{sfmd}\rangle} \approx \frac{\langle b^{spmd}\rangle}{\langle b^{sfmd}\rangle} = \frac{\mu^{blk} + q\mu^{reply}}{\mu^{blk}_{(P)} + \rho_P(q)(d_{msg} + \mu^{reply}_{(P)})}.$$

In regime **R2**, defined by $C_{\ll}$and $D_{\ll}$, the time to compute the reply dominates, as might be the case for small computations requires small pieces of information that require extensive indexing to find. In this regime, for large $N$,

$$\frac{\langle t^{spmd}\rangle}{\langle t^{sfmd}\rangle} \approx \frac{\mu^{blk} + q(\frac{2P-1}{P}\mu^{reply} + 2d_{msg} - \mu^{post}) - q^2\mu^{reply}(1 - \frac{P-1}{2P}\frac{\mu^{reply}}{\mu^{blk}+q\mu^{reply}})}{\mu^{blk}_{(P)} + \rho_P(q)(d_{msg} + \mu^{reply}_{(P)})}.$$

Regime **R3**, is defined by large message times, $D_{\gg}$, implying $C_{\ll}$. This might correspond to the situation where large latency inter-chip messages are sent. One has

$$\frac{\langle t^{spmd}\rangle}{\langle t^{sfmd}\rangle} \approx \frac{\mu^{blk} + q(\frac{2P-1}{P}\mu^{reply} + 2d_{msg} - \mu^{post}) - q^2\mu^{reply}(1 - \frac{P-1}{2P}\frac{\mu^{reply}}{\mu^{blk}+q\mu^{reply}})}{\mu^{pre}_{(P)} + d_{msg} + \rho_P(q)(d_{msg} + \mu^{reply}_{(P)})}.$$

Lower bounds for the speedup in these regimes, asymptotic in the number of iterations, $N$, are given by the ratio $\langle b^{spmd}\rangle/\langle b^{sfmd}\rangle$ of (5.15) and (5.17) in the three regimes. Comparison of these bounds with simulations are given in figure 5.11.

Both analysis and simulation of the lower bounds show that, as a function of $q$, the qualitative form of the speedup has a single minimum at about $1/P$ and reaches a maximum at either $q = 0$ or $q$ near 1. This is intuitive, as SFMD is most penalized for bulk-synchronous communication when the expected number of messages per block is 1 : $P$ processors each with a $1/P$ probability of sending a message in a given block. Conversely, SFMD is least penalized when there is no communication or when all processors must compute a reply.

I wish to examine these equations as functions of the message send probability, $q$. Let $\phi(p) \doteq \langle t^{spmd}\rangle/\langle t^{sfmd}\rangle$. Using $\langle \max_M X\rangle = M\cdot\int xP(x)^{M-1}dP(x) \leq M\cdot\int xdP(x) = M\langle X\rangle$ for any random variable $X$ with cdf $P(x)$, it is routine to show that $\phi(q)$ behaves as illustrated in figure 5.10. In particular, if we construct $\phi_0(q)$ by replacing the $\rho_P(q)$ term in the denominator by $P\cdot p$, and construct $\phi_1(q)$ by replacing the same term by one[9], then $\phi$ is bounded below by $\phi_0$ and $\phi_1$ for all $q \in [0, 1]$, and is well approximated by them near 0 and 1, respectively. In particular, for the values of $P$ of interest, $P \geq 8$, and for $q > 0.5$, $\rho_P(q)$ is essentially 1. Intuitively, the minimum of $\phi(q)$ should be near $q = 1/P$, as this corresponds to an expected one message per block, which will incur the maximum penalty

[9]This comes from expansion in a Taylor series about $q = 0$ and $q = 1$, respectively.

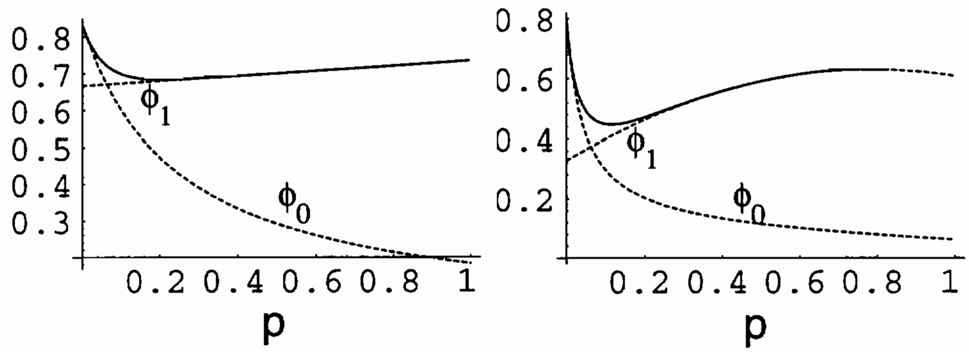


Figure 5.10: $\phi(p)$ in two regimes: the left panel illustrates regime **R1**, while the right panel illustrates regime **R2**. Regime **R3** is like **R2** except that the local maximum near 1 is not reached for $p < 1$.

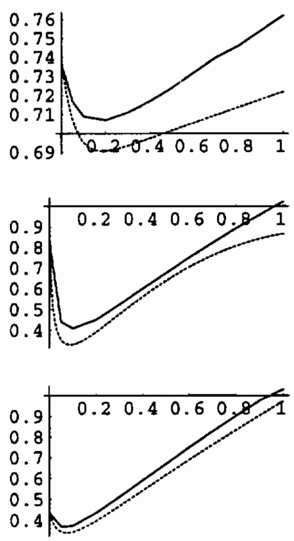


Figure 5.11: Simulation results compared to analytic lower bounds (dotted lines) in three regimes. The speedup ratio $\langle t^{spmd} \rangle / \langle t^{sfmd} \rangle$ is plotted against the message send probability, $q$. In the top panel, time in the sending block dominates ($\mu^{pre} = 100, \mu^{post} = 400, \mu^{reply} = 30, d_{msg} = 10$), in the middle panel, time to compute the reply dominates ($\mu^{pre} = 100, \mu^{post} = 30, \mu^{reply} = 400, d_{msg} = 10$), and in the bottom panel, message transfer time dominates ($\mu^{pre} = 100, \mu^{post} = 30, \mu^{reply} = 30, d_{msg} = 200$). In all cases, $P = 16$ and $N = 100$. $S_{pre}$ was a constant 100, to simulate a fixed block of pre-message code. $S_{post}$ and $S_{rep}$ were uniformly distributed on either $[10, 50]$ or $[200, 600]$.

for SFMD operation at the least penalty to SPMD operation. In fact, $\phi_0(1/P) = \phi_1(1/P)$ is a lower bound for $\phi(q)$ for $q \in [0, 1]$. We see that "small" $q$ must be very close to zero if SFMD operation is to be not too unfavorable compared to SPMD; essentially, this is the "no communication" case. On the other hand, for "large" $q \in [1/2, 1]$ the issue is not so clear. Suppose, henceforth, that $q \geq 1/2$. For each of the three regimes, ignoring small terms as defined by the dominance relation for that regime, $\phi(1/2) = \min_{q\in[1/2,1]} \phi(q)$, and the condition $\phi(1/2) \geq 1/2$ corresponds to

$$\begin{aligned} &\mu^{blk} \geq \tfrac{1}{2}\mu^{blk}_{(P)} && \text{in } \mathbf{R1} \\ &\mu^{blk} + \tfrac{3}{4}\mu^{reply} \geq \tfrac{1}{2}(\mu^{blk}_{(P)} + \mu^{reply}_{(P)}) && \text{in } \mathbf{R2} \\ &\mu^{pre} + \tfrac{3}{4}\mu^{reply} \geq \tfrac{1}{2}(\mu^{pre}_{(P)} + \mu^{reply}_{(P)}) && \text{in } \mathbf{R3} \end{aligned} \tag{5.18}$$

The distribution-free upper bound [Dav70]

$$\langle \max_P X \rangle \leq \mu_X + \frac{P-1}{\sqrt{2P-1}}\sigma_X,$$

where $\mu_X$ and $\sigma_X$ are the mean and standard deviation is strict, in that there is a distribution for which the inequality is an equality. It is, however, a substantial overestimate for many distributions. Let $\nu(P) \doteq (P-1)/\sqrt{2P-1} \approx \sqrt{P/2}$. If we substitute the upper bound in the conditions 5.18 we get stronger (perhaps overly strong) conditions

$$\begin{aligned} &\mu^{blk} \geq \nu(P)\sigma^{blk} && \text{in } \mathbf{R1} \\ &\mu^{blk} + \tfrac{1}{2}\mu^{reply} \geq \nu(P)(\sigma^{blk} + \sigma^{reply}) && \text{in } \mathbf{R2} \\ &\mu^{pre} + \tfrac{1}{2}\mu^{reply} \geq \nu(P)(\sigma^{pre} + \sigma^{reply}) && \text{in } \mathbf{R3}. \end{aligned} \tag{5.19}$$

For $P \leq 128$, $\nu(P) \leq 8$, and the above conditions hold in case

$$(\mu^{pre} \geq 8\sigma^{pre}) \vee (\mu^{post} \geq 8\sigma^{post}) \vee (\mu^{reply} \geq 16\sigma^{reply}). \tag{5.20}$$

So if the message probability is not too low ($q \geq 1/2$) and the variances of the *pre*, *post* and *post* times are not too great compared to their means, then $\phi(q) > 1/2$, *i.e.*, the slowdown of SFMD computation compared to SPMD computation is less than twofold. With a less than two-fold slowdown, a 128 PE SFMD chip would have greater raw performance a 16

PE SPMD chip whose (more complex) processors were each 4 times faster, even if the 16 PE chip were not memory bandwidth limited. In particular, much ILV computation falls into the "high message send probability, permutation communication pattern" regime.

Of course, this is a much simplified model, and very rough bounds, but I believe it gives us some intuition. First, it suggests the importance of reducing variance in search and message passing. In this regard, note that I have not yet looked at "averaging out" some of the variance by combining loop bodies so that multiple blocks ($S_{pre}$ and $S_{post}$) are done on a given PE before the barrier synchronization to begin the reply phase, and that multiple replies ($S_{rep}$) are computed on a given PE before the synchronization to end the reply phase. Second, it allows us to conclude that there are reasonable regimes involving substantial communication where SFMD architectures may be competitive with SPMD ones. Even if SFMD only gives the same performance as SPMD in some task, this extends the range of tasks an SFMD chip can usefully perform, making the chip more economically viable. From Amdahl's law, simply being (approximately) equivalent on some parts of the computation is important for allowing better performance on other parts of the task to translate into better overall performance on the entire task.

## 5.8 Miscellaneous Design Issues

One point about SFMD execution, compared to SIMD has not yet been made. There are really two sources of parallelism in a typical SIMD implementation. In addition to parallelism from SIMD execution, there is parallelism between the host and the SIMD array. A common, and important, use overlaps testing and incrementing the outer loop index variable on the sequencer while the SIMD array computes the inner loop [ANSC94]. A superscalar processor normally overlaps these computations as they are independent instruction sequences. In an SFMD machine, many loops may still be data-independent, and executed in SIMD mode, and one expects that it would be worth including shared functionality on each chip for loop variable computation on the sequencer. In SFMD, when loops are handled on a per-PE basis using a simple, non-superscalar PE, overlapping is not possible. If the loop count is data-dependent, but known before the loop is entered, a DSP "zero overhead loop" mechanism would suffice, as it would for loops that are exited by a "break" construct, but have a maximum iteration count. A reasonable example of the latter would be a relaxation loop, but loops that search tree structures can be constructed this way by giving an unreachably large iteration count. A final possibility is to include a small amount of hardware for this specific form of superscalar execution. The hardware requirements for this, especially if the overlapping is encoded in the instructions through a Very Long Instruction Word (VLIW) approach, may be minimal.

## 5.9 Related Work

### 5.9.1 Instruction Caching

An SFMD architecture includes per-PE instruction memories for small, "inner loop" bodies of code. There are a couple of related ideas on using such instruction caches.

Rockoff [Roc93] demonstrates that for an SIMD system, distributing instructions from off-chip incurs too large a performance penalty due to lower off-chip speeds and bandwidth, and (for post-1993 VLSI processes) is generally inferior to the use of an on-chip instruction cache.

Manning and Meyers [MM93] describe simulations of an instruction caching idea whose goal is massive asynchronous SPMD processing without the cost of separate instruction memories. Unlike SFMD, there is no notion of "virtual synchrony". The idea is to replace separate instruction memories by adding smaller *instruction cache (I-cache)* to each PE of a massively parallel $O(1000)$ system. A data-parallel program is divided into "locales", and I-cache requests locales as necessary on a token-ring bus. All I-caches snoop the bus for other caches requesting the same locale. If the desired locale is seen on the bus, it need not be requested. They use a Markov model of program movement between locales derived from real programs in their analysis and presents results on PE utilization that are favorable for massive systems where the number of PEs is much larger than the number of locales. For smaller number of PEs, this is unlikely to hold as simultaneous request for a locale unlikely [AGWFH94, WHAG+92]. Lundstrom [Lun87] presents much the same idea as part of the design for the never-completed Burroughs "Flow Model Processor".

### 5.9.2 Other SIMD-MIMD hybrids

There is a plethora of ideas for hybrid architectures somewhere between SIMD and MIMD.

Contemporary processors aimed at multimedia applications are beginning to include some SIMD-like functionality (see, for example [Lee95]). The idea is to partition the 32- or 64-bit ALU so that it can operate on, say, 1 64 bit, or 2 32-bit, or 4 16-bit, or 8 8-bit quantities. What is required is to disable carries over the boundaries of the smaller data. Operations on data smaller than 32 bits use fixed point arithmetic. Partitioning gives a very restricted form of SIMD, without either local addressing or local conditional execution, and provides an economical extension of uniprocessor designs to accommodate a moderate amount of very regular parallel execution.

There are a number of vector processor hybrids, an example of which is [WAK+96], targeted at very regular sensory processing applications such as neural network processing of speech. These add fixed point vector processing hardware, suitable for very regular kinds of processing such as non-sparse matrix multiplication, to a standard microprocessor.

Several researchers have looked at emulating MIMD execution on massively parallel SIMD machines, to increase the range of problems they can address [AGWFH94, WHAG+92, SW95]. The emulation overhead generally negates any advantage from the massive parallelism.

Conversely, and more successfully, SIMD execution can be emulated on an array of MIMD processors by coupling SPMD execution with barrier synchronization. The best known example is the CM-5 [Cor92], although the Cray T3D also has hardware support for fast barriers. The barrier synchronization in the CM-5 is "soft", in the sense that a PE indicates a desire to participate in a barrier, and may then perform other work until notified asynchronously that the other PEs participating in the barrier are ready. The barrier thus acts to enforce global ordering, but the penalties for asynchronous notification (interrupts or polling) are still incurred. This is reasonable for a system with potentially thousands of processors, like the CM-5, where waiting for all PEs to reach the barrier may take a long time, and where individual PEs may execute multiple processes, and switch between them while waiting. It differs from the SFMD idea, targeted at comparatively small numbers of simple processors each running a single thread, where barriers are "hard". With the hard barriers of SFMD, no asynchronous overhead is incurred in communication while operating in SIMD mode after the barrier.

A number of experimental processors have both SIMD and MIMD/SPMD modes, with the capability of quickly switching between them. This necessarily involves barrier synchronization of some sort at the transition from MIMD to SIMD.

PASM [NSD93] is a research machine designed to allow dynamic repartitioning of the PEs into number of independently operating sub-machines. Each of these sub-machines is a mixed SIMD/MIMD hybrid, allowing fast switching between SIMD and MIMD modes. Much PASM research has been targeted at the cost/performance of switching between the two modes [BKS91, WSA+94] and has led to an understanding of the efficiency of the barrier MIMD mode of computations [DZO92, BCJ90]. Triton [PWTH93] is a similar SIMD/MIMD hybrid. EXECUBE [Kog94] is a similar hybrid, targeted at very massive systems, and designed for maximum performance per transistor, rather than per chip. As the goals of these machines is generality, no restrictions such as disallowing communication during SPMD mode are enforced.

The OPSILA computer [DBAG88, AB86] is a vector processor with SPMD extensions. The basic vector processing model has neither local conditional execution nor local addressing. These are enabled by the addition of small local instruction memories and an "SPMD mode". SPMD mode is used for calculations requiring local addressing, such as histogram computation, or local conditional execution, such as the body of a list traversal procedure. The available literature shows no use of loops within SPMD mode. Since OPSILA is fundamentally a vector machine, interprocessor communication is only via bulk vector operations such as permutations or scatter-gathers. As such, IPC within SPMD mode is disallowed. The programming language for OPSILA distinguishes SIMD and SPMD execution and variables used by SIMD and SPMD modes. OPSILA is thus essentially an SFMD architecture, although not emphasizing that programming semantics are SIMD and not concerning itself with VLSI and chip microarchitecture issues.

### 5.9.3 Vision-specific Designs

Jonker [Jon93, JKK95] has proposed a system targeted at low and intermediate level vision using hardware-supported "bucket queues" as the representation for ILV data, corresponding to use of arrays in low-level vision processing. The architecture is designed for pipelined processing, and is related to SFMD in that separate processors operate autonomously, controlled by individual state-machines downloaded into a reconfigurable logic array.

Mention must be made of the Image Understanding Architecture (IUA), an ongoing research project targeted at low through high level vision processing [Wee93, Wee94, WLH+89]. The IUA has separate hardware for the three stages of low, intermediate, and high level vision. The low level hardware is SIMD, while the intermediate level is SPMD running on a collection of commercial DSPs with a high bandwidth interconnect. There is no special relationship, other than connectivity, between the SIMD and SPMD hardware or operation, so there is no notion of SFMD-like function. Weems provides a thorough discussion of the processing needs of the various levels in [Wee91].

## 5.10 Future Work

Of course, experience with SFMD on real machines, for real applications is what is needed. Prior to building an SFMD machine, though, there are a number of useful analyses to do. It would be useful to examine the FIFO-based implementation of SFMD further, both analytically and through simulation. Preliminary simulations suggest the overhead of using a FIFO due to stalls on a "FIFO full" condition is not large. For understanding the coding implications of a given size FIFO, one would like some idea of an "equivalent buffer size": for a given IM size in the non-FIFO implementation of SFMD, what size FIFO gives a performance degradation of at most, say 5%. This, of course, will depend on the distribution of execution times for the code. Elucidating this dependence would be of interest, and having an idea of equivalent buffer size would be useful in designing code for a specific application.

There are a number of reasons why it might be useful for a single SFMD PE to emulate multiple virtual PEs. A single PE switching between different virtual PEs might better tolerate memory latency, using multi-banked sense-amp-caches with different banks for the different virtual PEs. In my comparison with SPMD execution, each loop iteration caused a synchronization for replying to, and then receiving, messages. This breaks the loop body into three phases, "pre-reply", "reply" and "receive", separated by the two barrier synchronizations. Emulating multiple virtual PEs would provide a natural mechanism for executing pre-reply phases for multiple loop iterations, then a single barrier synchronization, then the reply phases for the multiple loop iterations, a single barrier, and then multiple receives. Executing multiple loop operations in a single phase would have the effect of reducing the variance in the times of the phases, and so improving the performance of SFMD relative to SPMD.

For the non-FIFO version of SFMD, one can pipeline results between PEs executing different functions. This would again potentially increase the range of applicability of an SFMD chip, but it is unclear how to model this in the programming language, or whether any special hardware support is needed.

## 5.11 Summary

In this chapter I introduced and analyzed the SFMD class of computer architectures. SFMD extends SIMD by allowing data-dependent control flow for the individual PEs. This extension to data-dependent execution is increasingly important as conventional architectures are including low-parallelism SIMD vector processing in their design, targeted at multimedia processing (for example, [Lee95]). "Pure" SIMD and vector designs will thus have a harder time differentiating themselves from mainstream processors. As seen in chapter 3, support for algorithms having small computational kernels that exhibit irregular, data-dependent control flow is essential in extending one of the traditional range of applications of SIMD designs, low level image processing, to intermediate level and contextual processing. By extending SIMD designs into "neighboring" task domains, SFMD makes it more likely that a given chip can do more of the processing necessary for a task without needing to have another chip do part of the work, possibly incurring the cost of data transfer.

