Performance Prediction for Loop Restructuring Optimization

Theresa Alexander
B.S. Physics/Computer Science, Carnegie Mellon University

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

October 1993

The thesis "Performance Prediction for Loop Restructuring Optimization" by Theresa Alexander has been examined and approved by the following Examination Committee:

Michael Wolfe
Associate Professor
Thesis Research Adviser

Steve Otto
Assistant Professor

Allen D. Malony
Assistant Professor
University of Oregon

Dedication

I dedicate this work to all women in science and engineering. May there be many more degrees earned by women at all levels of education.

Acknowledgements

I would like to thank my advisor, Dr. Michael Wolfe, for helping my interest in architecture and compilers to grow and develop into a Master's thesis and an education. I very much appreciate the advice and support I received along the way. I would also like to thank the Sparse research group, especially Eric Stoltz and Michael Gerlek, I appreciate the discussions of papers and ideas that help stimulate my interest in the field and also the company and support while my thesis progressed.

I would like to thank my parents for raising me with the knowledge that there were no limitations on what I could could accomplish. I thank my brother for giving me a model of youth and enthusiasm in academics; and my sister for her endless support and understanding and long telephone calls across the country.

Finally, and most of all, I would like to thank my husband, Bill Alexander, for bottomless encouragement and patience; and also for helping me through a rough transition as I searched for my niche in academics and industry.

Contents

List of Tables

List of Figures

Abstract

Performance Prediction
for Loop Restructuring Optimization

Theresa Alexander, M.S.
Oregon Graduate Institute of Science & Technology, 1993

Supervising Professor: Michael Wolfe

Loop restructuring optimizations can effect the performance of a program in a number of ways. They can change memory access patterns that effect data locality and cache utilization. They can also enable parallelism of a loop. Furthermore, loop restructuring may change conditions that enable or disable other optimizations. It is not well understood how to find the optimal loop structure of a program.

Performance prediction is one way to give feedback to a user in order to compare predicted performance times of a program under different loop structures. We attempt to empirically create a linear performance model by fitting parameters of programs known at compiler time to the execution time of the programs. We use regression analysis to calculate the coefficients and intercept of the linear model. The challenge is to find a complete set of parameters that reflect both the characteristics of the code and the target architecture.

Chapter 1

Introduction

Some of the common problems with writing scientific programs for either single or multiprocessor computers include the fact that although there are a number of optimizing loop transformations, it is not well understood which transformation or combination of transformations will give the best results. Program transformations have a large effect on cache and local memory accesses. Loop interchanging may increase data locality so that memory accesses follow the cache line increasing cache utilization. Loop transformations may also enable parallelization or vectorization of a loop.

Performance prediction in a compiler would enable the programmer or the compiler to compare predicted execution times for each loop construction. We address this by attempting to form a statistical model which will fit parameters known at compile time with execution times of scientific loops. We build the model by symbolically counting a number of program features at compile time. When the program is run, its execution time is measured, along with the dynamic value of the program features.

We chose a representative group of scientific code that we felt would cover characteristics of most scientific code. We then divided the problem into two subsets, single and multiprocessor execution. The target architecture used was a 20 processor Sequent Symmetry, a shared memory multi-user system. The Sequent Symmetry executes each job on a single processor unless a compiler directive requests a loop to be distributed to more than one process. Note that there is a difference between executing a loop on more than one process and executing a loop on more than one processor. If there are enough free processors each process is assigned to a different processor. Otherwise more than one process can be assigned to the same processor. In the course of this evaluation, the load on the Sequent Symmetry was low and each process was assigned to its own processor. Thus we will not include load values in our performance prediction. The effects of system load are addressed later in the paper.

1.1 Related Work

Some research has been done in ordering optimizing transformations. Whitfield and Soffa [WS90] approached the problem from a theoretical direction. They developed a framework based on axiomatic specification technique. The framework makes use of preconditions and postconditions that must exist before and after applying optimizations. Transformations can create conditions that must be present in order to apply other optimizations and they can destroy conditions that might have enabled other optimizations. Using this information and including the perceived usefulness of each optimization, an ordering is presented in order to get the best use out of optimization transformations.