SFMD adds little or no expense to SIMD in terms of programming complexity, tools, or environment. It is relatively inexpensive in terms of area, and fits well into designs that put all models or task parameters on-chip, avoiding memory bandwidth limitations, and reducin power dissipation, packaging, and system integration costs. An implementation of SFMD may require reducing the available on-chip memory by perhaps 5%, while over a wide range of tasks, SFMD provides a 1.5 to 2-fold improvement in performance compared to SIMD. Compared to a multiprocessor running SPMD code, there is a performance degradation on code with infrequent messaging and on code having a large variance in its execution time. However, this performance degradation may be mitigated by an SPMD design having more processors, or possibly by virtualization techniques to reduce variance. Even in cases where an SFMD chip is only comparable in performance to an SPMD chip, it still extends the range of applicability of the SFMD chip and increases its economic viability.

# Chapter 6

# Conclusion

## 6.1 What Has Been Done

My interest in this work has been in the cost-effective parallel implementation of contextual processing. The framework used was to view *models* as encapsulated pieces of contextual knowledge that are applied "top down" in a *model matching* process. A large recursive model for ordered input provides context for the interpretation of sequences of its components through conditional probabilities of components given previously interpreted components. Smaller, individual object models, provide context for the interpretation of their components during the process of matching the model to data. This context can be in the form of conditional probability statements, hard geometric constraints, or "soft" geometric constraints in the form of deformation energy. The "geometric context" can be used to direct search. The key point is that use of context leads to irregular, data-dependent control flow.

For ordered input such as text and speech, we saw that use of a large, recursive model based on $n$-gram probabilities could be useful for correcting the interpretations of sequences of components, and for dealing with novel words. The HOVS algorithm for making use of this probabilistic contextual information has a SIMD implementation that provides very fast processing for large vocabularies. However, for large vocabulary tasks where the goal is to identify the correct (non-novel) word from the vocabulary, I concluded that effective use of parallel model matching was unlikely, due to the effectiveness of search mechanisms that make use of the ordering.

Conversely, for unordered input such as vision, search seems hard to apply. In this case, identifying objects by matching structured models seems a reasonable approach. Doing so, however, leads to algorithms with pronounced data-dependent irregularity that is inefficient for SIMD-style computation. Examination of a variety of these algorithms, and related algorithms for feature grouping, led us to a number of "stylized facts" that suggested certain architectural requirements for the support of these algorithms. The architecture should be a hybrid, with a fast uniprocessor coupled to a parallel array. The PEs of the parallel array should have large memories, and should communicate by an asynchronous autonomous network to allow overlapping computation with communication. Finally, the PEs should support algorithms having small computational kernels that exhibit irregular, data-dependent control flow.

Next I examined VLSI and microarchitectural trends to quantify the tradeoffs in designing such a parallel array. The essential fact from current VLSI trends is that the growth in the number of transistors that can be put on a chip will substantially outpace growth in the off-chip bandwidth. I examined a highly parallelizable model-matching task being executed by such an on-chip parallel array. When off-chip bandwidth was assumed equal between a uniprocessor and a parallel array, off-chip bandwidth limitations limited parallel speedup to a modest amount unless the subtasks evaluated in parallel were very compute-intensive. This suggested that a design using off-chip memory would have limited parallelism and would be hard to differentiate from mainstream processors, making it difficult to justify its development. However, current interest in, and development of, so-called "embedded DRAM" processes suggest the feasibility of a design using on-chip rather than external memory. Such a design could have a high degree of parallelism, sidestepping limitations on parallel speedup due to off-chip bandwidth constraints.

Examination of current microarchitectural trends delineated the techniques used in superscalar designs to improve performance and, in particular, to tolerate latency due to off-chip memory bandwidth limitations. We saw that these techniques were expensive in terms of area, power, and complexity, may have scaling problems due to quadratic growth and non-locality in required interconnect, and were unlikely to function well for highly data-dependent execution with unpredictable branching and poor locality of reference. Designs based on these techniques, while tolerating memory latency to some degree, will also suffer from off-chip bandwidth limitations. This suggests that a highly parallel design, with many simple processors per chip using on-chip memory, may have a niche. Such a design could be architected as a vector or SIMD processor. In such a design, sharing instruction processing hardware allows processors to be much simpler; simplicity being now much more important in terms of reduced area, due to the larger number of processors. However, conventional architectures that share instruction processing hardware (SIMD or vector architectures) perform poorly on the computational kernels exhibiting irregular, data-dependent control flow that we have seen are needed for model matching and feature grouping algorithms.

I then introduced the SFMD architecture class, adapting SIMD execution and allowing the sharing of instruction hardware while significantly outperforming SIMD on small computational kernels with irregular data-dependent control flow. SFMD has the same semantics as SIMD, and can be implemented as an extension to an existing SIMD architecture. SFMD has the low programming and debugging costs of SIMD, and requires little or no change in an existing SIMD programming environment. The architectural changes required have a moderate silicon cost, and a low design cost; in particular, as SFMD concerns itself solely with instruction delivery, little redesign of an existing simple SIMD PE is likely to be needed.

While the raw performance of SFMD improves on SIMD by a factor of 1.5 to 2 on a variety of tasks with data-dependent control flow, it, in turn, is outperformed by an SPMD architecture on tasks with sparse communication and highly varying computation and communication times. I discuss below improving the relative performance by reducing variance in the communication and computation times. When communication times are not too sparse, the performance gap between SFMD and SPMD may be counteracted by the ability of SFMD to put more processors on a chip. In any case, SFMD outperforms

SIMD on these kinds of tasks, and extends the range of an existing SIMD design into the realm of tasks with irregular control flow.

## 6.2 What Remains To Be Done

### 6.2.1 Practicalities

Of course, I would like to examine the performance of SFMD on real examples, or at least get better time distribution estimates for my analyses. Unfortunately, there is a problem. I have contacted a number of vision researchers, in order to obtain realistic uses of structural model matching. At present, there really are no realistic examples: structural models, while of great research interest, are too computationally expensive to use in practice. There is a "chicken and egg" problem, between having an application for which to develop a "context engine" and developing an application which requires a (not yet existent) context engine for useful performance. With increasing performance by mainstream processors, I can hope this issue becomes resolved.

What is particularly needed is a "killer" application to drive development of the engine. MPEG-4, which may use a model-based representation to achieve extremely high compression rates is a possibility, as are other forms of video and multimedia processing. To justify development of a context engine chip, the application probably must require a low-cost, perhaps portable, solution. For example, interpretation of satellite images is an unlikely candidate, since this can probably be done cost-effectively by a large conventional parallel machine. (An system for interpreting satellite images might be able to make use of a context engine if it existed, but is unlikely to drive its development, as conventional solutions are probably adequate.)

### 6.2.2 The Asynchronous Autonomous Network

I have been treating the *asynchronous autonomous network* as a black box. It is clear one needs to develop a model of the network that can speak to issues of messaging latencies, throughput, silicon area and design cost. Generally, I am thinking in terms of each PE having a small DMA engine and some memory for storing messages, which may or may not be separate from the PE's local memory. As a minimal implementation, one might have a bus, with one PE's DMA engine designated as "sender" in a round-robin schedule and all other PE's DMA engines snooping the bus to determine if the message is for that PE. This may be inadequate for a 128 PE chip; on the other hand, a network capable of full permutation routing may not be needed unless it is felt necessary for the SIMD portion of the applications.

The use of the network in SFMD mode has some special features. As messages are not "seen" by the PE until the barrier is reached, regardless of when they were sent, the network message latency can perhaps be larger than otherwise. Sending messages in SFMD mode may also require less "instantaneous" bandwidth needed than normal SIMD execution. If the code sending the messages contains loops, substantial conditional execution, or is substantially affected by memory and functional unit latencies, the different PEs will rapidly become desynchronized [AGWFH94] and so the injection of messages into the network will be spread out over time. (Note: this would argue against a fixed round-robin schedule in the bus-snooping implementation suggested above, and for a more dynamic arbitration scheme.)

After the barrier, the set of received messages can be processed in SFMD mode; sending, receiving and processing multiple messages per barrier will improve performance by reducing variance. All messages must be sent (and received) before the barrier, and, subject to that constraint, the order and time of their receipt is irrelevant to their processing. Thus, when multiple messages are sent per barrier, there is no penalty (other than a larger message memory) in having each PE's DMA engine store pending "send" messages until arbitration allows that PE to send, and then inject all the pending messages into the network. This amortizes the arbitration cost over the set of messages sent at once, and should improve the effective bandwidth.

Consequently, there are aspects of SFMD computation suggesting that, compared to SIMD, a lower performance, less expensive network may be adequate, when combined with a technique of variance reduction by performing multiple (partial) loops per barrier (see below). Studying this in more detail would be of interest.

### 6.2.3 Variance Reduction and Virtualization

We saw in chapter 5 that the culprit in the poor performance of SFMD compared to SPMD is variance in execution and messaging times. Reducing variance by doing multiple (say, $n$) (partial) loop bodies barrier should improve performance by reducing variance by a factor of $\sqrt{n}$. Consider a loop on a single PE, as in the comparison of SFMD with SPMD:

```
for i=1:N
  pre(i)
  barrier
  reply(i)
  barrier
endfor
```

where previously received message are processed and new messages are sent during `pre(i)`, and messages are replied to during `reply(i)`. SPMD outperforms SFMD due to sparseness of communication during `pre` and due to `pre` and `reply` execution times that have large standard deviations compared to their means. Consider rewriting the above loop as

```
for i=1:N
  pre(i)
endfor
barrier
for i=1:N
  reply(i)
endfor
barrier
```

where keeping track of which messages are for what loop iterations is taking place behind the scene. Of course, this transformation makes assumptions about non-dependence between loop iterations. Rewriting loops in this way makes communication less sparse, and reduces the "mean / standard deviation" in execution times by a factor of $\sqrt{N}$ (assuming statistical independence in execution times). As we saw, rewriting loops this way may also provide benefits in allowing a simpler, less expensive interconnection network.

One would like to relieve the programmer of the complexity of rewriting loops this way, either by compiler optimizations or by presenting a view of multiple virtual PEs (VPEs) for each hardware PE. Compiler optimizations for variance reduction perform transformations of the above type automatically, based on analysis of dependence between loop iterations, and are well understood [Wol96].

With virtualization, a single PE would have a loop of the form

```
for vpe = 1:v
  for i=1:N/v
    pre(vpe,i)
    barrier
    reply(vpe, i)
    barrier
  endfor
endfor
```

Assuming non-dependence between processing on different VPEs, this is transformed into

```
for i=1:N/v
  for vpe = 1:v
    pre(vpe,i)
  endfor
  barrier
  for vpe = 1:v
    reply(vpe, i)
  endfor
  barrier
endfor
```

It is clear these are the same transformations; the advantage of a virtualization perspective is in simplifying the programmer's view of things, allowing him or her to write code for some convenient number of VPEs without worrying about the exact number of actual PEs on the chip. Hardware support for virtualization could also be used to hide memory latency time, by fast switching between VPEs, and thus raise the IPC rate of the PEs. However, the cost of support for such virtualization and fast switching is unclear, especially in the presence of the highly pipelined PEs that will be necessary for competitive clock speeds.

### 6.2.4 Some Speculations on Cortical Models

Large-scale cortical models may or may not ever be the appropriate approach to applying contextual knowledge in a computer, but it is certainly plausible, as the cortex is the seat of much of the processing I am interested in emulating. Cortical models are characterized by extremely large numbers of processors (neurons), with sparse connectivity and sparse activation. Bailey and Hammerstrom [BH88, Bai93] have shown that multiplexed hierarchical interconnect is a reasonable approach to implementing sparse connectivity. Sparse activation suggests, and probably demands, multiplexing of multiple neurons onto a single processor, but this becomes problematic due to load balancing issues. Of course, if processing time varies among neurons, SFMD is a natural candidate architecture. However, even if the processing time is the same, SFMD may have a role to play. At a higher level of granularity, something like the *neuronal group* of Edelman [Ede86], there are likely to be reoccuring patterns of (sparse) activation. These begin to look like the structured models I have been considering, and for which SFMD was designed.

## 6.3 Final Words

Contextual processing involves irregular computation. The SFMD class of architectures provides a relatively cheap extension of SIMD processing for this purpose, suitable for on-chip multiprocessing and low-power, "delivery" applications. It allows averaging out irregularity, while keeping the other advantages of SIMD processing.

# Bibliography


[AB86] M. Auguin and F. Boeri. The OPSILA computer. In M. Cosnard, Y. Robert, P. Quinton, and M. Tchuente, editors, *Parallel Algorithms and Architectures*, pages 143–153. North-Holland, 1986.

[AGWFH94] Nael Abu-Ghazaleh, Philip A. Wilsey, Xianzhi Fan, and Debra Hensgen. Variable instruction issue for efficient MIMD interpretation on SIMD machines. In H. J. Siegel, editor, *Proc. 8th International Parallel Processing Symposium, Cancun, Mexico*, pages 304–310. IEEE Comp. Soc. Press, 1994.

[AJ97] K. Asanović and D. Johnson. Torrent architecture manual. *Technical Report CSD-97-930*, Computer Science Division, University of California at Berkeley, 1997.

[AK96] Y. Amit and A. Kong. Graphical templates for model registration. *IEEE PAMI*, 18(3):225–236, March 1996.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf., Atlantic City, H.J.*, pages 483–485. AFIPS Press, 1967.

[ANSC94] James B. Armstrong, Mark A. Nichols, Howard Jay Siegel, and Kenneth H. Casey. Image correlation: A case study to examine SIMD/MIMD tradeoffs for scalable parallel algorithms. In *International Conference on Parallel Processing, Penn. State Univ.*, pages I–241 – I–245. CRC Press, 1994.

[AP92] H. M. Alnuweiri and V. K. Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE PAMI*, 14(10):1014–1034, October 1992.

[Asa98] K. Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, 1998.

[Ass94] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors*. Semiconductor Industry Association, 1994.

[Ass97] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors.* Semiconductor Industry Association, 1997.

[BA90] J. Ben-Arie. The probabilistic peaking effect of viewed angles and distances with application to 3-D object recognition. *IEEE PAMI*, 12(8):760–776, August 1990.

[Bai93] J. Bailey. *A VLSI Interconnect Strategy for Biologically Inspired Artificial Neural Networks.* PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, 1993.

[Bal79] Dana Ballard. Generalizing the Hough transform to detect arbitrary shapes. *Technical Report TR 55*, Dept. of Computer Science, University of Rochester, 1979.

[Bas93] Ronen Basri. Recognition by prototypes. In *IEEE Conf. on Computer Vision and Pattern Recognition, New York, NY*, pages 161–167. IEEE Comp. Soc. Press, 1993.

[BBB+95] W. Bowhill, S. Bell, B. Benschneider, A. Black, S. Britton, R. Castelino, D. Donchin, J. Edmondson, H. Fair, P. Gronowski, A. Jain, P. Kroesen, M. Lamere, B. Loughlin, S. Mehta, R. Mueller, R. Preston, S. Santhanam, T. Shedd, M. Smith, and S. Thierauf. Circuit implementaion of a 300-MHz 64-bit second-generation CMOS Alpha CPU. *Digital Technical Journal*, 7(1):100–105, 1995.

[BC82] R. C. Bolles and R. A. Cain. Recognizing and locating partially visible objects: the local feature focus method. *Int. J. Robotics Research*, 1(3):57–82, 1982.

[BCJ90] Edward C. Bronson, T. L. Casavant, and L. H. Jamieson. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. *IEEE Trans. Parallel and Distributed Systems*, 1(2):195–205, April 1990.

[BCK92] R. M. Bolle, A. Califano, and R. Kjeldsen. A complete and extandable approach to visual recognition. *IEEE PAMI*, 14(5):534–548, May 1992.

[BCK+97] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In *Proc. 24th Ann. Intl. Symp. on Computer Architecture (ISCA '97), Denver, CO*, pages 55–62. ACM Press, 1997.

[BCKM90] Ruud M. Bolle, Andrea Califano, Rick Kjeldsen, and Rakesh Mohan. Active 3D object models. In *Third Intl. Conf. on Computer Vision*, pages 329–333. IEEE Comp. Soc. Press, 1990.

[BD94] Shashi D. Buluswar and Bruce A. Draper. Non-parametric classification of pixels under varying outdoor illumination. In *ARPA Image Understanding Workshop, Monterey, CA*, pages 1619–1625. Morgan Kaufmann, 1994.

[BDBH89] J. Brolio, B. A. Draper, J. R. Beveridge, and A. R. Hanson. ISR: A database for symbolic processing in computer vision. *IEEE Computer*, 22(12):22–30, December 1989.

[BG97] D. Burger and J. Goodman. Billion-transistor architectures. introduction to a special issue. *IEEE Computer*, 30(9):46–48, September 1997.

[BGK96] D. Burger, J. R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. *Computer Architecture News*, 24(2):78–89, May 1996.

[BGKN89] L. R. Bahl, P. S. Gopalakrishnan, D. Kanevsky, and D. Nahamoo. Matrix fast match: A fast method for identifying a short list of candidate words for decoding. In *ICASSP-89, Glasgow, UK*, pages 345 – 348. IEEE Comp. Soc. Press, 1989. (paper S6.24).

[BH86] R. C. Bolles and P. Horaud. 3DPO: A three-dimensional part orientation system. *Int'l. J. Robotics Research*, 5(3):3–26, 1986.

[BH88] J. Bailey and D. Hammerstrom. Why VLSI implementations of associative VLCNs require connection multiplexing. In *Proc. Intl. Conf. on Neural Networks, Denver, CO*, pages 158–164. Morgan Kaufmann, 1988.

[BJM90] L. R. Bahl, F. Jelinek, and R. L. Mercer. A maximum likelihood approach to continuous speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*, pages 308–319. Morgan Kaufmann, 1990.

[BKS91] T. B. Berg, S-D. Kim, and H. J. Siegel. Limitations imposed on mixed-mode performance of optimized phases due to temporal juxtaposition. *J. Parallel and Distributed Computing*, 13(2):154–169, October 1991.

[BM94] H. A. Bourlard and N. Morgan. *Connectionist Speech Recognition*. Kluwer Academic, 1994.

[BS92] Suchendra M. Bhandarkar and Minsoo Suk. Qualitative features and the generalized Hough transform. *Pattern Recognition*, 25(9):987–1006, 1992.

[Bur97] D. Burger. System-level implications of processor-memory integration. In *Proc. 24th Ann. Intl. Symp. on Computer Architecture (ISCA '97), Denver, CO*, pages 1–10. ACM Press, 1997.

[BYP+91] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proc. 18th Ann. Intl. Symp. on Computer Architecture, Toronto, Canada*, pages 276–286. ACM Press, 1991.

[Can86] J. Canny. A computational approach to edge detection. *IEEE PAMI*, 8(6):679–698, November 1986.

[CB90] P. B. Chou and C. M. Brown. The theory and practice of Bayesian image labeling. *Int'l. J. Computer vision*, 4(2):185–210, 1990.

[CEJK92] T. Collette, H. Essafi, D. Juvin, and J. Kaiser. SYMPATIX: a SIMD computer performing the low and intermediate levels of image processing. In D. Etiemble and J.-C. Syre, editors, *PARLE '92 Parallel Architectures and Languages Europe, Paris, France*, pages 147–161. Springer-Verlag, 1992. (LNCS, v. 605).

[CHS91] O. I. Camps, R. M. Haralick, and L. G. Shapiro. PREMIO: an overview. In *IEEE Workshop on Directions in Automated CAD-based Vision, Maui, Hawaii*, pages 11–21, 1991.

[CJ91] D. T. Clemens and D. W. Jacobs. Space and time bounds on indexing 3-D models from 2-D images. *IEEE PAMI*, 13(10):1007–1017, October 1991.

[CKP95] William J. Christmas, Josef Kittler, and Maria Petrou. Structural matching in computer vision using probabilistic relaxation. *IEEE PAMI*, 17(8):149 – 164, August 1995.

[CM91] Andrea Califano and Rakesh Mohan. Multidimensional indexing for recognizing visual shapes. In *IEEE Conf. on Computer Vision and Pattern Recognition, Maui, Hawaii*, pages 23–34. IEEE Comp. Soc. Press, 1991.

[Con83] J. Conrad. *Lord Jim*. Buccaneer Books, 1983.

[Cor92] Thinking Machines Corporation. *Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, 1992.

[Cri97] R. Crisp. Direct Rambus technology: the new main memory standard. *IEEE Micro*, 17(6):18–28, November/December 1997.

[Dav70] H. A. David. *Order Statistics.* Wiley, 1970.

[DBAG88] P. Duclos, F. Boeri, M. Auguin, and G Giraudon. Image processing on a SIMD/SPMD architecture: Opsila. In *9th Intl. Conf. on Pattern Recognition, Rome, Italy*, pages 430–433. IEEE Comp. Soc. Press, 1988.

[DBKK98] J. Dreibelbis, J. Barth, R. Kho, and H. Kalter. An ASIC library granular DRAM macro with built-in self test. In *Proc. Intl. Solid-State circuits Conf (ISSCC98), San Francisco, CA*, pages 74–75. IEEE Press, 1998.

[DZO92] H. G. Dietz, A. Zaafrani, and M. O'Keefe. Static scheduling for barrier MIMD architectures. *J. Supercomputing*, 5(4):263–289, 1992.

[Ede86] G. Edelman. *Neural Darwinism: The Theory of Neuronal Group Selection.* Basic Books, 1986.

[EG95] A. Essen and S. Goldstein. Performance evaluation of the superscalar speculative execution HaL SPARC64 processor. In *Proc. Hot Chips VII, Stanford Univ., Palo Alto, CA*, pages 59–73. IEEE Comp. Soc. Press, 1995.

[ERB+95] J. Edmondson, P. Rubinfeld, P. Bannon, B. Benschneider, D. Bernstein, R. Castelino, E. Cooper, D. Dever, D. Donchin, T. Fischer, A. Jain, S. Mehta, J. Meyer, R. Preston, V. Rajagopalan, C. Somanathan, S. Taylor, and G. Wolrich. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–132, 1995.

[FD91] A. A. Farag and E. J. Delp. Edge linking by sequential search. In *Model-Based Vision Development and Tools, Proc. SPIE*, pages 198–216. SPIE, 1991. (SPIE v. 1609).

[FH86] O. D. Faugeras and M. Hebert. The representation, recognition, and locating of 3-D objects. *Intl. J. Robotics Research*, 5(3):27–52, Fall 1986.

[FH89] Allan L. Fisher and Peter T. Highnam. Computing the Hough transform on a scan line array processor. *IEEE PAMI*, 11(3):262–265, March 1989.

[Fis94] M. A. Fischler. The perception of linear structure: A generic linker. In *ARPA Image Understanding Workshop, Monterey, CA*, pages 1565–1579. Morgan Kaufmann, 1994.

[FJ91] P. J. Flynn and A. K. Jain. BONSAI: 3-D object recognition using constrained search. *IEEE PAMI*, 13(10):1066–1075, October 1991.

[Fos96] R. C. Foss. Implementing application specific memory. In *Proc. Intl. Solid-State Circuits Conf. (ISSCC96), San Francisco, CA*, pages 260–261 and 210–211. IEEE Press, 1996. (paper FP 16.1).

[Fox88] G. C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proc. Third Conf. on Hypercube Concurrent Computers and Applications*, pages 897–955. ACM Press, 1988.

[Fox89] G. C. Fox. 1989 – the first year of the parallel supercomputer. In *Proc. Fourth Conf. on Hypercubes, Concurrent Computers, and Applications*, pages 1–37. ACM Press, 1989.

[FPC+97] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *Proc. 24th Ann. Intl. Symp. on Computer Architecture (ISCA '97), Denver, CO*, pages 120–130. ACM Press, 1997.

[GBKQ96] N. Gaddis, J. R. Butler, A. Kumar, and W. J. Queen. A 56-entry instruction reorder buffer. In *Intl. Solid-State Circuits Conf. (ISSCC96), San Francisco, CA*, pages 212–213 and 168–169 and 447. IEEE Press, 1996. (paper FP 13.2).

[Gei95] Davi Geiger. Dynamic programming for detecting, tracking, and matching deformable contours. *IEEE PAMI*, 17(3):294–302, March 1995.

[Ger92] D. Gerogiannis. Programming intermediate level vision tasks on parallel machines. In *11th Intl. Conf. on Pattern Recognition D: Architectures for Vision and Pattern Recognition, The Hague, The Netherlands*, volume IV, pages 119–123. IEEE Comp. Soc. Press, 1992.

[GO92] D. Gerogiannis and S. Orphanoudakis. Efficient use of parallelism in intermediate level vision tasks. In *11th Intl. Conf. on Pattern Recognition D: Architectures for Vision and Pattern Recognition, The Hague, The Netherlands*, volume IV, pages 160–164. IEEE Comp. Soc. Press, 1992.

[GR90] L. S. Gillick and R. Roth. A rapid match algorithm for continuous speech recognition. In *DARPA Speech and Natural Language Workshop, Hidden Valley, Pennsylvania*, pages 170–172. Morgan Kaufmann, 1990.

[Gri90] W. Eric L. Grimson. *Object Recognition by Computer: The Role of Geometric Constraints.* MIT Press, 1990.

[GRV95] T. Gautier, J. L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In A. Ferreira and J. Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems, Proc. 2nd Intl. Workshop, IRREGULAR '95, Lyon, France, Sept. 4-6, 1995*, pages 1–25. Springer-Verlag, 1995. (LNCS, v. 950).

[GW92] R. C. Gonzalez and R. E. Woods. *Digital Image Processing.* Addison-Wesley, 1992.

[Ham90] D. Hammerstrom. A VLSI architecture for high-performance, low-cost, on-chip learning. In *Proc. Int'l. Joint Conf. Neural Networks, San Diego*, pages II–537 - II–543. IEEE Comp. Sci. Press, 1990.

[He91] Y. He. *Planar Shape and Handwritten Word Recognition Using Hidden Markov Models.* PhD thesis, SUNY, Buffalo, 1991.

[Hin92] G. E. Hinton. Adaptive elastic models for hand-printed character recognition. In J.E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing 4, Denver, CO*, pages 512–519. Morgan Kaufmann, 1992.

[HNO97] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[HP90] J. Hampshire and B. Pearlmutter. Equivalence proofs for multi-layer perceptron classifiers and the Bayesian discriminant function. In Touretzky, Elman, Sejnowski, and Hinton, editors, *Proc. 1990 Connectionist Models Summer School*, pages 115–122. Morgan Kaufmann, 1990.

[HP96] J. Hennessey and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[HQ90] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1990.

[HU90] Daniel P. Huttenlocher and Shimon Ullman. Recognizing solid objects by alignment with an image. *Intl. J. of Computer Vision*, 5(2):195–212, 1990.

[Jac96] D. W. Jacobs. The space requirements of indexing under perspective projections. *IEEE PAMI*, 18(3):330–333, March 1996.

[Jel69] F. Jelinek. Fast sequential decoding algorithm using a stack. *IBM J. Research and Development*, 13:675–685, November 1969.

[JKK95] P. P. Jonker, E. R. Komen, and M. A. Kraaijveld. A scalable real-time image processing pipeline. *Machine Vision and Appl.*, 8(2):110–121, 1995.

[Jon93] Pieter P. Jonker. An SIMD-MIMD architecture for image processing and pattern recognition. In M. A. Bayoumi, L. S. Davis, and K. P. Valavanis, editors, *Computer Architectures for Machine Perception (CAMP '93), New Orleans, Louisiana*, pages 222–230. IEEE Comp. Soc. Press, 1993.

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proc. 3th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III), Boston, MA*, pages 272–282. ACM Press, 1989.

[JZL96] Anil K. Jain, Yu Zhong, and Sridhar Lakshmanan. Object matching using deformable templates. *IEEE PAMI*, 18(3):267–277, March 1996.

[KD92] S. W. Keckler and W. J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proc. 19th Ann. Intl. Symp. on Computer Architecture, Gold Coast, Australia*, pages 155–160. ACM Press, 1992.

[KHG+91] P. Kenny, R. Hollan, V. Gupta, M. Lennig, P. Mermelstein, and D. O'Shaughnessy. A*-admissible heuristics for rapid lexical access. In *ICASSP-91, Toronto, Canada*, pages 689–692. IEEE Comp. Soc. Press, 1991.

[KLL+93] P. Kenny, P. Labute, Z. Li, R. Hollan, M. Lennig, and D. O'Shaughnessy. A new fast match for very large vocabulary continuous speech recognition. In *ICASSP-93, Minneapolis, MI*, pages II–656 – II–659. IEEE Comp. Soc. Press, 1993.

[Kog94] Peter M. Kogge. EXECUBE - a new architecture for scaleable MPPs. In *International Conference on Parallel Processing, Penn. State Univ.*, pages 34–38. CRC Press, 1994.

[KPP+97] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, H. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[Lee95] Ruby B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

[LEE+97] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting Thread-Level parallelism to Instruction-Level parallelism via simultaneous multithreading. *ACM Trans. Computer Systems*, 15(3):322–354, August 1997.

[LLNK96] J. Lotz, G. Lesartre, S. Nalfziger, and D. Kipp. A quad-issue out-of-order RISC CPU. In *Intl. Solid-State Circuits Conf. (ISSCC96), San Francisco, CA*, pages 210–211,166–167,446. IEEE Press, 1996. (paper FP 13.1).

[Low85] D. Lowe. *Perceptual Organization and Visual Recognition.* Kluwer, 1985.

[Low90] B. Lowerre. The Harpy speech understanding system. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*, pages 576–586. Morgan Kaufmann, 1990.

[LS97] M. H. Lipasti and J. P. Shen. Superspeculative microarchitecture for beyond AD 2000. *IEEE Computer*, 30(9):59–66, September 1997.

[LSW88] Y. Lamdan, J. T. Schwartz, and H. J. Wolfson. Object recognition by affine invariant matching. In *IEEE Conf. on Computer Vision and Pattern Recognition, Ann Arbor, MI*, pages 335–344. IEEE Press, 1988.

[Lun87] S. F. Lundstrom. Applications considerations in the system design of highly concurrent multiprocessors. *IEEE Trans. Computers*, C-36(11):1292–1309, November 1987.

[LW88] Y. Lamdan and H. J. Wolfson. Geometric hashing: A general and efficient model-based recognition scheme. In *Second Intl. Conf. on Computer Vision, Tampa, FL*, pages 238–249. IEEE Comp. Soc. Press, 1988.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proc. 19th Ann. Intl. Symp. on Computer Architecture, Gold Coast, Australia*, pages 46–57. ACM Press, 1992.

[LWS96] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and load value prediction. In *Proc. 7th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Cambridge, MA*, pages 138–147. ACM Press, 1996.

[Mat97] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, September 1997.

[ME93] S-M Moon and K. Ebcioglu. On performance and efficiency of VLIW and superscalar. In *Intl. Conf. on Parallel Processing, Penn. State Univ*, pages II–283 – II–287. CRC Press, 1993.

[Mel96] H. Melville. *Moby Dick*. Buccaneer Books, 1996.

[MM93] Serge M. Manning and David G. Meyer. Analysis of asynchronous execution streams with I-caching in massively parallel systems. *J. Parallel and Distributed Computing*, 19(3):279–291, November 1993.

[Mor] Nelson Morgan. Personal communication. (Intl. Computer Science Inst., Berkeley, CA).

[MR97] John Moody and Thorsteinn Rögnvaldsson. Smoothing regularizers for projective basis function networks. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing 9, Denver, CO*, pages 585–591. MIT Press, 1997.

[ND92] P. J. Narayanan and L. S. Davis. Replicated data algorithms in image processing. *CVGIP: Image Understanding*, 56(3):351–365, November 1992.

[NHUTO92] H. Ney, R. Haeb-Umbach, B.-H. Tran, and M. Oerder. Improvements in beam search for 10000-word continuous speech recognition. In *ICASSP-92, San Francisco, CA*, pages I–9 – I–12. IEEE Comp. Soc. Press, 1992.

[Nil86] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1986.

[NO94] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. 21st Ann. Int. Symp. on Computer Architecture, Chicago*, pages 166–175. ACM Press, 1994.

[NSD93] Mark A. Nichols, Howard Jay Siegel, and Henry G. Dietz. Data management and control-flow aspects of an SIMD/SPMD parallel language/compiler. *IEEE Trans. Parallel and Distributed Systems*, 4(2):222–234, February 1993.

[Ols93] Clark F. Olson. Fast alignment using probabilistic indexing. In *IEEE Conf. on Computer Vision and Pattern Recognition, New York, NY*, pages 387–392. IEEE Press, 1993.

[Ols95] C. F. Olson. Probabilistic indexing for object recognition. *IEEE PAMI*, 17(5):518–522, May 1995.

[PAC+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent DRAM: IRAM. *IEEE Micro*, 17(2):34–44, March/April 1997.

[Pau91] D. B. Paul. Algorithms for an optimal A* search and linearizing the search in the stack decoder. In *ICASSP-91, Toronto, Canada*, pages 693–696. IEEE Comp. Soc. Press, 1991.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity effective superscalar processors. In *Proc. Int'l Symp. Computer Architecture, New York*, pages 206–218. ACM Press, 1997.

[PPE+97] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer*, 30(9):51–57, September 1997.

[PWTH93] Michael Philippsen, Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. Project Triton: Towards improved programmability of parallel machines. In *Proc. 26th Hawaii Conf. on System Sciences*, pages 192–201. IEEE Press, 1993.

[Rab90] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*, pages 267–296. Morgan Kaufmann, 1990.

[RB93] I. D. Reid and J. M. Brady. Recognition of object classes from range data. In *Fourth Intl. Conf. on Computer Vision, Berlin, Germany*, pages 302–307. IEEE Comp. Soc. Press, 1993.

[Rei91] C. C. Reinhart. *Specifying Parallel Processor Architectures for High-Level Computer Vision Algorithms.* PhD thesis, Univ. Southern California, 1991.

[Rei93] Thomas H. Reiss. Object recognition using algebraic and differential invariants. *Signal Processing*, 32(3):367–395, 1993.

[RH93] Isidore Rigoutsos and Robert Hummel. Distributed Bayesian object recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition, New York, NY*, pages 180–186. IEEE Press, 1993.

[Ris86] J. Rissanen. Complexity of strings in the class of Markov sources. *IEEE Trans. Information Theory*, IT-32(4):526–532, July 1986.

[RL90] L. R. Rabiner and S. E. Levinson. Isolated and connected word recognition - theory and selected applications. In Alex Waibel and Kai-Fu Lee, editors, *Readings in Speech Recognition*, pages 115–153. Morgan Kaufmann, 1990.

[Roc93] T. Rockoff. *An Analysis of Instruction-Cached SIMD Computer Architecture.* PhD thesis, Carnegie Mellon University, 1993.

[RST94] D. Ron, Y. Singer, and N. Tishby. The power of amnesia. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing 6, Denver, CO*, pages 176–183. Morgan Kaufmann, 1994.

[RV94] N. Ranganathan and S. Venugopal. An efficient VLSI architecture for template matching. In *International Conference on Parallel Processing, Penn. State Univ.*, pages I–224 – I–231. CRC Press, 1994.

[SB88] G. Salton and C Buckley. Parallel text search methods. *CACM*, 31(2):202–215, February 1988.

[SB95] Kuntal Sengupta and Kim L. Boyer. Organizing large structural modelbases. *IEEE PAMI*, 17(4):321–332, April 1995.

[SDC94] P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[SG93] K. B. Sarachik and W. E. L. Grimson. Gaussian error models for object recognition. In *IEEE Conf. on Computer Vision and Pattern Recognition, New York, NY*, pages 400–406. IEEE Comp. Soc. Press, 1993.

[SH90] F. K. Soong and E.-F. Huang. A fast tree-trellis search for finding the $n$-best sentence hypotheses in continuous speech recognition. *J. Acoustical Soc.*, 87:105–106, May 1990.

[SH91] Frank K. Soong and Eng-Fong Huang. A tree-trellis based fast search for finding the N best sentence hypotheses in continuous speech recognition. In *ICASSP-91, Toronto, Canada*, pages 705–708. IEEE Comp. Soc. Press, 1991.

[Sha97] A. Sharma. *Semiconductor Memories.* IEEE Press, 1997.

[SHJ97] P. Smyth, D. Heckerman, and M. Jordan. Probabilistic independence networks for hidden Markov probability models. *Neural Computation*, 9(2):227–269, February 1997.

[Sic75] H. S. Sichel. On a distribution law for word frequencies. *J. American Statistical Assoc.*, 70:542–547, 1975.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proc. 3th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS III), Boston, MA*, pages 290–302. ACM Press, 1989.

[Ski91] D. B. Skillicorn. Models for practical parallel computation. *Int'l J. Parallel Programming*, 20(2):133–158, 1991.

[SPN96] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. *Computer Architecture News*, 24(2):90–101, May 1996.

[SS95] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proc. IEEE*, 83(12):1609–1624, December 1995.

[Sto87] H. S. Stone. Parallel querying of large databases: A case study. *IEEE Computer*, 19(10):11–21, October 1987.

[SV97] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer*, 30(9):68–74, September 1997.

[SW95] Wei Shu and Min-You Wu. Asynchronous problems on SIMD parallel computers. *J. Parallel and Distributed Computing*, 6(7):704–713, July 1995.

[TEE+95] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 22nd Ann. Intl. Symp. on Computer Architecture, Santa Margherita, Ligure, Italy*, pages 191–202. ACM Press, 1995.

[TGH92] K. B. Theobald, G. R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *25th Ann. Intl. Symp. on Microarchitecture*, pages 10–19. ACM Press, 1992.

[TH91] Kenneth B. Thornton and Robert M. Haralick. Model-based point matching. In *Model-Based Vision Development and Tools, Proc. SPIE*, pages 251–261. SPIE, 1991. (SPIE v. 1609).

[TMH+98] R. Torrance, I. Mes, B. Hold, D. Jones, J. Crepeau, P. DeMone, D. MacDonald, C. O'Connell, P. Gillingham, R. White, S. Duggins, and D. Fielder. A 33 GB/s 13.4Mb integrated graphics accelerator and frame buffer. In *Proc. Intl. Solid-State Circuits Conf. (ISSCC98), San Francisco, CA*, pages 340–341. IEEE Press, 1998.

[USA94] USAF. Model-driven automatic target recognition, report of the ARPA/SAIC system architecture study group, October 1994. (USAF Moving and Stationary Target Acquisition and Recognition Program (MSTAR), Automatic Target Recognition Branch, Wright Laboratory, Wright-Patterson AFB, Dayton, OH).

[vHK92] Reinhard v. Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. *ACM SIGPLAN Notices*, 27(7):188–199, July 1992.

[WAK+96] John Wawrzynek, K. Asanović, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. SPERT-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–87, March 1996.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proc. 4th Intl. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), Santa Clara, CA*, pages 176–188. ACM Press, 1991.

[WB94] S-J. Wang and T. O. Binford. Model-based edgel aggregation. In *ARPA Image Understanding Workshop, Monterey, CA*, pages 1589–1593. Morgan Kaufmann, 1994.

[Web92] Jon A. Webb. Steps toward architecture-independent image processing. *IEEE Computer*, 25(2):21–31, February 1992.

[Wee91] Charles C. Weems. Architectural requirements of image understanding with respect to parallel processing. *Proc. IEEE*, 79(4):537–547, April 1991.

[Wee93] Charles C. Weems. The second generation image understanding architecture and beyond. In M. A. Bayoumi, L. S. Davis, and K. P. Valavanis, editors, *Computer Architectures for Machine Perception (CAMP '93), New Orleans, Louisiana*, pages 276–285. IEEE Comp. Soc. Press, 1993.

[Wee94] Charles C. Weems. The next generation image understanding architecture. In *ARPA Image Understanding Workshop, Monterey, CA*, pages 587–594. Morgan Kaufmann, 1994.

[Wee97] C. Weems. Asynchronous SIMD : an architectural concept for high performance image processing. In C. Weems, editor, *Computer Architectures for Machine Perception (CAMP '97), Cambridge, MA*, pages 235–243. IEEE Comp. Soc. Press, 1997.

[Wei93] Daphna Weinshall. Model-based invariants for 3-D vision. *Intl. J. of Computer Vision*, 10(1):27–42, 1993.

[WHAG+92] Philip A. Wilsey, Debra A. Hensgen, Nael B. Abu-Ghazaleh, Charles E. Slusher, and David Y. Hollinden. The concurrent execution of non-communicating programs on SIMD processors. In H. J. Siegel, editor, *Fourth Symposium on the Frontiers of Massively Parallel Computation, McLean, Virginia*, pages 29–36. IEEE Comp. Soc. Press, 1992.