The optimizations considered in the above reference include the following:

DCE Dead Code Elimination

CTP Constant Propagation

ICM Invariant Code Motion

LUR Loop Unrolling

SMI Strip Mining

FUS Loop Fusion

INX Loop Interchanging

After considering the preconditions and postconditions and also the importance of each optimization, a final ordering was determined. Figure 1.1 depicts the ordering.

This graph suggests the following ordering:

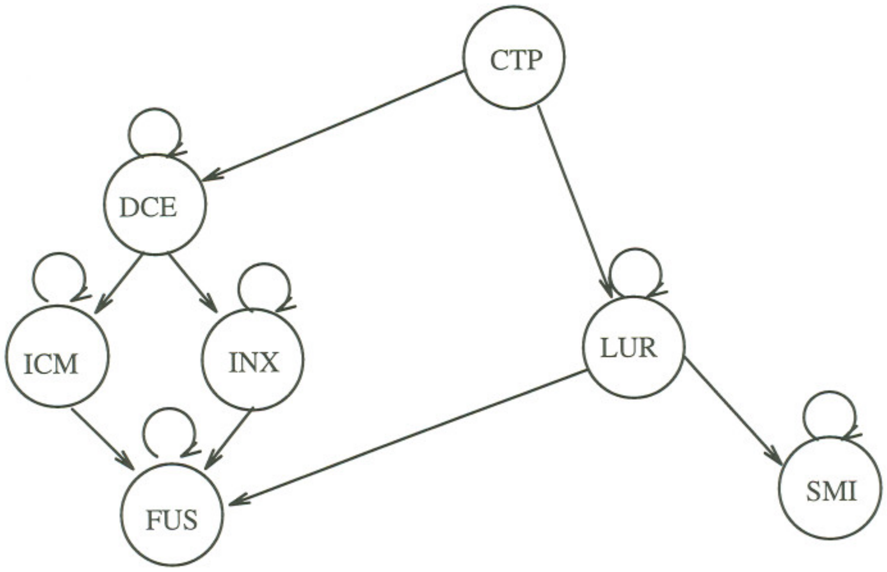

Figure 1.1: Ordering Graph

1. **CTP** - Constant Propagation

2. **DCE** - Dead Code Elimination

3. **INX** - Loop Interchanging

4. **ICM** - Invariant Code Motion

5. **INX** and **ICM**

6. **LUR** - Loop Unrolling

7. **FUS** - Loop Fusion

8. **SMI** - Strip Mining

Although our study did not include analysis of the interactions of each of these optimizations, it is within the capabilities of the prediction model. Each optimization will change the measured parameters somewhat and the predicted performance will allow the user to make an informed decision concerning which transformations to apply. In order to determine the optimal ordering of transformations, however, all legal orderings would have to be tried and the performance predictions compared.

Gannon, Jalby and Gallivan [GJG88] describe a method for using data dependence to estimate cache and local memory demand in scientific code. These estimates take the form of "reference" windows for each variable that reflect the set of elements that should be kept in the cache. In important special cases the size of the window can be estimated and a lower bound can be predicted for the number of cache hits. This estimate is useful to guide program transformations in an attempt to optimize cache performance.

They show that data dependence analysis can be extended to give a more refined algebraic structure to a class of data dependences associated with array index expressions that are common in scientific code. These dependences are called "uniformly generated". A reference window is associated with each data dependence. The reference window of a data dependence between two statements describes the set of elements (section of the array) that must be kept in a fast memory level to make sure that any data referenced by both statements will stay in the fast level as long as both statements continue to be executed and continue to reference that data item.

They further show that program transformations like loop interchange and blocking or tiling have a substantial effect on the size of the windows and therefore on the demand for space in the fast level.