[WI95] M. D. Wheeler and K. Ikeuchi. Sensor modeling, probabilistic hypothesis generation, and robust localization for object recognition. *IEEE PAMI*, 17(3):252–265, March 1995.

[WLH+89] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash. The image understanding architecture. *Intl. J. Computer Vision*, 2(3):251–282, 1989.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

[WRF95] M. Weinberger, J. Rissanen, and M. Feder. A universal finite memory source. *IEEE Trans. Information Theory*, 41(3):643–652, May 1995.

[WSA+94] Daniel W. Watson, H. J. Siegel, M. K. Antonio, M. A. Nichols, and M. J. Atallah. A block-based mode selection model for SIMD/SPMD parallel environments. *J. Parallel and Distributed Computing*, 21(3):271–288, June 1994.

[WTS+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, September 1997.

[YHO97] T. Yamauchi, L. Hammond, and K. Olukotun. A single chip multiprocessor integrated with DRAM. In *Proc. 24th Ann. Intl. Symp. on Computer Architecture (ISCA '97), Denver, CO*, pages 255–261. ACM Press, 1997.

[ZA92] Y. Zhang and G. B. Adams III. Exploiting instruction level parallelism with the DS architecture. In *Proc. Intl. Conf. Parallel Processing, Penn. State Univ.*, pages I–230 – I–237. CRC Press, 1992.

[Zip32] G. K. Zipf. *Selected Studies of the Principle of Relative Frequency in Language*. Harvard Univ. Press, 1932.

# Appendix A

# SIMD HOVS pseudo-code

This appendix gives pseudo-code for various SIMD implementations of the HOVS algorithm. For clarity and brevity, the pseudo-code focuses on the main loop, and omits initialization and the handling of starting and ending conditions.

In these implementations, inputs from the classifier at iteration $t$ are $\log m(h_t; o_t)$; using logarithmic representations is needed to avoid underflow problems from repeated multiplications. All context probabilities, $p(h_t | h_{t-\alpha}^{t-a})$, are thus represented as log-probabilities, too.

**Notation:** `p.x` is a variable local to a particular PE `p`, being accessed *only* on that PE. When such variables are accessed in parallel for all (active) PEs, the notation `*.x` is used. `p.vp[]` is the array of virtual PEs for `p`. `v.x` refers to a variable local to a virtual PE, and is shorthand for `p.vp[k].x` for some `k`; analogously, `*.v.x` is shorthand for `*.vp[k].x`. `host.x` refers to a variable on the host. All unqualified variables are global: distributed to each PE, but with the same value on each PE. I use `p` and `v` to refer both to the (virtual) PE, or its integer identifier, as convenient.

Each (V)PE corresponds to a set of contexts, all having a common final state or state sequence, and maintains the information for that state (sequence) at each timestep. This PE (VPE) is denoted `nxtp` (`nxtv`) in the code, and its identity is broadcast from the host (when it cannot be determined trivially). Nodes of the context graph are numbered, context node `n` having number `n.num`, such that (V)PE `p` corresponds to the set of nodes, `n` for which `*.lo[p] <= n.num <= *.hi[p]`. As the `lo` and `hi` information is global, it may be either stored locally, or broadcast. Each (V)PE `p` has local variables `p.LO == *.lo[p]` and `p.HI == *.hi[p]`.

Each (V)PE contains two arrays, holding the context information. `prob[n][h]` contains the context probabilities $p(h|n)$ for the various contexts, `n`, in the (V)PE. `nxtctxt[n][h]` contains the information to implement the $\hat{\gamma}_h(n)$ function. As it is not done in parallel, computing the $\hat{\gamma}_h(n)$ function may also be done on the host. This is the likely implementation in most cases, as the computation is small, and doing it on the host means that PEs need not store the `nxtctxt` array, which essentially *halves* the local storage used by the algorithm. Nonetheless, for completeness the pseudo-code will assume the $\hat{\gamma}_h(n)$ is computed on the PE array. For the nodes corresponding to a particular PE, the `prob` and `nxtctxt` arrays are laid out in PE-local memory so that the address of each array for context node `n` can be easily calculated from `n.num`, `p.LO`, and `p.HI`. Denote these calculations as `probAddr(n.num, p.LO, p.HI)` and `nxtAddr(n.num, p.LO, p.HI)`.

Finally, the local variable `cScore` holds the broadcast value of the current classifier output.

```
while not end of string
  {
    for h in 1..|H|                                  // O(|H|)
      {
        // broadcast classifier output score for h at this time
        cScore = <broadcast score>

        // calculate score in context
        *.score = *.prob[h] + *.prevScore + cScore

        // find PE with maximum score
        maxp = PMAX{*.score}                          // O(1)

        // point-to-point communication between nxtp and maxp
        nxtp = <PE for h>
        nxtp.nxtNode = maxp.nxtctxt[h]
        nxtp.nxtScore = maxp.score

        // record backppointer on host
        host.backptr[p][t] = maxp
      }
    // update all PEs for next timestep
    *.prevScore = *.nxtScore
    *.prob = *.probAddr(*.nxtNode, *.LO, *.HI)
    *.nxtctxt = *.nxtAddr(*.nxtNode, *.LO, *.HI)
    t++
  }
```

Figure A.1: SIMD HOVS pseudo-code for the basic case when $|H| = P$. Globally, there is one PE for each state of $H$, and the PE for a state $h$ contains all contexts ending in $h$. Computation of `nxtp` is trivial, as the PEs are 1:1 with the states. Complexity is $O(|H|) = O(|H|^2/P)$.

```
while not end of string
  {
    for h in 1..|H|
      {
        // broadcast classifier output score for h at this time
        cScore = <broadcast score>

        // find max score for the virtual PEs on each PE
        // score is for h in that vpe's active context
        *.localmax = minusInfinity
        for k in 1..K
          {
            with *.v = *.vp[k]
            // calculate score in context
            *.v.score = *.v.prob[h] + *.v.prevScore + cScore
            // record max and vpe giving it
            if *.v.score > *.localmax
                *.localmax = *.v.score
                *.localmaxv = k
          }

        // find PE with maximum score
        maxp = PMAX{p.localmax}


        //  VPE with contexts ending in 'h' (exactly 1 such)
        (nxtp, nxtk, nxtv)  = <virtual PE for  h>

        // point-to-point communication between nxtp and maxp
        with v = nxtp.vp[nxtk]
        v.nxtScore = maxp.localmax
        v.nxtNode = maxp.vp[localmaxv].nxtctxt[h]
        host.backptr[h][t] = maxp
      }
    for k in 1..K
      {
        with *.v = *.vp[k]
        *.v.prevScore = *.v.nxtScore
        *.v.prob = *.v.probAddr(*.v.nxtNode, *.v.LO, *.v.HI)
        *.v.nxtctxt = *.v.nxtAddr(*.v.nxtNode, *.v.LO, *.v.HI)
      }
    t++
  }
```

Figure A.2: SIMD HOVS pseudo-code when $|H|$ exceeds the number of PEs, $P$. Each PE contains $K = \lceil |H|/P \rceil$ VPEs; globally, there is one VPE for each state of $H$, and the VPE for a state $h$ contains all contexts ending in $h$. Complexity is $|H|\lceil |H|/P \rceil \approx |H|^2/P$.

```
while not end of string
  {
    for h in 1..|H|
      {
        // broadcast classifier output score for h at this time
        cScore = <broadcast score>

        for nxtv with contexts ending in 'h'          // O(KP/|H|)
          {

            // find max score for the VPEs on each VP
            // score is for h in that vp's active context
            *.localmax = minusInfinity
            for k in 1..K
              {
                with *.v = *.vp[k]
                // calculate score in context
                *.v.score = *.v.prob[h] + *.v.prevScore + cScore
                // determine if context of *.v can precede those of nxtv
                *.v.canPrecede = (lo[nxtv] <= *.v.nxtctxt[h].num <= hi[nxtv])
                // record max and vp giving it
                if *.canPrecede && *.score > *.localmax
                  *.localmax = *.v.score
                  *.localmaxv = k
              }

            // find PE with maximum score
            maxp = PMAX{*.localmax}

            // find PE and vpe index for nxtv
            (nxtp, nxtk)  = <PE and vpe index for nxtv>

            // point-to-point communication between nxtp and maxp
            with v = nxtp.vp[nxtk]
            v.nxtScore = maxp.localmax
            v.nxtNode = maxp.vp[localmaxv].nxtctxt[h]

            // record backpointer on host
            host.backptr[h][t] = maxp
          }
      }
    for k in 1..K
      {
        with *.v = *.vp[k]
        *.v.prevScore = *.v.nxtScore
        *.v.prob = *.v.probAddr(*.v.nxtNode, *.v.LO, *.v.HI)
        *.v.nxtctxt = *.v.nxtAddr(*.v.nxtNode, *.v.LO, *.v.HI)
      }
    t++
  }
```

Figure A.3: SIMD HOVS pseudo-code when $|H|$ is less than the number of PEs, $P$. Each VPE contains the contexts in the intersection of the active fringe with a complete subtrees of the context tree; all contexts have the same final state. $K$ is the maximum number of VPEs (or context subtrees) associated with any PE. The canPrecede compatibility condition is needed as only some contexts can precede a given context (e.g. $h^1h^2$ can only precede contexts of the form $h'h^1$). Complexity of the algorithm is $K^2P =$ (no. of contexts)$^2/P$.

```
while not end of string
  {
    for h1 in 1..|H|
      {
        // broadcast classifier output score for h1 at this time
        *.cScore = <broadcast score for h1>

        for h2 in 1..|H|
        {
          nxtp = <processor for h2>  // p_{h2}
          nxtk = <vpe index for h1>
          nxtv = nxtp.vp[nxtk]       // v_{h1.h2}
          thisk = <vpe index for h2>

          // calculate score in context
          with *.v = *.vp[thisk]       // v_{h2.h_p}
          *.score = *.v.prob[h1] + *.v.prevScore + *.cScore

          // find PE with maximum score
          maxp = PMAX{*.score}

          // point-to-point communication between nxtp and maxp
          with v = nxtp.vp[nxtk]      // v_{h1.h2}
          v.nxtScore = maxp.score
          v.nxtNode = maxp.vp[thisk].nxtctxt[h1]
          host.backptr[h1][h2][t] = maxp
        }
      }
    for k in 1..|H|
      {
        with *.v = *.vp[k]
        *.v.prevScore = *.v.nxtScore
        *.v.prob = *.v.probAddr(*.v.nxtNode, *.v.lo, *.v.hi)
        *.v.nxtctxt = *.v.nxtAddr(*.v.nxtNode, *.v.lo, *.v.hi)
      }
    t++
  }
```

Figure A.4: SIMD HOVS pseudo-code for recursion on 2-tuples of states, for $|H| = P$. Each PE, $p$ is associated with a particular state, $h$; I write $p = p_h$ and $h = h_p$, as convenient. Each PE has $|H|$ VPEs, each associated with a pair of states; for $p_{h^1}$, the VPEs are $\{v_{h^1 h^2} | h^2 \in H\}$. The VPE $v_{h^1 h^2}$ contains the context information $\{p(h|\rho(h^1 h^2))\}$.

# Appendix B

# Exhaustive Search: The Effect of Bandwidth on Parallelism

In section 4.4 we saw that for multiple on-chip processors with individual access to memory ("per-PE external memory"), chip pin count limits the number of processors to 16 or 32 for near-term process generations. Another possible architecture is for PEs to have only on-chip individual memories, and to share off-chip data I/O hardware. In this case, take the off-chip memory bandwidth to be about the same as that of a single microprocessor implemented on the same size chip. A parallel implementation may then outperform a single processor implementation due to both parallel computation and parallel access to on-chip memory (per-PE local memories)[1]. I will examine these effects by looking at exhaustive search of a set of models for ones that match given data. I look at exhaustive search not only because it is occurs in some cases (for example, vector quantization in high dimensions and deformable models, but because it offers the greatest potential for parallelism. With restricted off-chip bandwidth and per-PE local memories, the natural way to implement exhaustive search of a set of models is to *preload* the ones most likely to match into the local memories.

In this appendix, I show that, even with preloading, memory/data-path area tradeoffs imply that the potential parallelism is quite limited, unless the probability of accessing a

[1]A parallel implementation may be at a disadvantage due to the need to route requests from multiple on-chip processors to multiple external memory modules. I will ignore these effects since we will see that, even without them, potential parallelism is limited.

non-preloaded model is small or the amount of computation per model is large. Except for situations when matching a model takes much longer than reading it from off-chip, essentially all models must be preloaded for substantial parallel speedup to be realized. Although the degree of potential parallelism depends on many factors, especially the size of the model base, for "reasonable" large model base sizes (say, 1000 models, of 1-5 kB each), and "reasonable" amounts of work per model (say, computation time per model is a factor of 10 larger than the time to load it from off-chip), parallelism is limited to around 16 or 32 for near term process generations. Larger numbers of processors substantially reduce the allowed size of the model base. Essentially, there are three cases: First, if the entire model set fits on-chip, then linear speedup is obtained. This points in the direction of many small processors per chip. Second, if the model set does not fit on-chip, and there is little work per model, then parallelism is limited by bandwidth limitations and area tradeoffs, and having many PEs per chip is pointless. Third, independent of model set size, if there is much computation per model, then some parallelism is justified: if $k$ is the ratio of the time model evaluation time to the time to load a model from off-chip memory, the parallelism up to $k$ is achievable. Because of differences in clock rates on and off chip, factors of $k > 32$ may correspond to 60-100 operations per (32 bit) datum; applications with this level of computational density may be rare.

As exhaustive search allows the greatest possible parallelism, if it has limited potential parallelism due to bandwidth limitations, then so do all other potential applications. I conclude that there are two viable architectural alternatives for on-chip parallelism: if the range of target applications allows the working set of models to fit entirely on-chip, then an architecture of many small processors may be preferred. In all other cases, a few (16-32) complex PEs will be preferred.

## B.1 Modeling

My basic strategy is to assume off-chip memory technology is equivalent for both sequential and parallel architectures, so that off-chip fetches have the same speed for both. I further assume off-chip bandwidth is the same for both, if nothing else because of similar pin limitations for both. I then optimistically assume that the sequential architecture must make all fetches off-chip, while the parallel architecture may have some *preloaded* models on-chip. We are effectively assuming that, relative to on-chip data cache size, model-base size is large and temporal locality in model access is small. We are also assuming that the sequential architecture uses a general-purpose strategy for its on-chip data cache, so that the caching of models is ineffective. The basic result is that, even with these optimistic assumptions and for the highly parallelizable task of model matching, the potential on-chip parallelism is quite limited unless either, essentially all models can be preloaded, or the amount of computation done per model is large. This implies that, given the external memory modeling assumptions, a relatively general purpose parallel architecture should have only a modest number of PE's per chip, and hence that these PE's can afford to be relatively complex.

### B.1.1 Two forms of model matching

I look at two algorithmic forms for model-matching. In both, model matching is generalized to execution of a set of "tasks" having the following properties: the tasks are completely independent of one another, and each have the form "fetch all data, then process it all" (corresponding to getting the model parameters and then matching the model with the input). The set of possible tasks is known in advance, with some probability distribution over their occurrence The number of tasks to be done at any given time, $n$, has distribution $\mathcal{N}$. For simplicity, the analysis assumes tasks are all the same size. I do not model the pre- and post- matching processes of getting the input data to be matched against, and communicating the results of the matching: both are assumed to be either negligible as to time required, or independent of the algorithm and hardware architecture. To avoid overuse of the word "model", I will speak instead of *fetching* and *evaluating* a *task.*

In both algorithmic forms, the parallel architectures evaluate $P$ tasks at once, taking (on average) $k$ time per task, where a time unit is the time required (on average) for a sequential architecture to fetch a single task. As parallel architectures may have the potential for utilizing larger memory bandwidth than sequential architectures, I introduce a *bandwidth* factor, $b \geq 1$, so that the parallel architectures fetch $b$ models from memory in one time unit.

The sequential architectures evaluate one task at a time, taking (on average) $k/q$ time units. The factor $q$ measures the speed advantage of the sequential processor compared to a single PE. I assume $1 \leq q \leq P$ (if $P < q$, parallelism is pointless). This gives the following formula for the expected time on a simple sequential architecture that does not overlap fetching with evaluation, for either algorithmic form:

$$\langle time^{SEQ} \rangle = (\frac{k}{q} + 1)\langle n \rangle_{\mathcal{N}}, \tag{B.1}$$

where $\langle . \rangle_{\mathcal{X}}$ denotes expectation with respect to distribution $\mathcal{X}$.

The two algorithmic forms differ in how the tasks to be done are selected. I assume there is a fixed set of tasks indexed by $i$, $1 \leq i \leq N$, with a probability structure, $p_i$, such that $i \geq j \Rightarrow p_i \geq p_j$. For the non-stochastic form, *fixed*, the selection of tasks is deterministic: start at the first task and continue until a satisfactory match is found or all tasks are complete. Evaluation of a task may result in an "satisfactory match" signal, indicating that no more tasks need be done. Since $p_i$ is the probability that $i$ is the first satisfactory match, it is also the probability of evaluating $i$ tasks.

In *random*, the selection algorithm is viewed as data-dependent, and is modeled stochastically as follows. Add a "null" task, $p_0$, to the set of tasks (probably, but not necessarily, $p_0 \geq p_1$). Assuming $N$ large, approximate task selection without replacement by selection with replacement, and view the selection process as having $N+1$ independent trials, each determining if a particular task is to be selected, task $i$ being selected with probability $p_i$. This gives a multinomial distribution with parameters $(N+1, p_0, \ldots, p_N)$. Selection of the "null" task corresponds to not executing a task, so if it is chosen $K$ times, the number of tasks to actually be evaluated is $N - K$. Other tasks can actually be selected for evaluation only once, but if $N$ is large and $p_0$ is relatively large, the chance of a task being selected more than once is small. Note that tasks are selected independently of one another; any correlation of tasks is not modeled. Note also that, for *fixed*, the computation time for the selection is trivial, for *random* it may not be. In the latter case, I take the selection time to be the same for both the sequential and parallel algorithms; this is plausible, but not certain.

When there is DMA hardware it is reasonable to overlap fetching and evaluating models by prefetching the next set of tasks. For *fixed*, prefetch of the next tasks can be done with 100% accuracy: the exact ones to fetch are known in advance; they are the next ones in the task order. For *random*, the selection process is stochastic, so what to prefetch is not known in advance. Instead, when estimating the effect of additional DMA hardware, I assume, optimistically, that the next tasks to be fetched can be selected before the current tasks are evaluated, so that overlap of fetch and evaluation can take place. Doing the selection early will presumably increase the expected number of tasks fetched, $\langle n \rangle$, but this increase will be about the same for both the parallel and sequential architectures, as the selection algorithm is assumed to be the same for both. It will actually be slightly more for the parallel architecture due to tasks being prefetched $P$ at a time, but we will see $P$ is necessarily relatively small, so this effect may be neglected. I model the time needed for selection as part of the task evaluation time, $k$. Thus, for comparisons between architectures, the selection cost is hidden in the $k$ and $\langle n \rangle$ terms. For the sequential case, I get

$$\langle time^{SEQ-DMA} \rangle = \max\{\frac{k}{q}, 1\}\langle n \rangle_{\mathcal{N}} \tag{B.2}$$

### B.1.2 Effects of preloading tasks

Given a known distribution of the set of possible tasks, it makes sense to prefetch (*preload*) the most common ones. I wish to examine the effects of having prefetched some of the tasks into local memory. Let $m$ be a random variable giving the number of models evaluated that have not been preloaded. Let $A$ be the number of preloaded tasks. For convenience, assume that tasks are indexed so that the "preload" contains the first $A$ models; this is consistent with $i > j \Rightarrow p_i \leq p_j$, as the most probable models will be the ones that are preloaded.

For an SIMD architecture, models are evaluated in parallel, $P$ at a time, each "$P$-block" taking $k$ time units. To evaluate $n$ tasks, there are $\lceil \frac{n}{P} \rceil$ blocks to be matched. The $m$ tasks not preloaded must be fetched, each task taking $1/b$ time units. This gives

$$time(m, n) = k\lceil \frac{n}{P} \rceil + \frac{m}{b}$$

where $time(m, n)$ is the time to evaluate $n$ tasks, with $m$ of the $n$ outside the preload. This equation incorporates into $k$ the time spent fetching from on-chip (local) memory[2] Thus, $k$ ca not be too small; for simplicity I will limit $k \geq 1$ in the simulations.

Let $W$ be some distribution over the task input data, with realization $w$. Assume $m$ and $n$ are independent of $w$; this is admittedly unrealistic in that common inputs $w$ may correspond to common tasks, which are then more likely to be preloaded. Given this, one can then estimate the expected time spent per input as follows:

$$\begin{aligned}
\langle \text{time per input} \rangle_W &\approx \langle \langle time(m, n) \rangle_{\mathcal{M}|n} \rangle_{\mathcal{N}} \\
&= \langle \langle k\lceil \frac{n}{P} \rceil + \frac{m}{b} \rangle_{\mathcal{M}|n} \rangle_{\mathcal{N}} \\
&= \frac{k}{P} \langle P\lceil \frac{n}{P} \rceil \rangle_{\mathcal{N}} + \frac{1}{b} \langle \langle m \rangle_{\mathcal{M}|n} \rangle_{\mathcal{N}} \\
&= \frac{k}{P} (\langle n \rangle_{\mathcal{N}} + quant_{\mathcal{N}}(n, P)) + \frac{1}{b} \langle \langle m \rangle_{\mathcal{M}|n} \rangle_{\mathcal{N}} \qquad \text{(B.3)}
\end{aligned}$$

Here

$\mathcal{N} \doteq \text{Prob}[\mathcal{N} = n]$ is the distribution of the number of tasks to evaluate, $n$,

[2]As $k$ is the same for both the sequential and parallel implementations, any difference between the two implementations in time spent fetching from local memory is absorbed into $q$, the factor by which the sequential implementation outperforms a single PE in evaluating a task. This is appropriate, as a major reason memory accesses might be faster in the sequential implementation is from overlapping memory accesses with computation, using superscalar techniques and taking advantage of instruction level parallelism.

$(\mathcal{M}|n) \doteq \text{Prob}[\mathcal{M} = m|n]$ is the conditional probability of having $m$ of the $n$ tasks be outside the preload, and

$quant_{\mathcal{N}}(n, P) \doteq \langle P\lceil \frac{n}{P} \rceil - n \rangle_{\mathcal{N}}$ measures quantization error when the number of tasks is not a multiple of the number of processors. Note that $0 \leq quant_{\mathcal{N}}(n, P) < P$, and that $quant_{\mathcal{N}}(n, P)$ is approximately $(P-1)/2$ if the density for $N$ is smooth, and its support is large with respect to $P$ (cf. equations B.26 – B.28 and discussion).

Ignoring quantization, equation B.3 simply says that the average time per input is the time per task, $k$, times the average number of tasks per processor, $\langle n \rangle / P$, plus the expected time to load the tasks not in the preload, $\langle m \rangle_{\mathcal{M}|n}$.

To compute the expected number of tasks to fetch, $\langle m \rangle_{\mathcal{M}|n}$, let $A$ be the number of preloaded tasks, $N$ be the total number of possible tasks, and $m$ be a realization of $\mathcal{M}|n$, the number of tasks not in the preload, given that there are $n$ tasks to be evaluated. Let $p_A = 1 - \sum_{i=1}^{A} p_I(i)$ be the total probability mass of the non-preload.

For *random*, let $X_i$ be the random variable giving the number of times task $i$ was selected, $0 \leq i \leq N$, then $(X_0, \ldots, X_N) \sim Multinomial(N{+}1, p_0, (1{-}p_0)p_1 \ldots, (1{-}p_0)p_N)$ has a multinomial distribution.

Let $Y_0 \doteq X_0$, $Y_A \doteq \sum_{i=1}^{A} X_i$, and $Y_{A^c} \doteq \sum_{i=A+1}^{N} X_i$. $Y_0$ is the number of times the 'null' task was selected, $Y_0 = N + 1 -$ (number of tasks chosen), with realization $N+1-n$. $Y_A$ is the number of chosen tasks in the preload, with realization $n - m$, and $Y_{A^c}$ is the number of chosen tasks not in the preload, $m$. As exhaustive disjoint sums of multinomial components are multinomial, $(Y_0, Y_A, Y_{A^c}) \sim Multinomial(N{+}1, p_0, (1{-}p_0)(1{-}p_A), (1{-}p_0)p_A)$, and as marginal distributions of multinomial components are binomial, $Y_0 \sim Binomial(N{+}1, p_0)$, $Y_A \sim Binomial(N+1, (1-p_0)(1-p_A))$, and $Y_{A^c} \sim Binomial(N+1, (1-p_0)p_A)$. This gives

$$\begin{aligned} \langle n \rangle_{rand} &= N + 1 - \langle Y_0 \rangle \\ &= (N+1)(1-p_0), \qquad \text{(B.4)} \\ \langle n - m \rangle &= \langle Y_A \rangle \end{aligned}$$

$$\begin{aligned}
&= (N+1)(1-p_0)(1-p_A) \\
&= (1-p_A)\langle n\rangle, and && \text{(B.5)} \\
\langle m\rangle &= \langle Y_{A^c}\rangle \\
&= (N+1)(1-p_0)p_A \\
&= p_A \cdot \langle n\rangle. && \text{(B.6)} \\
& && \text{(B.7)}
\end{aligned}$$

So (B.3) becomes

$$\langle time^{SIMD}_{random}\rangle = \frac{k}{P}(\langle n\rangle_{\mathcal{N}} + quant_{\mathcal{N}}(n,P)) + \frac{p_A}{b}\langle n\rangle_{\mathcal{N}}. \tag{B.8}$$

Compared to equation B.3, this says that, for *random*, the expected number of tasks to load is the expected total number of tasks times the probability that a random task is not preloaded.

For *fixed*, if a task is evaluated, then all preceding tasks must have been done. This corresponds to

$$p_{\mathcal{M}|n}(m) = \delta_{m,\max\{0,n-A\}} = \delta_{m,(n-A)_+}$$

which (of course) is deterministic, not stochastic. Here, $\delta$ is Kronecker delta, and $(x)_+ = \max\{0, x\}$. This gives

$$\langle m\rangle_{\mathcal{M}|n} = \langle \delta_{m,(n-A)_+}\rangle_{\mathcal{M}|n} = (n-A)_+$$

and

$$\langle time^{SIMD}_{fixed}\rangle = \frac{k}{P}(\langle n\rangle_{\mathcal{N}} + quant_{\mathcal{N}}(n,P)) + \frac{1}{b}\langle (n-A)_+\rangle_{\mathcal{N}}. \tag{B.9}$$

Compared to equation B.3, this says that, for *fixed*, the expected number of tasks to load is the expected value of the number of tasks to be evaluated in excess of the number of tasks in the preload.

## B.2 Effects of DMA

I consider also a "SIMD-DMA" architecture: SIMD with additional hardware for overlapping evaluation with fetches from off-chip into local memory. To derive a formula for $time(m,n)$ for *fixed* on a SIMD-DMA architecture, first note that there are $n-m$ tasks in the preload. If $m=0$, then the time is just

$$time^{DMA}_{fixed}(0,n) = k\lceil \frac{n}{P} \rceil$$

If $m>0$, first all the $n-m$ models in the preload are evaluated, and then the remaining $m$ models outside the preload. Again, the density of $M|n$ is $\delta_{m,(n-A)_+}$, so $\langle time(m,n)\rangle_{\mathcal{M}|n} = time((n-A)_+,n)$, and $n-m=A$.

So, the first $\lfloor \frac{A}{P} \rfloor$ $P$-blocks in the preload are evaluated, then, if $A$ is not a multiple of $P$, another $P-(A-P\lfloor \frac{A}{P} \rfloor) = P-(A \bmod P)$ tasks are fetched, and a $P$-block of them and the $(A \bmod P)$ remaining unprocessed tasks in the preload are evaluated, and then the remaining $\lceil \frac{1}{P}(m-(P-A \bmod P)) \rceil$ or $\lceil \frac{m}{P} \rceil$ (depending on whether $P$ divides $A$) $P$-blocks not in the preload are fetched and evaluated. This gives, for $m>0$,

$$\begin{aligned} time^{DMA}_{fixed}(m,n) &= k\lfloor \frac{A}{P} \rfloor + \max\{k, \frac{(P-(A \bmod P))}{b}\rceil * \hat{\delta}_{0,A \bmod P}\} \\ &\quad + \max\{k, \frac{P}{b}\} * \lceil \frac{1}{P}(m - \hat{\delta}_{0,A \bmod P}(P-(A \bmod P))) \rceil \end{aligned}$$

where $\hat{\delta}_{x,y} = 1 - \delta_{x,y}$ is 1 iff $x \neq y$. Notice that I do not model overlapping fetching with the evaluation of the preloaded tasks: I am assuming that many inputs will be processed, so that I want the common, preloaded tasks to remain in the preload, and not be replaced by speculative fetching of less common tasks.

This is messy, and any real implementation is likely to only keep an even multiple of $P$ tasks in the preload[3], anyway, for ease of implementation (this assumes tasks are relatively small compared to the preload size).

[3]This assumes equal sized tasks; for unequal size tasks, one might balance the estimated total task time, not simply the total count.

So, assume that $P$ divides $A$, for the following estimate:

$$time^{DMA}_{fixed}(m,n) = \begin{cases} k\lceil \frac{n}{P} \rceil & \text{if } m = 0 \\ k\lceil \frac{A}{P} \rceil + \max\{k, \frac{P}{b}\}\lceil \frac{m}{P} \rceil & \text{if } m > 0 \end{cases}$$

Recalling that $m = (n-A)_+$ and using

$$\min(n,A) = \begin{cases} n & \text{if } m = 0 \\ A & \text{if } m > 0 \end{cases}$$

this can be written

$$time^{DMA}_{fixed}((n-A)_+, n) = k\lceil \frac{\min\{n,A\}}{P} \rceil + \max\{k, \frac{P}{b}\}\lceil \frac{(n-A)_+}{P} \rceil$$

giving

$$\begin{aligned} \langle time^{DMA}_{Fixed} \rangle &= \langle time^{DMA}_{fixed}((n-A)_+, n)\rangle_{\mathcal{N}} \\ &= \frac{k}{P}(\langle \min\{n,A\}\rangle_{\mathcal{N}} + quant_{\mathcal{N}}(\min\{n,A\}, P)) \\ &\quad + \max\{\frac{k}{P}, \frac{1}{b}\}(\langle (n-A)_+\rangle_{\mathcal{N}} + quant_{\mathcal{N}}((n-A)_+, P)) \qquad \text{(B.10)} \end{aligned}$$

I analyze *random* as follows: evaluation of a $P$-block takes time $\max\{k, F/b\}$, where $F$ is a random variable giving the number of fetches from off-chip, *i.e.* the number of preload misses. As tasks are selected independently, we may view the entire set of selected tasks as already constructed, using the $p_i$'s. For these selected tasks, the probability that one chosen at random is not in the preload is $p_A$. We can thus view selecting the next $P$ tasks as a set of $P$ independent trials, with probability of success (not being in the preload) equal to $p_A$. So $F \sim \text{Binomial}(P, p_A)$, and the expected time to evaluate a $P$-block is $\langle \max\{k, f/b\}\rangle_{\mathcal{F}}$.

Let

$$h(k, P, p_A, b) \doteq \frac{1}{P}\langle \max\{k, \frac{f}{b}\}\rangle_{\mathcal{F}} \qquad \text{(B.11)}$$

be the expected time for a single task, then $h(k, P, p_A, b)$ is concave monotone increasing in $p_A$, and has the following properties:

$$h(k, P, p_A, b) = \frac{k}{P} \text{if } k \geq \tfrac{P}{b} \qquad \text{(B.12)}$$

$$\begin{aligned} h(0,P,p_A,b) &= \frac{p_A}{b} && \text{(B.13)}\\ h(k,1,p_A,b) &= k(1-p_A)+\max\{k,\frac{1}{b}\}p_A && \text{(B.14)}\\ h(k,P,1,b) &= \max\{\frac{k}{P},\frac{1}{b}\} && \text{(B.15)}\\ h(k,P,0,b) &= \frac{k}{P} && \text{(B.16)}\\ \max\{\frac{k}{P},\frac{p_A}{b}\} &\le h(k,P,p_A,b) \le \frac{k}{P}(1-p_A)+\frac{p_A}{b} && \text{(B.17)} \end{aligned}$$

the latter bounds becoming tight for $P \gg k$. To evaluate $n$ models requires evaluating $\lfloor \frac{n}{P} \rfloor$ $P$-blocks, then a single ($n$ mod $P$) block. This becomes messy, due to $h()$, so I adopt a slightly more pessimistic model, and say that evaluating $n$ models requires evaluating $\lceil \frac{n}{P} \rceil$ $P$-blocks. This gives

$$\begin{aligned} \langle time^{DMA}_{random}\rangle &= \langle (P\lceil \frac{n}{P}\rceil) h(k,P,p_A,b)\rangle_{\mathcal{N}} \\ &= h(k,P,p_A,b)(\langle n\rangle_{\mathcal{N}} + quant_{\mathcal{N}}(n,P)), && \text{(B.18)} \end{aligned}$$

which is, ignoring quantization terms, the expected time to evaluate a task times the expected number of tasks.

## B.3 Asymptotic behavior of *random*

To get some intuition, I look at the asymptotic behavior of the formulae for *random* , neglecting *quant* terms, which gives

$$\begin{aligned} \langle time^{SIMD}_{random}\rangle &\approx (\tfrac{k}{P}+\tfrac{p_A}{b})\langle n\rangle_{\mathcal{N}} \\ \langle time^{DMA}_{random}\rangle &\approx h(k,P,p_A,b)\langle n\rangle_{\mathcal{N}} \end{aligned}$$

The first equation says that each model takes an average of $\frac{k}{P}$ to evaluate, and that a fraction $p_A$ of them must be fetched from outside the preload, each taking time $\frac{1}{b}$. Using the lower bound for $h(k,P,p_A,b)$ in the second equation gives a RHS of $\max\{\frac{k}{P},\frac{p_A}{b}\}\langle n\rangle_{\mathcal{N}}$ which says that, on average, the time taken by the DMA version is the larger of the evaluation time and the time to load an average number of models from off-chip. Asymptotically, I get:

$k \gg P$ : When task granularity is very large, $h() = \frac{k}{P}$, and $p_A$ is irrelevant, giving perfect parallelism; $\langle time^{SIMD}_{random} \rangle = \langle time^{DMA}_{random} \rangle = \frac{k}{P}\langle n \rangle$

$p_A = 0$ : When everything is found in preload, $h() = \frac{k}{P}$, and again we have perfect parallelism; $\langle time^{SIMD}_{random} \rangle = \langle time^{DMA}_{random} \rangle = \frac{k}{P}\langle n \rangle$

$p_A = 1$ : There is no preload; $h() = \max\{\frac{k}{P}, \frac{1}{b}\}$, and

$$\begin{aligned}
\langle time^{SIMD}_{random} \rangle &= (\tfrac{k}{P} + \tfrac{1}{b})\langle n \rangle \\
\langle time^{DMA}_{random} \rangle &= \tfrac{k}{P}\langle n \rangle \qquad (k \geq \tfrac{P}{b}) \\
\langle time^{DMA}_{random} \rangle &= \tfrac{1}{b}\langle n \rangle \qquad (k \leq \tfrac{P}{b})
\end{aligned}$$

The SIMD equation corresponds to always loading $P$ models and then evaluating them. The first DMA equation occurs when evaluation takes longer than fetching, and corresponds to all fetches being 'hidden' by the evaluations. The second DMA equation is the converse, where all evaluation time is 'hidden' by fetch time.

$k \ll P$ : For very small tasks, $h() \approx \max\{\frac{k}{P}, \frac{p_A}{b}\}$, and

$$\begin{aligned}
\langle time^{SIMD}_{random} \rangle &= (\tfrac{k}{P} + \tfrac{p_A}{b})\langle n \rangle \\
\langle time^{DMA}_{random} \rangle &= \max[\tfrac{k}{P}, \tfrac{p_A}{b}]\langle n \rangle
\end{aligned}$$

Note that $\langle time \rangle \rightarrow 0$ as $k \rightarrow 0$ and $p_A \rightarrow 0$, i.e. when no fetches need be done, and evaluations take no time. However, $k$ is really limited away from 0 as it includes the load time from local memory.

$P = 1, k < \frac{1}{b}$ : For a single processor with local memory, doing small tasks; $h() = k(1 - p_A) + \frac{p_A}{b}$, and

$$\begin{aligned}
\langle time^{SIMD}_{random} \rangle &= (k + \tfrac{p_A}{b})\langle n \rangle \\
\langle time^{DMA}_{random} \rangle &= (k(1 - p_A) + \tfrac{p_A}{b})\langle n \rangle
\end{aligned}$$

Here DMA reduces average evaluation time by hiding it whenever an off-chip fetch is done.

## B.4 Task Distributions

For speedup comparisons with sequential architectures I must make assumptions about the task distribution, $p_i$. For both *random* and *fixed*, we may index the tasks so that $i < j \Rightarrow p_I(i) \geq p_j$. Realistic extreme cases for $p_i$ with respect to weighting of lower indexed tasks, are the hyperbolic and uniform distributions. The uniform distribution puts as little probability mass in the preload as possible, while the hyperbolic distribution puts a great deal. Besides illustrating situations where models are highly likely to be found in the preload, the hyperbolic distribution is realistic for word model matching, corresponding to Zipf's Law of work frequency distributions [Zip32] [4] For *random*, I also vary $p_0$, determining the expected fraction of tasks selected, $\langle n\rangle^{rand} \approx (N+1)(1-p_0)$, independently of the probability mass of the preload.

If there are $N$ total possible tasks, the uniform distribution gives mass $1/N$ to each task, while the hyperbolic distribution gives the $i$'th task mass $1/(iZ(N))$, where $Z(N) = \sum_1^N \frac{1}{i} = C + \log N + \frac{1}{2N} - O(N^{-2})$ and $C \approx 0.577$ is Euler's constant. We have

$$\langle n\rangle_{uniform} = \frac{N+1}{2} \tag{B.19}$$

$$\langle n\rangle_{hyperbolic} = \frac{N}{Z(N)} \tag{B.20}$$

$$\langle n\rangle^{random} = (N+1)(1-p_0) \tag{B.21}$$

$$\begin{aligned} p_A^{uniform} &= 1 - \sum_{i=1}^{A} \frac{1}{n} \qquad \text{(B.22)} \\ &= 1 - \frac{A}{N} \end{aligned}$$

$$p_A^{hyperbolic} = 1 - \sum_{i=1}^{A} \frac{1}{iZ(N)} \tag{B.23}$$

[4] Actually, Zipf suggests a power law, $Prob(n) = (1/Z)n^r$, where $n$ indexes the words in the vocabulary, $n = 1$ corresponding to the most common word, $n = 2$ to the second most common, and so on; Z is a normalizing constant; and $r$ is a constant, $r > 1$. The hyperbolic distribution corresponds to $r = 1$. However, power laws, while having the correct qualitative properties, do not fit the observed distribution well, and other distributional forms have been suggested [Sic75]. For my purposes, the relevant property shared by all these distributions is that of putting a great deal of probability mass in the most common words (models). So the use of the hyperbolic distribution can be justified as a convenient example of such a distribution.

$$= \quad 1 - \frac{Z(A)}{Z(N)}$$

and one can derive the following approximations, used in the simulations:

$$\langle (n-A)_+ \rangle_{uniform} \approx \langle n \rangle_{unif} * (1-a)^2 \tag{B.24}$$

$$\langle (n-A)_+ \rangle_{hyperbolic} \approx \langle n \rangle_{hyperbolic} * (1 - a + a \log a) \tag{B.25}$$

where $a = A/N \in [0,1]$ gives the fraction of models contained in the preload, $a > 0$ as we are assuming $A \geq P$, and I adopt the usual convention that $a \log a = 0$ when $a = 0$, i.e., when $A = 0$. The error in the first approximation is $\frac{a(1-a)}{2} \in [0, 1/2]$, the error in the second is $\frac{a-1}{2Z(N)} + O(\frac{1}{AZ(N)}) \in [-1/2, 0] + O(\frac{1}{AZ(N)})$.