This work is very similar to ours as far as the goal of optimizing programs with program transformations. Our research produces feedback through performance prediction much as theirs does through the size of the active reference windows. We consider program optimization on a more global level, however, taking into account all contributions to performance, rather than just cache and local memory.

Quite a bit of performance analysis research has been performed on an Alliant FX/8 system running the Cedar operating system at University of Illinois. Dimpsey and Iyer [DI88] used statistical cluster analysis to form a state transition model that jointly characterizes user concurrency and system overhead. The sample workloads they used were two distinct, real, scientific workloads. Software facilities were used to measure both the amount of concurrency in the workload and the overhead associated with virtual memory and system operations. For both workloads they found system behavior to be highly predictable, largely due to slow changes in system states between states of high user concurrency and high system overhead.

Other work by Dimpsey and Iyer measure performance degradation due to multiprogramming overhead on a time shared, interactive, multiprogramming environment [DI89]. They found that there is a correlation between the completion time of a program and the amount of system overhead measured during program execution. This study is restricted to operating system overhead consisting of process creation and scheduling, context switching, vector maintenance, device interrupts, synchronization, and paging. It does not, however, consider system overhead caused by increased memory contention for physical and cache memories due to multiprogramming. Their results show that the multiprogramming overhead of parallel jobs consumes at least 4% of processor time. With two or more serial jobs introduced to the system, the overhead becomes 5.3%.

The approach used by Dimpsey and Iyer is similar to our approach. However they concentrate on measurements of machine performance taken by instrumentation added to the machine. Our research focusses on measurements taken in program analysis and the relationship to performance time.

Menasce and Barroso [MB92] developed a method for performance prediction of scientific code on shared memory multiprocessors with emphasis on studying the nondeterministic behavior caused by memory contention. They determined that the execution time is dependent on the shared memory access time which is in turn dependent on the pattern of memory access requests, network contention, and memory module contention. Their methodology obtains the execution time of parallel programs composed of several concurrent tasks. The precedence relationship between the tasks is a task graph. The execution time of each task is shown to be given by the sum of two terms: a fixed term and a nondeterministic term which is a function of the contention for shared memory.

We incorporate some of this information in our model. We measure memory contention between different processors in parallel execution. We do not, however, consider memory and network contention caused by multiple programs in the multiprocessing system.

Chapter 2

Abstract Machine Model

The performance time of a program depends on many architectural features of the computer it is run on. Although the user may not be concerned or knowledgeable about the details of the computer he or she is using, in order to optimize the program they may have to be taken into account. Using a performance model is a way of encapsulating these details in a tool and alleviating the need for the user to be an expert on the machine. Some of the architectural features that our model includes are the instruction pipeline, the memory hierarchy, multiprocessor overhead, vector execution, interleaved memory, and loop overhead. Each of these is described in more detail in the following sections.

2.1 Processors

Most modern architectures have pipelined execution. A typical pipeline may be divided into instruction fetch, instruction decode and register fetch, execution and effective address calculation, memory access, and write back stages. Each of these steps can be performed simultaneously and thus the execution of multiple instructions is overlapped. Because floating point operations take more cycles in the execute stage than the other stages of the pipeline, floating point functional units are often also pipelined. Thus multiple floating point operations can take place at the same time. Also, multiple functional units may be available, increasing potential parallelism.

Pipeline stalls can have a serious effect on performance. It is not possible to keep the pipeline full all the time due to pipeline hazards. These hazards include resource conflicts, when there is not enough hardware to keep all of the current instructions executing at the same time due to demand for functional units or memory ports that are busy. Data hazards occur when an instruction cannot proceed in the pipeline until the result of a previous instruction is ready. Control hazards occur when the next instruction fetch depends on the result of a previous instruction. Branches are often condition, so the address of the next instruction fetch is not known until the conditional is resolved. Many architectures will perform branch prediction to alleviate the penalty of control hazards.