We can also approximate $quant_{\mathcal{N}}(\cdot, P) = \langle P\lceil \frac{n}{P} \rceil \rangle_{\mathcal{N}}$. Let $p_{\mathcal{N}}(n)$ be the density for $n$, so $p_{\mathcal{N}}(n) = p_I(n)$ for *fixed* and $p_{\mathcal{N}}(n) = Binomial(N+1, 1-p_0)$ for *random*. Writing out the expectations for the terms in the various formulae,

$$\begin{aligned} quant_{\mathcal{N}}(n, P) &= \sum_{n=1}^{N} p_{\mathcal{N}}(n)(P\lceil \frac{n}{P} \rceil - n) \\ &\approx \frac{P-1}{2} \end{aligned} \tag{B.26}$$

$$\begin{aligned} quant_{\mathcal{N}}(\min\{n, A\}, P) &= \sum_{n=1}^{N} p_{\mathcal{N}}(n)(P\lceil \frac{\min\{n, A\}}{P} \rceil - \min\{n, A\}) \\ &= \sum_{n=1}^{A} p_{\mathcal{N}}(n)(P\lceil \frac{n}{P} \rceil - n) + \sum_{n=A+1}^{N} p_{\mathcal{N}}(n)(P\lceil \frac{A}{P} \rceil - A) \\ &\approx \frac{P-1}{2} p_A + (A \bmod P)(1 - p_A) \\ &\approx \frac{P-1}{2} p_A \end{aligned} \tag{B.27}$$

$$\begin{aligned} quant_{\mathcal{N}}((n-A)_+, P) &= \sum_{n=1}^{N} p_{\mathcal{N}}(n)(P\lceil \frac{(n-A)_+}{P} \rceil - (n-A)_+) \\ &= \sum_{n=A+1}^{N} p_{\mathcal{N}}(n)(P\lceil \frac{n-A}{P} \rceil - (n-A)) \\ &\approx \frac{P-1}{2}(1 - p_A) \end{aligned} \tag{B.28}$$

where the approximations are made by

$$\sum_{x=1}^{N} p_{\mathcal{X}}(x)(P\lceil\frac{x}{P}\rceil - x) \approx (P-1)/2,$$

i.e. assuming that the support of the density is wide, smooth and unrelated to $P$, and that $N$, $N-A$ and $A$ are large relative to $P$. This holds for the distributions I will consider. Note also that formula (B.10) using $quant_{\mathcal{N}}(\min\{n, A\})$ already assumes $(A \bmod P) = 0$. For simulations, I will use a parameter $Q \in \{0, \frac{1}{2}, 1\}$ to give best,worst and "normal" case performance, replacing $(P-1)/2$ in the above approximations by $Q(P-1)$.

## B.5 Area Tradeoffs

To look at tradeoffs, note that finite chip area implies that the amount of local memory, and hence $a$ and $p_A$, decreases as the number of PE's increases. From equations 4.1 one gets

$$P = \frac{800s}{c + z \cdot M} \tag{B.29}$$

where $c$ gives the area of the data path of a single PE, in $10^6\lambda^2$, $s$ gives the total chip area, in $10^9\lambda^2$, $z$ gives the size of 1 KB of memory, in $10^6\lambda^2$, and $M$ gives the size of a PE's local memory, in kB. Some typical values of $c$ are $c = 30$ for a simple fixed point processor, $c = 70$ for a simple 32b floating-point processor, and $c = 500$ for a complex floating point processor, on the order of a PPC 604. To get estimates biased towards large numbers of PE's per chip, I use the numbers for single-ported SRAM, with 1 KB of SRAM taking $z = 5 \times 10^6\lambda^2$. From table 4.3.1, $s \in \{19, 42, 68, 101, 208, 506, 1200\}$ for the next 7 process generations. We can thus remove $M$ as a free parameter:

$$M = \frac{800s/z}{P} - \frac{c}{z} \tag{B.30}$$

The total on-chip memory is $PM$. I introduce another task parameter, $d$, giving the number of models storable in one kB. $d$ can be used to replace $A$, the number of models in the preload, thus incorporating area constraints: $A = dPM$ if all on-chip memory were used for the preload. For use later, I introduce a parameter, $i_m$, measuring the amount of per-PE instruction memory, in kB:

$$A = d(PM - Pi_m). \tag{B.31}$$

One can then replace the $p_A$ parameter: $p_A = 1 - \sum_1^A p_I(i)$ is $1 - A/N$ in the uniform case, and $1 - Z(A)/Z(N)$ in the hyperbolic case.

Two important special cases are when all tasks can fit in the preload, $p_A = 0$, and when none can, $p_A = 1$. The latter situation obtains when a single task exceeds the size of an individual PE's local memory, in which case my modeling assumptions and consequent speedup equations are incorrect. Define $P_{all}$ to be the maximal $P$ for which $p_A = 0$ (that is, $A = N$), and $P_{none}$ to be the maximal $P$ for which $p_A < 1$. $P_{none}$ then corresponds to 1 task per PE, or $A = P$. Using equations B.29, B.30, and B.31, we have

$$P_{all} = \frac{(800s/z) - N/d}{(c/z) + i_m} \tag{B.32}$$

and

$$P_{none} = \frac{800s/z}{(c/z) + i_m + 1/d} \tag{B.33}$$

## B.6 Speedup: some simplifications

It should be clear that the quantization term $quant_{\mathcal{N}}(\cdot, P) \approx (P-1)/2$ (equations B.26 – B.28) will be negligible compared to $\langle n \rangle$ for most model distributions. If we neglect quantization, then two regimes present themselves with simple formulae for speedup: $P \leq P_{all}$ (all models in preload) and $P \geq P_{none}$ (no models in preload). In these regimes the formulae for the various task distributions are the same, as are the times for both the SIMD and DMA cases. For either form of speedup[5], $\langle time^{SEQ-DMA} \rangle / \langle time_*^{DMA} \rangle$, we have

$$speedup = \ P \max\{\tfrac{1}{q}, \tfrac{1}{k}\} \quad \text{if } P \leq P_{all} \tag{B.34}$$

[5] I ignore the $\langle time^{SEQ} \rangle$ case as it is unrealistic these days. Similar equations hold for $\langle time^{SEQ-DMA} \rangle / \langle time^{SIMD} \rangle$.

and

$$speedup \approx \frac{\min\{P,kb\}}{\min\{k,q\}} \quad \text{if } P = P_{none}. \tag{B.35}$$

where the latter equation is derived using $A = 0$ rather than $A = P$, which will be close if $\langle n \rangle \gg P$. These equations show that speedup is linear for $P \leq P_{all}$.

For a particular architectural setup, defining $s$, $c$, and $m$, equation B.32 is linear in $N/d$, the size in kB of the set of all tasks. Figure B.1 illustrates this for some reasonable values of $s$, $c$, and $m$, using an SRAM size of $5 \times 10^6 \lambda^2$. For example, a current generation ($s = 19$) chip with 16 simple floating point processors ($c = 70$) can preload an entire task set of about 2.5 megabyte (B.1 (c)), and this increases to about 6.5 megabytes in the next generation ($s = 42$) (B.1 (g)). For a current generation chip with more complex processors ($c = 500$), a task set of about 1.5 - 2.0 MB can be preloaded. Similar calculations assuming an SRAM size of $5 \times 10^6 \lambda^2$ show that, in that case, using complex processors ($c = 500$) is not feasible until the next generation ($s = 42$). When simple ($c = 70$) processors are used, a current generation chip with 16 PEs can preload about 1 MB, increasing to 2-3 MB in the next generation.

Similarly, figure B.2 shows some graphs of $P_{none}$ from equation B.33. Both figure B.2 and examination of equation B.33 show that $P_{none}$ will be substantially greater than $P \approx 16 - 32$ for non-minimal sized chips with simple floating point processors (B.2 (a),(c)), and hence that it will be possible to preload a substantial number of tasks in this case. However, preloading a large number of tasks will not be possible for a current or next generation chip with 32 very complex processors (B.2 (b),(d)), especially when processors have substantial amounts of instruction memory. For further generations ($s \geq 42$), substantial preloading will be possible, even with very complex processors (not shown).

## B.7 Simulations

In the regime $P_{all} \leq P \leq P_{none}$ the mutual effects of the various parameters are more complicated, and so they were studied via simulation.

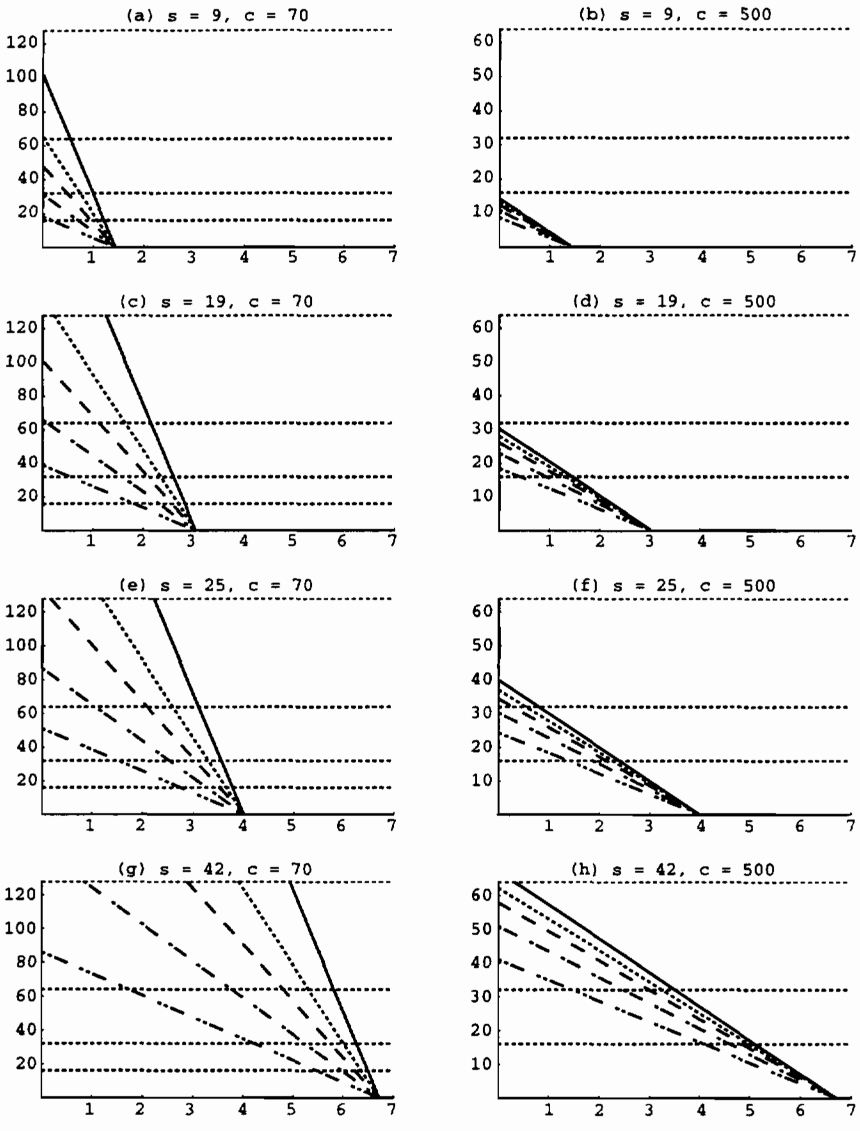


Figure B.1: The maximum number of processors for which the entire task set can be preloaded, $P_{all}$, as a function of the total task set (modelbase) size in MB, $(10^{-3}N/d)$. The horizontal axis is the task set size in MB, and the vertical axis is the maximal number of processors. The different lines on each set of axes correspond to different amounts of instruction memory, $m \in \{0, 4, 8, 16, 32\}$, with $m = 0$ the topmost line, and $m = 32$ the bottommost. Horizontal lines are drawn at $P_{all} \in \{16, 32, 64, 128\}$ for comparison purposes.

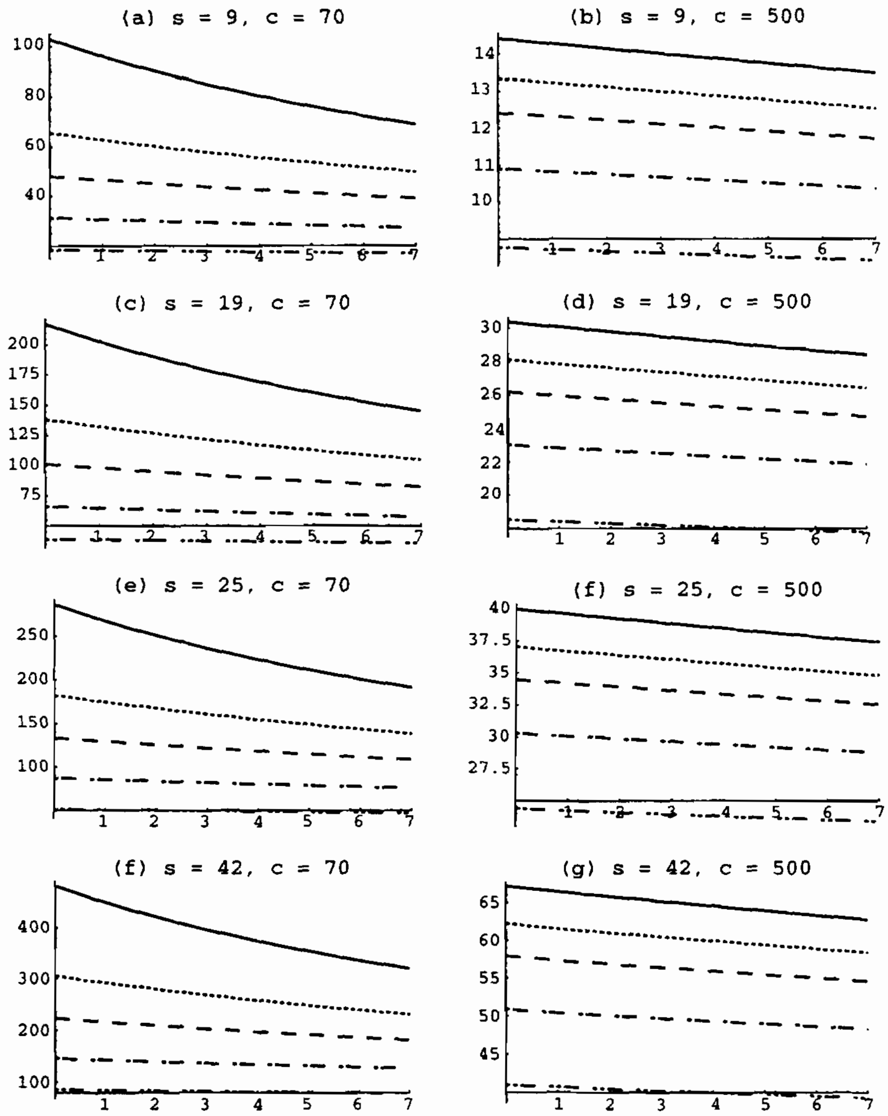


Figure B.2: The maximal number of processors for which a processor can contain an entire task, $P_{none}$, as a function of $(10^{-3}N/d)$, the task set (modelbase) size in MB. The horizontal axis is the task set size in MB, and the vertical axis is the maximal number of processors. The different lines on each set of axes correspond to different amounts of instruction memory, $m \in \{0, 4, 8, 16, 32\}$, with $m = 0$ the topmost line, and $m = 32$ the bottommost. Horizontal lines are drawn at $P_{all} \in \{16, 32, 64, 128\}$ for comparison purposes.

Table B.1: Free parameters of the analysis, their types and constraints

| name | description | type | bounds |
|---|---|---|---|
| $P$ | number of processors | architectural | $1 \leq P$ |
| $q$ | comparative speed of sequential | architectural | $1 \leq q \leq P$ |
| $b$ | comparative memory bandwidth | architectural | $1 \leq b \leq P$ |
| $s$ | chip area ($10^9\lambda^2$) | architectural | $0 < s$ |
| $i_m$ | instruction memory (kB) | architectural | $0 \leq m$ |
| $c$ | size of PE datapath ($10^6\lambda^2$) | architectural | $30 \leq c \leq 500$ |
| $k$ | speed of evaluating a model | algorithmic | $1 \leq k$ |
| $d$ | size of a task (tasks/kB) | application-specific | $0 < d$ |
| $N$ | total number of tasks | application-specific | $P \leq N$ |
| $p_0$ | $\langle n \rangle_{rand} = (N+1)(1-p_0)$ | application-specific | $0 \leq p_0 \leq 1$ |
| $Q$ | best/avg/worst quantization effects | simulation | $0 \leq Q \leq 1$ |

Table B.1 lists the parameters I have introduced, together with constraints or reasonable bounds on their values.

Panel (a) of figure B.3 shows the general shape of the speedup curves in the interval between $P_{all}$ and $P_{none}$. The topmost two curves show the lower and upper bounds for the *random-hyperbolic* case, the two lowest curves show the bounds for the *random-uniform* case, and the intermediate two curves show the *fixed-uniform* and *fixed-hyperbolic* cases, the latter being topmost. This ordering is preserved in all subsequent speedup figures, as is the association between model and the dashing used in the curve for that model. In subsequent speedup figures we see the same relationships, but the curves are distorted by (1) different values of $P_{all}$ and $P_{none}$ and (2) the different speedups at $P_{all}$ and $P_{none}$ The speedup at $P_{all}$ is approximately $P_{all}$, due to the linear growth for $P \in [1, P_{all}]$; the speedup for $P_{none}$ is given by equation B.35, but is usually $k$. For clarity, the value at $P_{none}$ has been continued for $P > P_{none}$ although I do not actually model speedup in that regime.

Two features of the figure are of note: there is a nearly linear increase in speedup from $P = 1$ to $P_{all}$ and there is an abrupt "cutoff" in the curves at $P = P_{none}$. The first effect holds for all curves for which $P_{all} = 0$, and is a simple consequence of the models: a speedup of $k \leq P$ is always possible as that much work can be done while the next model is being loaded. The second, "cutoff", effect is due to the quantization implied by requiring $A = 0 \pmod{P}$. To make the curves easier to read, this assumption has been relaxed in the simulations by allowing fractional numbers of models to be preloaded. Panel (b) of the figure shows the effect of enforcing integral numbers of models. The behavior is quite similar, except for the large speedup *random-hyperbolic* case and for $P \approx P_{none}$. Subsequent figures do not show quantization effects except for the "cutoff" effect at $P_{none}$.

Figure B.4 shows the speedup curves for the "default" configuration of a current generation chip ($s = 19$) with simple floating point processors ($c = 70$), using a set of $N = 1000$ tasks. We see for large tasks of 5 KB each ($d = 0.2$) that one never has all tasks in the preload. The consequence is that the computation is I/O bound and the speedup curves are relatively flat. The exception is the *random-hyperbolic* case, where the likelihood of finding the desired task preloaded is high even when few tasks fit in the preload. The figure also shows that, for this configuration, $P = 16$ or $P = 32$ are reasonably close to the optimal $P$, especially for the smaller $k$, but that a wide range of values give similar speedup, all close to $k$ (for $P \geq k$).

For smaller tasks of 1 KB each ($d = 1$), there is a linear regime up to $P = P_{all} \approx 150$, followed by a reversal and decay to the $P = k$ asymptote that is more or less rapid, depending on $k$. For large $k$, values of $P$ above $P_{all}$ provide only a modest improvement in performance (except for the upper bound for the *random-hyperbolic* case). Essentially, for the $P_{all} = 0$ ($d = 0.2$) case, $P = 16$ performs about as well as the optimum $P$; for the $P_{all} > 0$ ($d = 1$) case, $P = P_{all}$ is close to optimal. The exception is for the *random-hyperbolic* model, where the effective size of the task set is much smaller.

Figure B.5 shows the effect of large $k$, the amount of computation per model. A larger $k$ naturally decreases the effect of bandwidth limitations and allows more parallelism. In terms of the graphs, the effect is to meliorate the rapid loss of parallelism as $P$ increases from $P_{all}$ to $P_{none}$.