We model the processor by including measures of floating point adds, subtracts, multiplies, and divides. Because floating point operations take significantly more cycles to complete than integer operations, and because we limit our scope to scientific programs, we limit these counts to floating point operations and disregard integer operations. Pipeline hazards are measured only indirectly by including performance times in the model of programs whose execution includes pipeline stalls. Because their performance times include the time in which the pipeline was not full, that information and how it correlates with the parameters of the code is included in the model.

2.2 Memory Accesses

Most memory hierarchies include a small fast cache in addition to main memory itself. Caches take advantage of spatial and temporal locality of memory references. As data is referenced an entire cache line is loaded into the cache. If the same memory address is referenced again in a short amount of time that cache line will not have been replaced and the data will be found in the cache (temporal locality). In this case we have avoided a trip to main memory. If nearby addresses are referenced they will be found in the same cache line already in the cache (spatial locality). This also avoids main memory access. If the cache has a high hit rate average memory access time as seen by the processor is very fast. Cache misses, however, have a high penalty because they must access main memory which is relatively slow.

Accesses to main memory are limited by the latency and bandwidth of main memory. The latency is the number of cycles that it takes to access memory and the cycle time, the minimum time between requests to memory. The latency is important to the cache because it affects the cache miss penalty. If the cache line is large, bandwidth also plays a part in the time to load and store a cache line. Memory bandwidth is also a concern of vector processors because they load and store an entire vector of data at a time. Vector processors are discussed in more detail below.

We model memory access time by including measures of the total number of memory accesses. More information about the pattern of memory accesses is included by measuring the number of stride-1 memory accesses in the inner loop as well as the number of non-stride-1 accesses and invariant accesses in the inner loop.

2.3 Multiprocessors

There is additional cost associated with parallel execution on multiprocessor architectures. This includes operating system overhead to create each additional process. It also includes overhead from memory contention and cache coherence.

Consider the following Fortran subroutine.

```
      subroutine add(a, b, c)
      real a(100,100), b(100,100), c(100,100)
      do 10 i = 1, 100
c$doacross
        do 10 j = 1, 100
          c(i,j) = a(i,j) + b(i,j)
   10 continue
```

This subroutine includes a compiler directive written for the Sequent Fortran compiler that instructs the compiler that the inner loop should be performed in parallel. If there are 9 processors available for use, in addition to the processor that is already active, each time the inner loop is called 9 processes must be activated and put on the job queue for the processors. Each process will be responsible for calculating 10 iterations of the inner loop. The outer loop is performed 100 times and thus this overhead is incurred 100 times during the execution of this subroutine.

Another version of the loop is more optimal:

```
      subroutine add(a, b, c)
      real a(100,100), b(100,100), c(100,100)
c$doacross
      do 10 i = 1, 100
        do 10 j = 1, 100
          c(i,j) = a(i,j) + b(i,j)
  10  continue
```

In this version only 9 new processes are created. The current process also participates. Each process performs 10 iterations of the outer loop. Each of these iterations includes 100 iterations of the inner loop. However, no additional overhead is incurred to perform the inner loop.

Another factor that adds overhead is increased traffic on the memory bus. Now we have 10 processors accessing the same arrays. Consider the following case:

```
      subroutine add(a, b, c)
      real a(100,100), b(100), c(100,100)
      do 10 i = 1, 100
c$doacross
        do 10 j = 1, 100
          c(i,j) = a(i,j) + b(i)
  10  continue
```

In this subroutine the inner loop is performed in multiple processes. Each iteration of the inner loop accesses the entire b array. If this were performed on a single processor the b array would be cached and accesses would be very fast. In this case the access is a read access and thus may be cached locally on the processors as a read-only copy. This is dependent on the cache architecture.

The case of write conflicts is more complex. The case in which more than one processor is updating the same memory location will result in a race condition to determine which value is written last and thus will be the resulting value. The results of this situation is indeterminate. If the loop was parallelized while taking care not to violate data dependences this condition will not occur. However, false sharing occurs if more than one processor writes to memory locations that are on the same cache line. The cache coherence protocol dictates how different processors share data in their caches. A popular protocol is write-exclusive. In this case the processor performing the first memory access will load the cache line into its local cache. Then as the subsequent processors write to a memory location on the same cache line they will have to send an invalidate signal so that the first processor will either write its copy to memory or, if it has a read-only copy, invalidate the cache line in their cache. When all copies of the cache line have been written or invalidated, the cache line is read from memory into the cache local to the processor that has just requested the data. If several processors are writing data to the same general location in memory this overhead can become very significant.

2.4 Vector Processors

Vector processors include separate functional units for vector operations. Vector operations are operations between arrays of numbers (vectors). A typical vector operation may multiply two vectors of 64 floating point numbers element-wise and obtain a new vector containing the result. The sequential code for this operation may be a loop stepping through two arrays, during each iteration two array elements are multiplied and the result stored in a third array.

The vector unit contains several functional units which are deeply pipelined. The latency for these units can be several clock cycles, but a new operation can be started every clock cycle. Operations include arithmetic operations, and also memory accesses and effective address calculations. In many architectures multiple vector operations can be done at the same time, creating parallelism among the operations on different vector elements.

Vector operations have several advantages over pipelined scalar operations on the same input.

- Because the computation of each result is independent of the computation of the previous result there are no data hazards. This allows the deeply pipelined functional unit to operate without expensive stalls. The absence of data hazards was determined by the programmer or compiler when the vector instruction was written.
- Control hazards are reduced because the loop conditional no longer exists. The entire operation of the loop is contained in the vector operation.
- Fewer instructions must be fetched because a vector instruction contains operations for each element of the vector. This reduces the bottleneck of fetching instructions.

Vector operations also access memory in regular ways. Vectors either move across the row or column of an array. Depending on the language being used, one of these patterns will access memory contiguously. That is, each data element accessed is stored contiguously in memory. The other pattern will access memory at a constant interval, called the stride of the vector. If the program is iterating through the columns of an array in Fortran, each access will be offset from the last by the number of rows in the array. If the architecture contains vector registers, the vector elements are read into the vector register with a vector load and the data elements are accessed sequentially for the vector operations and then written back to memory using a vector store. Thus the stride will not effect the operation of the vector functional unit.

In the course of this study we did not model program execution on vector processors. However, measurements that would have been added to the model include the number of vector operations. This measurement, when correlated with the performance time of a program, would capture information such as the vector length on the machine and the startup time for a vector operation.

2.5 Interleaved Memory

Interleaved memory can decrease the effective memory access time of loading or storing a vector. Interleaved memory is organized into a number of banks. The low bits of an address are used for bank selection. Each bank can be accessed independently, allowing memory accesses in different banks to be performed in parallel. Although the memory bus remains the same size, the memory bandwidth increases because the access times are overlapped with other memory accesses in different banks. Vector data is often consecutive in memory (stride 1) so each access will be to the next memory bank. By the time one bank is accessed again, the previous access may have completed, allowing maximum speedup from the interleaved memory. In the worst case, the stride of the vector is the same as the number of memory banks. In this case, each memory access will access the same memory bank and there is no overlap of access times.

Interleaved memory is also advantageous for caches. A cache loads data a cache line at a time. Since a cache line is consecutive in memory, it accesses memory sequentially and makes full use of the memory banks. If the cache has a write back policy it will also write a full cache line at a time, getting even more efficiency out of the interleaved memory.

Although in this study we do not model interleaved memory, our measurements include stride-1 one and non-stride-1 memory references. In the case of memory operations this information includes the number of memory accesses that follow the cache line. In the case of vector loads and stores this information discloses how much efficiency can be extracted from the interleaved memory when loading or storing vectors, this efficiency is dependent on bank conflicts. Thus the necessary information to model interleaved memory in already included in the parameters for the performance model and could be implemented on the appropriate architecture.

2.6 Loop Overhead

A certain amount of overhead is associated with every loop entry in a program. Each loop contains a number of instructions that must be performed. It contains a conditional that determines whether the loop is executed. The conditional must be checked at every iteration of the loop. When the bottom of a loop is reached a jump operation must be performed to in order to bring the control back to the top of the loop. Loop indices must be incremented on every iteration. Together, this overhead includes three or more instructions that must be executed at every iteration of a loop.