Using the information in table 4.8, $s = 9$ corresponds, for any generation, to the smallest chip capable of 16 processors, data I/0 pins, but sharing instruction pins. Figure

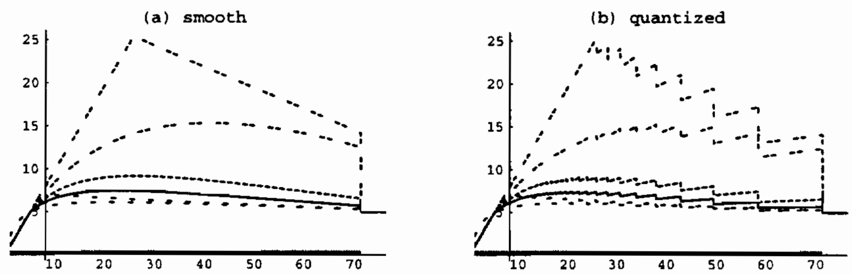


Figure B.3: Panel (a) shows the general shape of the speedup curves in the $[P_{all}, P_{none}]$ interval. The horizontal axis is number of processors, $P$, and the vertical axis is speedup. Speedup is defined as $t^{SEQ-DMA}/t_*^{DMA}$. The topmost two curves show the lower and upper bounds for the *random-hyperbolic* case, the two lowest curves show the bounds for the *random-uniform* case, and the intermediate two curves show the *fixed-uniform* and *fixed-hyperbolic* cases, the latter being topmost. Panel (b) shows the effect of enforcing integral numbers of models ($A = 0 \pmod{P}$). Note the "cutoff" effect at $P = P_{none}$. The line on the x-axis marks the interval $[P_{all}, P_{none}]$, the range of $P$ where some, but not all, tasks can be preloaded.

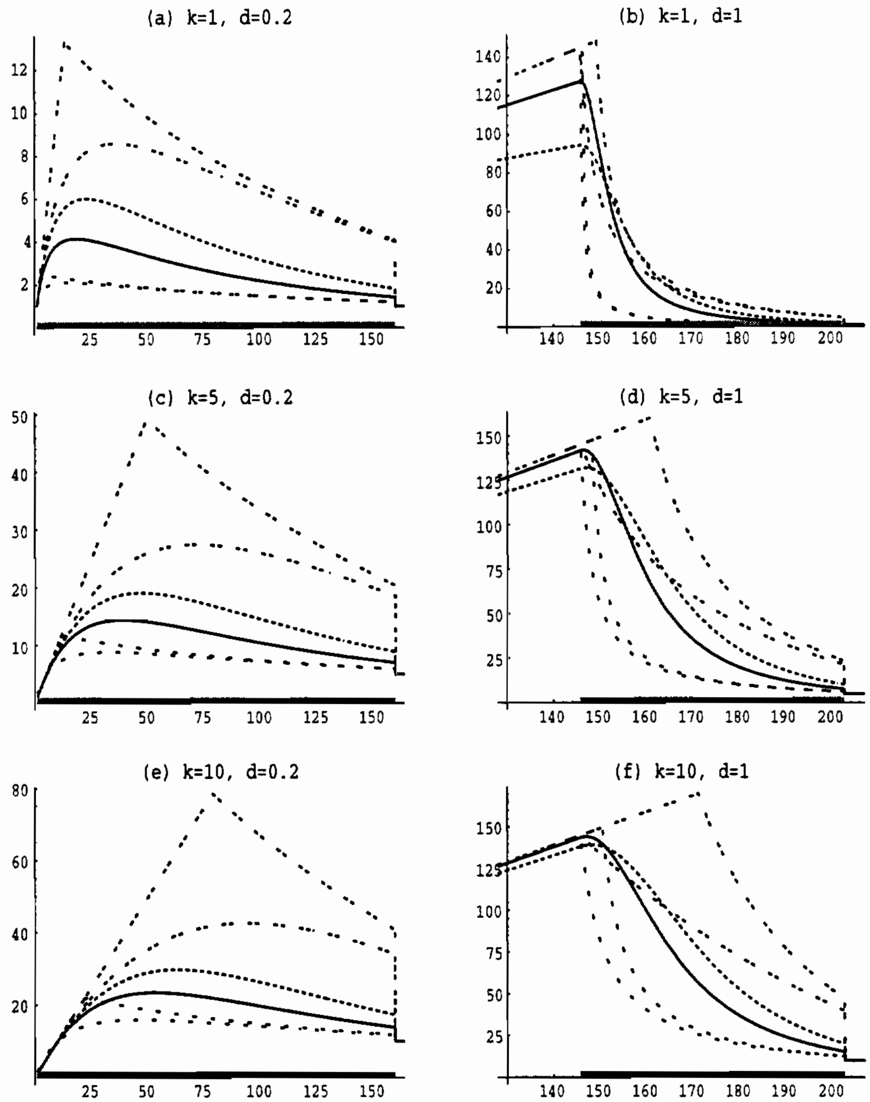


Figure B.4: Speedup ($y$-axis) as a function of the number of processors ($x$-axis, for varying $(k, d)$ and other parameters given their default values: $N = 1000$, $s = 19$, $b = 1$, $m = 0$, $c = 70$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. This set of values corresponds to a current generation chip ($s = 19$) with simple floating point processors ($c = 70$). The first column (a,c,e) shows potential parallelism is limited ($P = 16 - 32$) for larger model-bases (5 MB) that do not fit entirely on-chip. The second column (b,d,f) shows large potential parallelism ($P \geq 128$) for smaller model-bases (1 MB). It also shows that this potential parallelism quickly disappears when the model-base no longer fits on-chip, due to use of chip area for additional data-paths ($P > 150$).

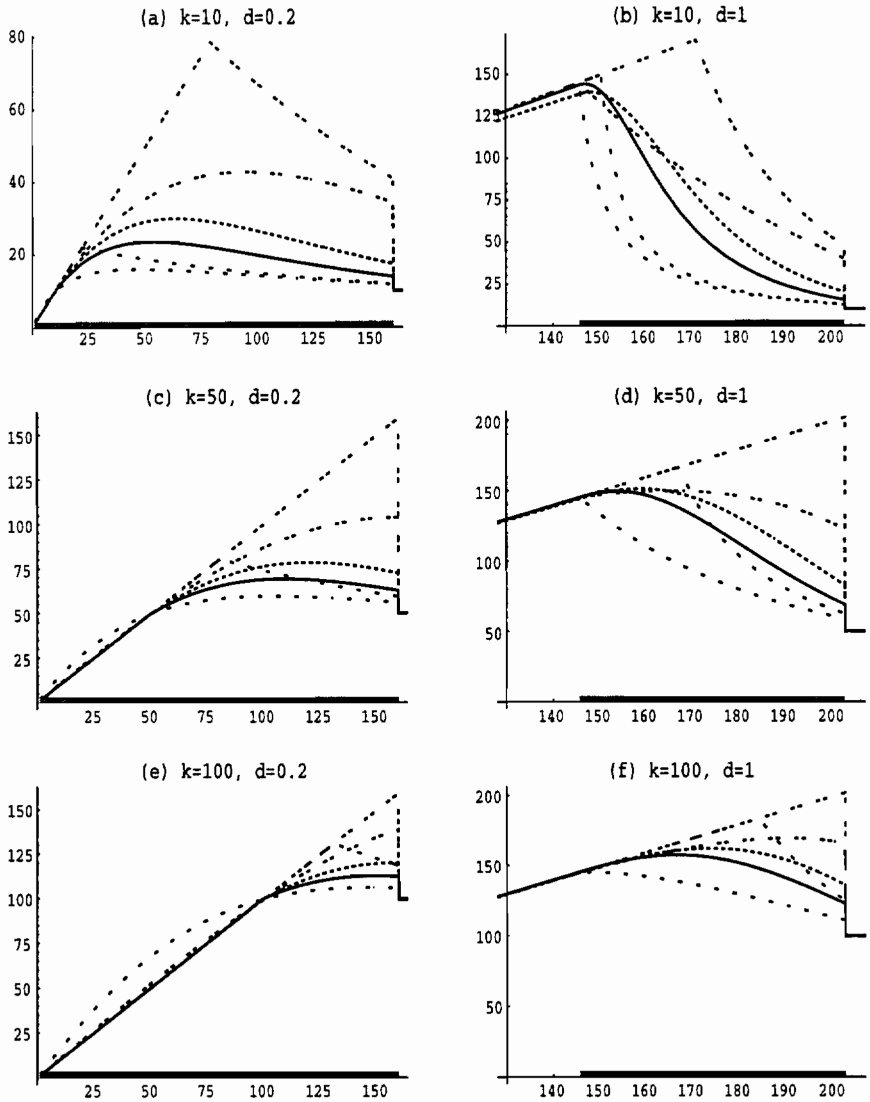


Figure B.5: Speedup versus number of processors, for varying $(k, d)$ and other parameters given their default values: $N = 1000$, $s = 19$, $b = 1$, $m = 0$, $c = 70$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. Compared to figure B.4, larger values of the computation/model ratio, $k$, are used.

B.6 shows that $s = 9$ needs $k > 10$ for effective parallelism with the large test set, even using small floating point processors ($c = 70$). Again, $P = 32$ is close to optimal for random-hyperbolic, otherwise $P = 16$ is as good as any. For comparison, figure B.7 shows the speedup curves for the same chip, under the assumption that the individual data I/O provides a twofold improvement in bandwidth. We see that the doubling of bandwidth essentially doubles the speedup for the large task set, and, for more work-intensive tasks ($k = 5, 10$), substantially increases the usable parallelism. For the smaller task set, doubling the bandwidth substantially increases available parallelism in the regime between $P_{all}$ and $P_{none}$.

Figures B.8 gives analogous results for the smallest chip capable of 32 processors, using the information in table 4.9 and assuming complex processors ($c = 500$).

Figures B.9 and B.10 examine the current and next generations ($s = 19$ and $s = 42$), assuming very complex processors ($c = 500$). We see that $P = 16$ is a reasonable choice for the current generation; while $P = 32$ or $P = 48$ are reasonable for the next one.

The choice of $N = 1000$ is arbitrary, and is intended mainly to reflect the notion of a large task set (large task set). For $N = 1000$, the task set size is 1 MB for $d = 1$, and 5 MB for $d = 5$. Figure B.11 looks at the case $N = 100$. For this task set smaller by an order of magnitude, we see that $P_{all}$ has become much larger, so that $P = 128$ and even $P = 192$ give good performance. Of course, as processors do not evaluate fractional models, $P = N = 100$ is the real maximum. Similarly, figure B.12 illustrates the case where the task set is made smaller by making the individual tasks smaller ($d = 5$, corresponding to 200 bytes per task), as might be seen in nearest neighbor search (where, for example, a model might be a reference vector of 50 32-bit floating point numbers). The problem, of course, is that these larger values of $P$ are very poor in the large task set case, while smaller values of $P$ give linear speedup even in the small task set case.

Finally, figure B.13 shows how available parallelism diminishes with increasing per-PE instruction memory, for 16 and 32 processor chips, and 1 MB task set size. We see that a minimal current generation 16 processor chip ($s = 9, c = 70$) can have around

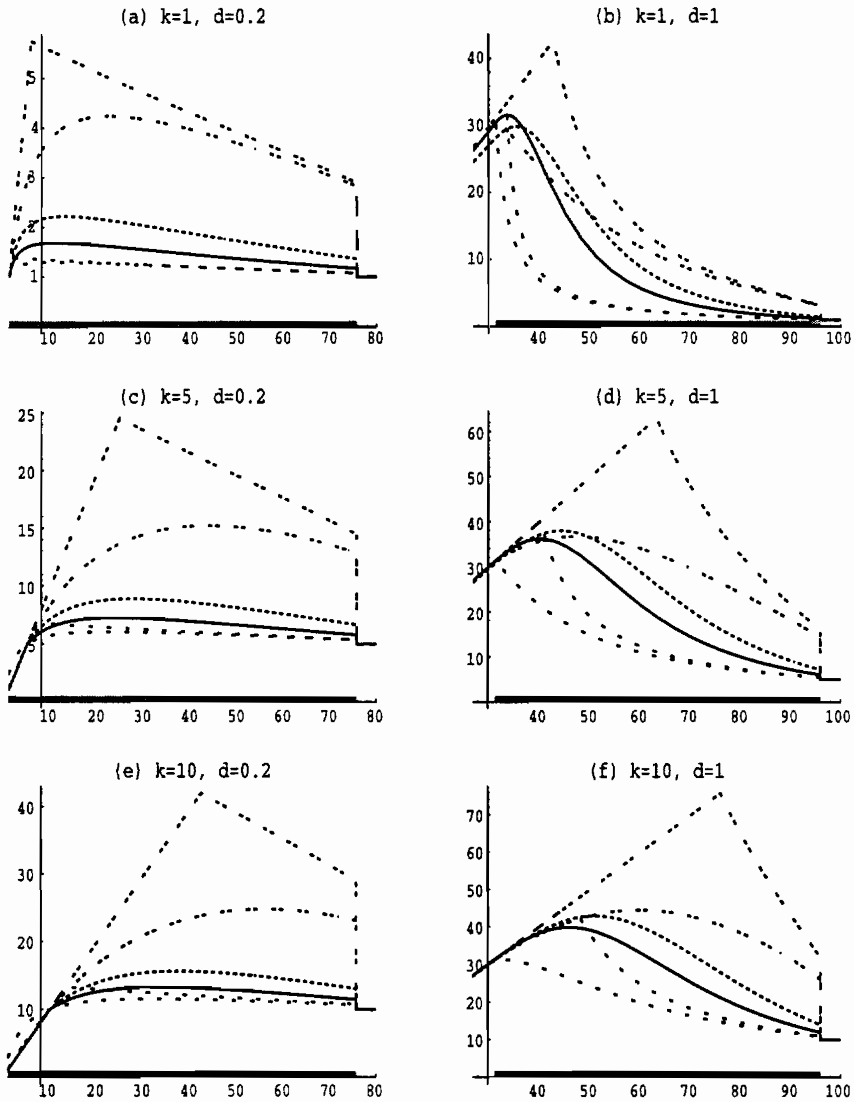


Figure B.6: Speedup versus number of processors, for $s = 9$, varying $(k, d)$. The case examined here, $s = 9$, is that of the smallest chip (in any generation) having 700 pins, and so being capable of having 16 PEs, each with their own off-chip memory. Chip is assumed to have small floating-point processors ($c = 70$), with other parameters given their default values: $N = 1000$, $m = 0$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. We see that the larger task set (a,c,e) needs about $k = 10$ to effectively use 16 processors, while the smaller task set (b,d,f) could utilize 32.

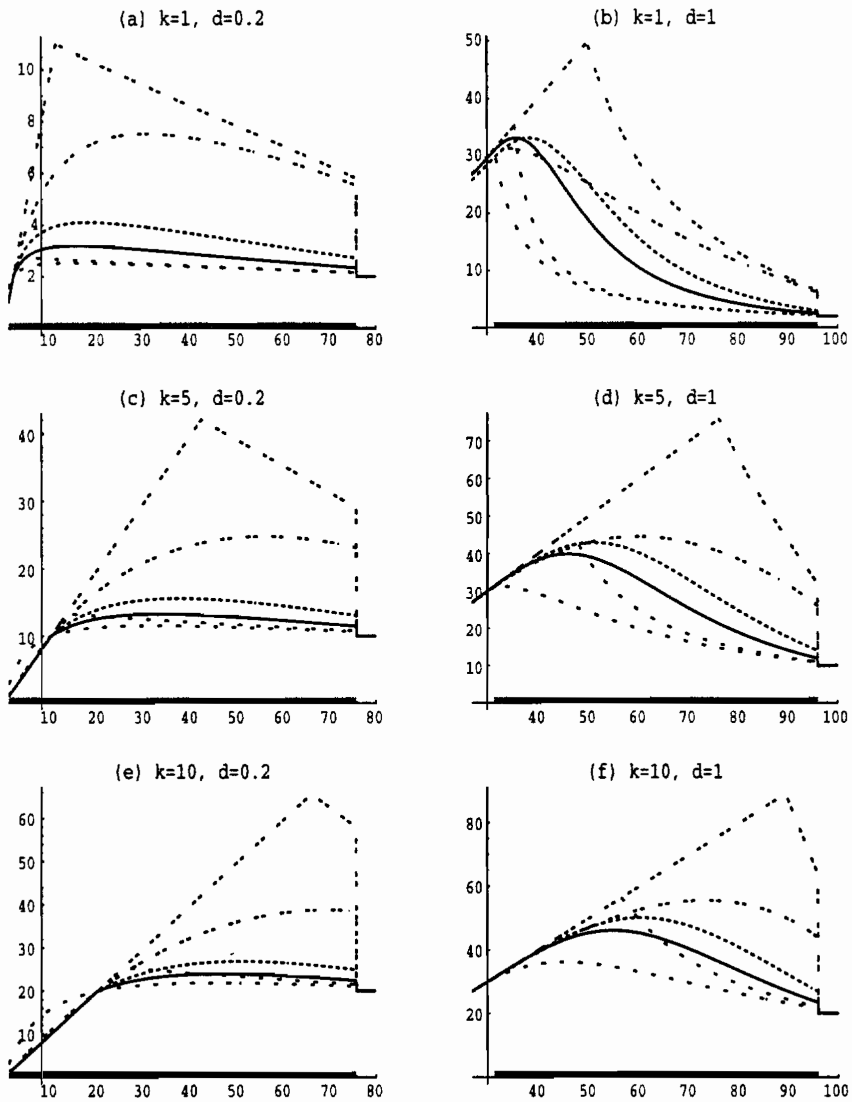


Figure B.7: Speedup versus number of processors, for $s = 9$ and $c = 70$, varying $(k, d)$. The case examined here is the same as that of figure B.6, except that use of per-PE data interfaces is assumed to give a twofold bandwidth improvement over the sequential processor ($b = 2$). This provides a modest improvement for the smaller task set (b,d,f), but, for the larger task set (a,c,e), allows effective use of 16 processors at $k = 5$, i.e., with half the compute/load time ratio.

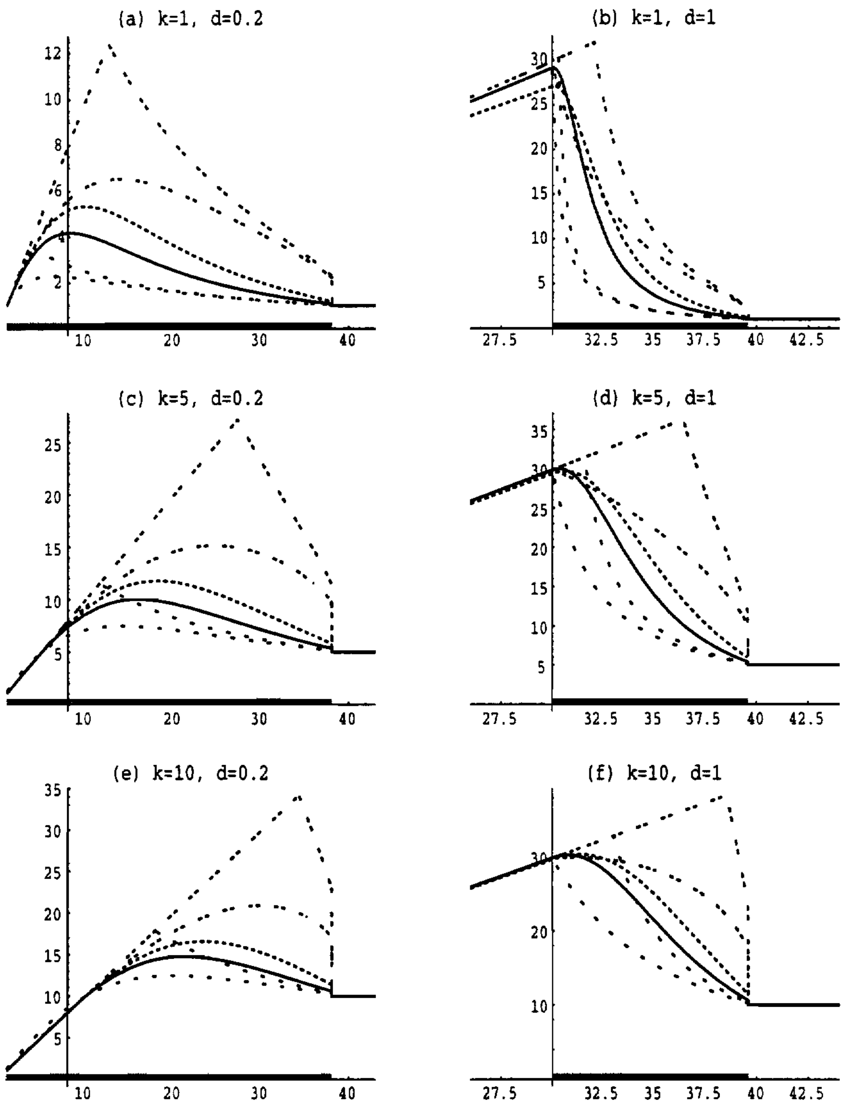


Figure B.8: The case examined here, $s = 25$, is that of the smallest chip (in any generation) having 1200 pins, and so being capable of having 32 PEs, each with their own off-chip memory . The chip is assumed to have complex processors ($c = 500$), with other parameters given their default values: $N = 1000$, $m = 0$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. We see the effects of limited on-chip memory caused by the the large size of the processors. With the larger task set, not even $P = 16$ processors can be effectively used.