Pipelined architectures may have the additional overhead of having to stall the pipeline as the loop conditional is being executed. Subsequent instructions will be fetched from different locations depending on the results on the result of the conditional and might not be fetched until the result is available.

Loop unrolling is one optimization that can reduce loop overhead. More than one iteration of the loop can be performed within the same loop body. In this case the loop is performed only a factor of the number of initial iterations and the overhead is reduced proportionally.

In order to model the performance cost of loop overhead, we measure the number of total loop iterations executed.

Chapter 3

Performance Model

The performance model is a linear model that can be applied to parameters measured from scientific programs to predict the performance time of these programs. The model is created from parameters measured from real scientific programs and the execution time of these programs. The linear relationship between these measured parameters and the performance time forms the performance prediction model.

$$performance_{time} = coef_0 + coef_1 P_1 + coef_2 P_2 + ... + coef_n P_n \tag{3.1}$$

where P_i are the parameters, and the coefficients are determined experimentally. Parameters are static counts taken from the code such as the number of floating point operations or the number of stride-1 memory accesses in the inner loop. Each parameter includes information important to predict performance of the program.

3.1 Performance Information

The performance model makes a prediction of performance based on measured parameters and the linear model. These parameters can be measured at compile time. Performance information that is included in the parameters includes the contribution to performance by floating point instructions, cache effects, memory contention, parallel process overhead, and others.

Floating point operations take many more cycles in the instruction pipeline than integer operations. Because scientific code is dominated by floating point operations, we model the performance of a code segment by the number of floating point operations and disregard integer operations.

Memory operations also have a large contribution and are included in the measured parameters. Each processor has a local cache which stores recently referenced data items. Memory accesses fall into two cases: those that hit in the cache and those that miss in the cache. In order to measure this effect, we calculate the number of stride one operations in the inner loop. Stride one operations are those where the array index is incremented such that each successive iteration references the next sequential array element stored in memory. In Fortran, this means that it is incremented along the column of the array. These operations correspond to memory accesses that are most likely to hit in the cache because of spatial locality. References that are stride one will follow the cache line; as long as the cache line is not replaced in the cache, each memory successive memory reference will refer to a data element in that cache line.

In the case of parallel loops, the loop iterations are divided among the participating processors. The number of processors that will participate may be known at compile time. Otherwise, the maximum number available may be assumed. If each iteration of a loop accesses the same location in memory, there will be memory contention. Each of the processors will try to load the data into their local cache. In the case of write operations only one processor can own the data at a time so this process is serialized and processors have to wait until the data item is released by the current owner before it can be accessed. In the case of read operations more than one copy can exist, but there will be additional overhead incurred by the increase in bus traffic as all of the processors try to load the data. We extract this effect on performance by measuring the number of memory operations that are invariant in the parallel loop. That is, each iteration accesses the same data element.

Another strong effect on performance come from the operating system overhead of forking new processes and synchronizing for parallel loops. This contribution is easily parameterized as it is proportional to the number of parallel forks in a code segment. The way that parallel loops are implemented is specific to the machine it is run on. On the Sequent Symmetry, parallel forks occur at each instance of a parallel loop construct. The number of forks at each parallel loop is equal to the number of processors that you have requested or the minimum number of loop iterations, whichever is less. If the parallel loop is not the outer loop, this cost is incurred at every instance the parallel loop occurs within the outer loop(s).

3.2 Uses of Performance Prediction

A performance model has a number of uses. The intention of this research is to use the performance prediction to compare performance of different loop transformations of a section of code. Performance evaluation and prediction is traditionally used both to investigate how to perform computations efficiently and to determine what resources a computer needs in order to achieve this. This includes uses by both software and hardware developers.