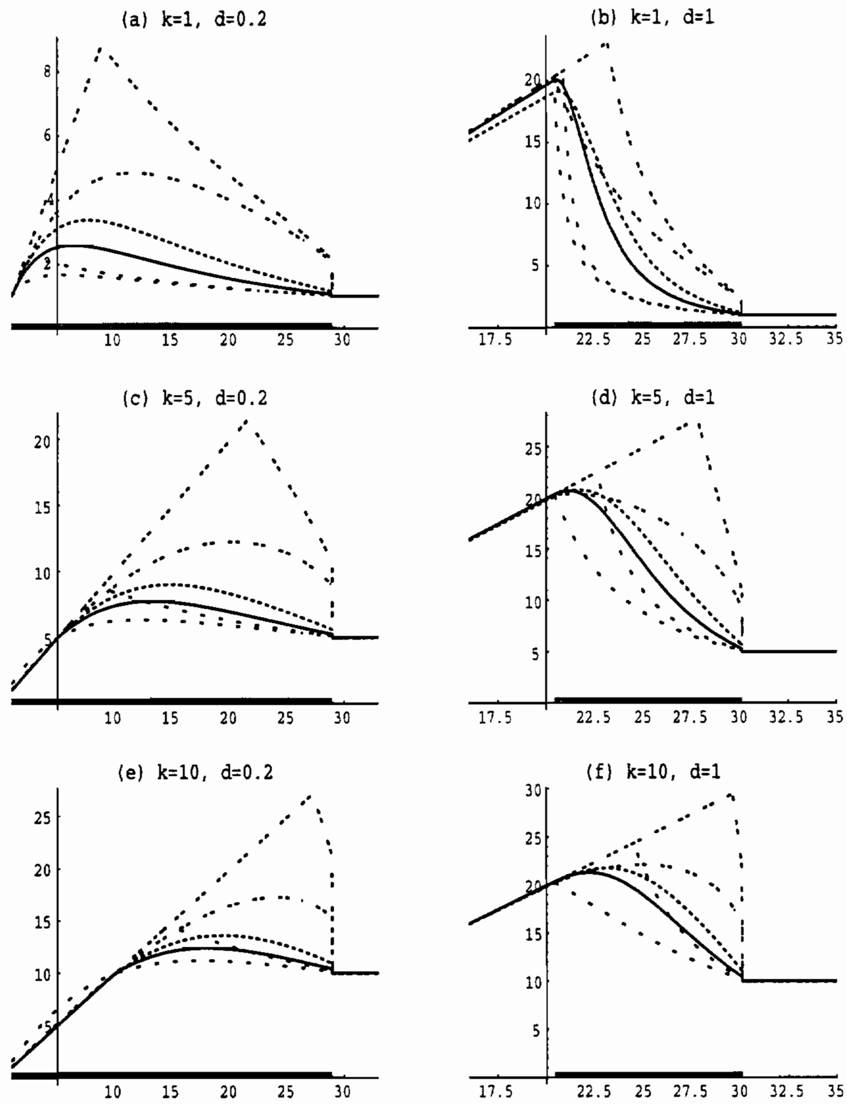


Figure B.9: Speedup versus number of processors, for $s = 19$ and $c = 500$, varying $(k, d)$. The case examined here is that of a large current generation chip with complex PEs. Other parameters given their default values: $N = 1000$, $b = 1$, $m = 0$, $c = 70$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. The graphs suggest that $P = 16$ processors is about optimal. For the larger task set, $k \approx 10$ is needed to effectively use 16 processors.

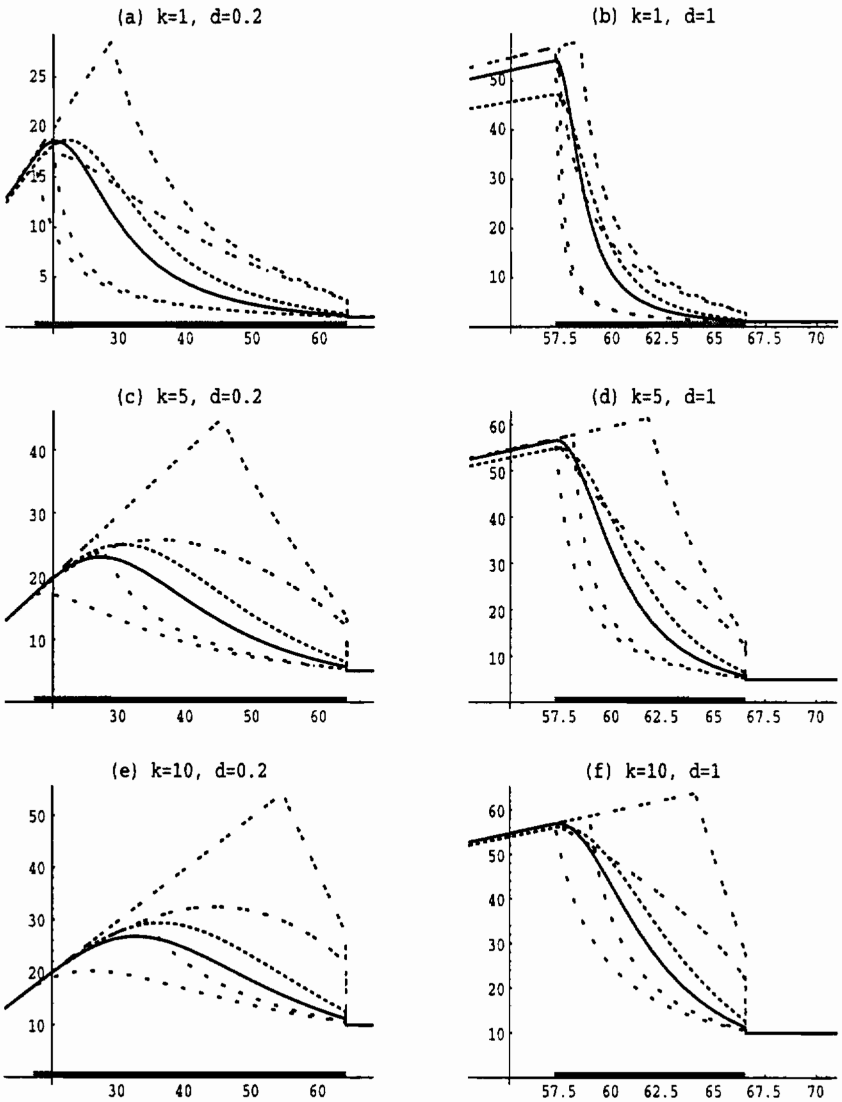


Figure B.10: Speedup versus number of processors, for $s = 42$, varying $(k, d)$. The case examined here, $s = 42$, is that of a next generation chip ($0.18\mu$), with very complex processors ($c = 500$). Other parameters given their default values: $N = 1000$, $b = 1$, $m = 0$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$. We see that $P = 32 - 48$ is reasonable except for the larger task set, and the smallest amount of work per model (panel (a)).

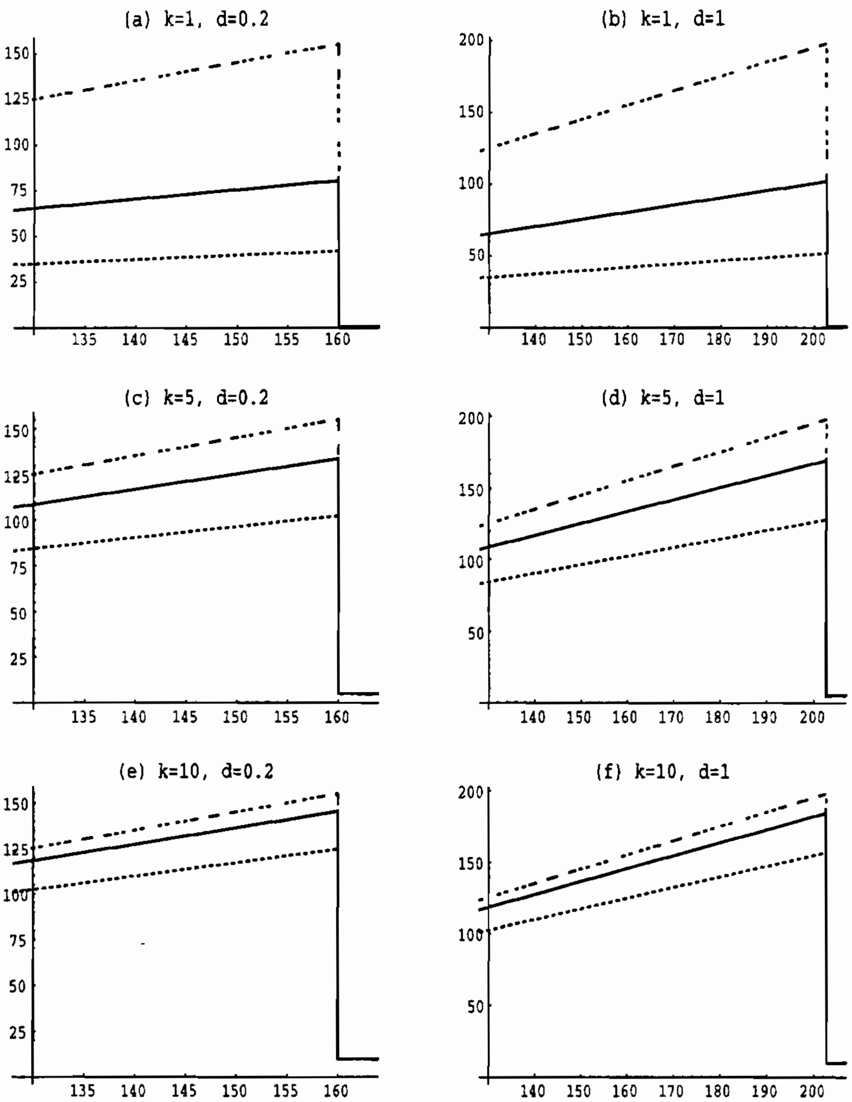


Figure B.11: Speedup versus number of processors, for $s = 19$, varying $(k, d)$. The case examined here is that of the current generation chip, with an order of magnitude fewer models in the model base (100 rather than 1000). Other parameters given their default values: $c = 70$, $b = 1$, $m = 0$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$.

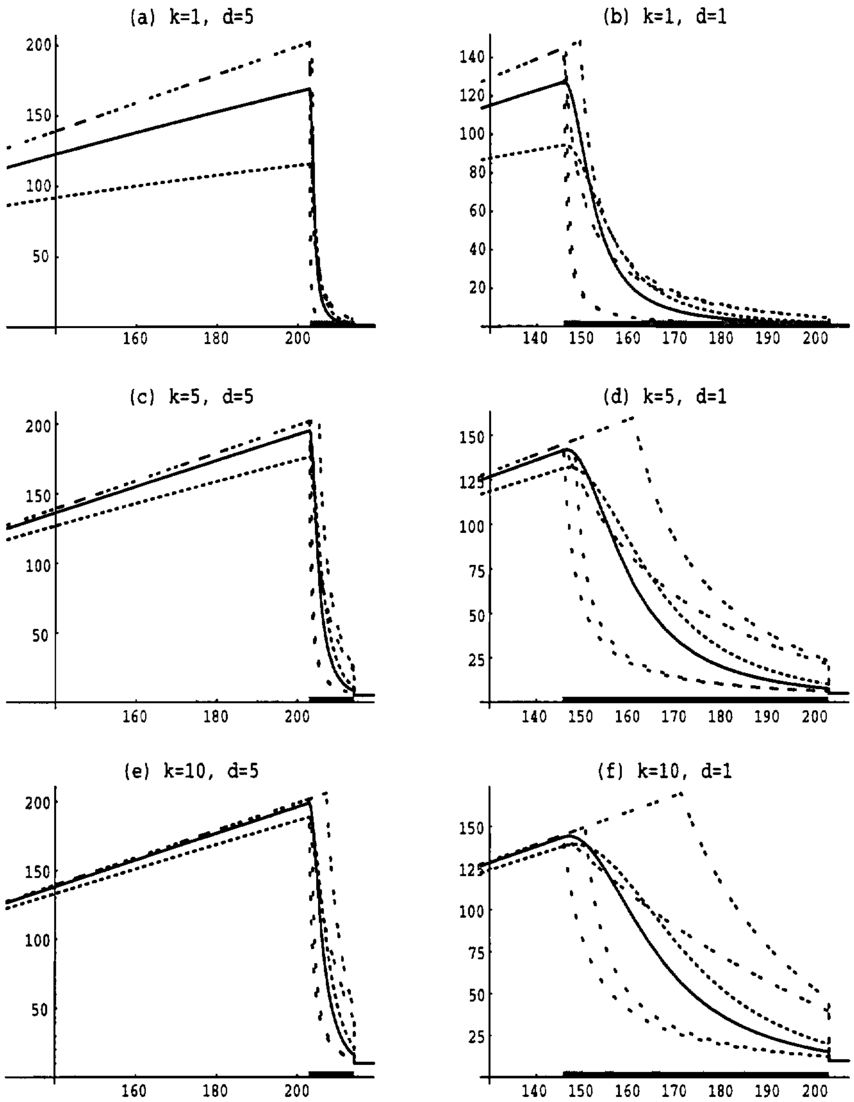


Figure B.12: Speedup versus number of processors, for $s = 19$, varying $(k, d)$. The case examined here is that of the current generation chip, with small models ($d = 5$, i.e., 200 bytes per model). Other parameters given their default values: $c = 70$, $b = 1$, $m = 0$, $p_0 = 0.9$, $q = 1$, $Q = 1/2$.

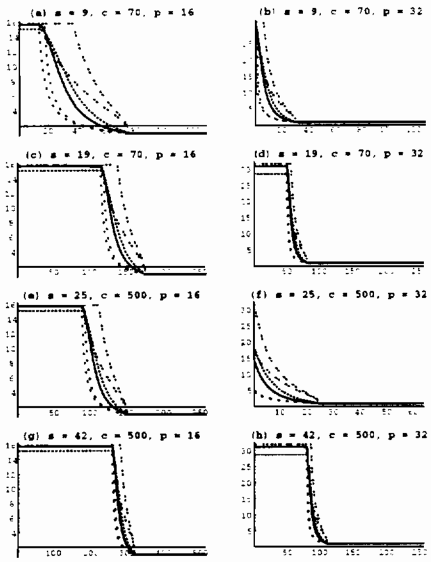


Figure B.13: Tradeoff of speedup with instruction memory size for $P = 16$ and $P = 32$. Abscissa is instruction memory size, in kB; ordinate is speedup. Other parameters given their default values: $b = 1$, $p0 = 0.9$, $q = 1$, $Q = 1/2$, $N = 1000$, $d = 1$.

16 kB of instruction memory per PE without affecting speedup, while larger or later generation chips can have over 64 kB. Chips with 32 processors must be non-minimally sized ($s = 19, 42$), but then may have over 48 kB of instructions.

## B.8 Discussion

In the simulations, the graphs of speedup versus number of processors have two general shapes. When $P_{all} > 0$, the graphs for all task distributions are linear from $(0, 0)$ to $(P_{all}, P_{all})$ and constant from $(P_{none}, k)$ to $(\infty, k)$. They differ between $(P_{all}, P_{all})$ and $(P_{none}, k)$, with $k$ modifying the shape slightly. Conversely, when $P_{all} \leq 0$, the graphs are close to constant $y = k$, with small $P$ in the range $P \approx 16 - 32$ optimal.

These and the other graphs show that there are essentially two regimes, with dramatically different behavior: one where all tasks can be preloaded, and one where they cannot. These two regimes are determined by the size of the task set and the amount of on-chip memory available to hold preloaded tasks. The former is application-specific, while the latter is a function of the architecture, especially the number and complexity of the PEs.

The actual speedup value, and to a lesser extent the shape of the curve, are affected as well by $k$, the amount of computation per task. The parameters $q$, the ratio by which a single large processor is faster than a single PE, and $b$, the bandwidth ratio between the parallel and sequential architectures, have simple linear effects on the speedup.

Unsurprisingly, the quantization assumption, $A = 0 \bmod P$, is important for $P \approx P_{none}$, i.e. when few models fit in a PE's local memory. Larger numbers of PEs make it harder to (approximately) satisfy this assumption, so it suggests that small $P$ designs will be more general-purpose.

We see that, when not all tasks can be preloaded, preloading is helpful mainly for *random-hyperbolic*, but this may be an important case as it corresponds to Zipf's Law, which describes the distribution of word models, and to other power laws. The essential point is that *effectively* the entire task set must be preloaded for good performance, and that extremely non-uniform task distributions such as the hyperbolic are needed if preloading the entire model base is to be avoided.

I have generally assumed model bases on the order of 1 to 5 megabytes in size. Smaller model bases allow greater parallelism, as they fit on-chip with larger numbers of PEs. However, speedup is very sensitive to having all models fit on-chip. Thus, a given design must be careful in choosing $P$ to allow for the largest model base in its range of target applications.

## B.9 Conclusions

One can distinguish three cases:

1. If the entire task set fits on-chip, then linear speedup is obtained. Because of the tradeoff between on-chip memory and PE complexity, this points in the direction of many small processors per chip. Figure B.1 shows that task set sizes of 1 to 3

megabytes may be reasonable.

2. If the task set does not fit on-chip, and there is little work per task, the small $k$ generally implies small parallel speedup ($\leq k$) due to bandwidth limitations and area tradeoffs. In this case having many PEs per chip ($P > k$) is pointless.

3. For any task set, if $k$ is large one has parallel speedup up to $P = k$, independently of the amount of area for preloading, as long as each PE's local memory can hold at least one task.

The implication is that if the task set does not fit on-chip, then, in general, having $P > k$ is pointless. Thus achieving significant (order-of-magnitude) speedup from the parallel architecture requires a substantial amount of work per task, say $k \geq 10$.

Recall that $k$ is the ratio of a task's computation time to the time it takes to load it. If this is measured in clocks or operations, the fact that the time to load, a value of a particular size is probably 2-4 times slower than the time to perform an arithmetic operation on it. Loading a task means loading some number of 'values' (component, data, parameters, ...) for the task, each of which takes, say, $l$ system clocks to load, hence $(2-4)l$ on-chip clocks. If operations take a single on-chip clock, then achieving a given value of $k$ requires computing $(2-4)lk$ operations per value. For example, with 32 bit values, and a 16 bit Rambus memory interface [Cri97] running at chip speed, achieving $k = 10$ requires 20 operations per value[6] . If the Rambus runs at 1/2 chip speed (say a 1.6 GHz chip clock and a 800 MHz Rambus clock), this becomes 40 operations per value. In reality, these memory data rates are maxima, and apply only when memory accesses are local to a column decode buffer, but none the less, achieving $k = 10$ is even more difficult than it might seem.

[6]Pin limitations suggest a 16 bit per-processor memory interface (requiring 31 data and control pins) for a 32 processor chip. For upward compatibility, the Rambus interface has been designed to support a 31 pin interface, although the mass market parts using Direct RDRAM have a 76 pin interface and do not multiplex data with addressing [Cri97]. A 76 pin interface precludes a 32 processor chip where each processor has its own path to memory.

Speedup also depends linearly on $q$, comparative speed of the sequential processor, so that while achieving $k = 10$ may take 20-40 operations per datum, achieving a speedup of 10 might take $(20 - 40)q$. So minimizing $q$ is also of the essence; generally, this points in the direction of using more complex PEs.

My conclusion is that there are two viable architectural possibilities: if the range of target applications allows each task set to fit entirely on-chip, then an architecture of many small processors may be preferred (there are other factors, such as Amdahl's law, that may still preclude using many PEs). In all other cases, a few complex PEs will be preferred.