Performance analysis of an application can focus specifically on the performance of a given application, or it may have a broader scope and be aimed at determining potential parallelism in computationally intensive applications. One could also analyze the performance of an application across a multitude of systems to determine the most suitable one. In a more general viewpoint, performance analysis can aid in understanding the effects of architectural features on an applications performance and also investigate the effects of the compiler and operating system on efficiency.

From the hardware developer's perspective, performance analysis is useful to determine the requirements of parallel architectures to support computationally intensive applications. These systems range from shared memory multiprocessors to distributed memory multi-computers with many variations in between. The granularity of the system also ranges from massively parallel systems to systems with small numbers of processors.

3.3 Parameters

The initial parameters measured and used in creating the performance model are discussed:

3.3.1 Sequential

Number of Floating Point Operations: The number of floating point operations is proportional to the amount of time it takes to execute each floating point operation. This measure was refined later in the experiment to include more information about the types of floating point operations.

Number of Memory Operations: The number of memory operations is proportional to the total time spent reading and writing to memory. More information is needed, however, to take cache effects into consideration. This information is included in the following parameters.

Number of Stride One Operations in Inner Loop: The stride one memory accesses in the inner loop are the accesses that follow the cache line. These memory accesses are singled out because they will have much shorter access time on average. The cache line will be loaded on the first cache miss and subsequent accesses will be already in the cache until the end of the cache line is reached.

Number of Non-stride-1 Operations in Inner Loop: The non-stride-1 memory accesses are the memory accesses that will usually not hit in the cache because subsequent accesses are offset in memory by a constant factor that may be greater than the size of the cache line. These accesses usually go to memory and are much more expensive in performance than a cache hit.

Number of Invariant Operations in inner loop: Invariant memory accesses are memory locations that are accessed in every iteration of the loop. If they are kept in a register then memory access is very fast. If it is not kept in a register and the cache line is not replaced from iteration to iteration, the access will hit in the cache. Otherwise the memory access must go to main memory.

Number of Loop Iterations: The number of loop iterations provides a measure for the amount of loop overhead associated with each iteration.

3.3.2 Parallel

The parameters for the parallel model are similar to the sequential model. However, the counts must be adjusted to take into account the fact that each processor only performs a fraction of the operations. The maximum number of operations in a process is performed by the original process that executes any sequential code and forks off other processes when parallel loops are reached. The number of operations executed by this process is the total number of operations executed in the sequential sections and 1/p of the operations performed in the parallel sections where p is the number of processors.

Maximum Number of Floating Point Operations in Parallel: The maximum number of floating point operations is proportional to the time each processor spends executing floating point operations.

Maximum Number of Memory Operations in Parallel: The maximum number of memory operations is proportional to the time spent by each processor reading and writing to memory.

Number of Parallel Stride-1 Operations in Inner Loop: The stride one accesses in the inner are the accesses that follow the cache line of the cache local to each processor. Although it is possible for these accesses to cause memory contention between processors, this measurement is left for a different parameter.

Number of Parallel Non-stride-1 Operations in Inner Loop: The non-stride-1 memory accesses are the memory accesses that will usually not hit in the cache of the local processor because subsequent accesses are offset in memory by a constant factor that may be greater than the size of the cache line.

Number of Parallel Invariant Operations in Inner Loop: Invariant memory accesses are memory locations that are accessed in every iteration of the loop. Since iterations is executed in parallel, each processor will be accessing the same location in memory. This number is a measure of memory contention in the parallel loop.

Number of Parallel Stride One Operations: This parameter is the same as the number of stride one operations in the inner loop in parallel if the inner loop is the parallel loop. However, it includes all stride one operations within parallel loops. This measurement as well as the next two reveal memory access patterns to shared memory.

Number of Parallel Non-Stride-1 Operations: The total number of non-stride-1 memory accesses that take place inside a parallel loop.

Number of Parallel Invariant Operations: The total number of invariant memory accesses that take place inside a parallel loop.

Number of Parallel Forks: The number of parallel forks parameterizes the contribution to the performance time caused by the overhead of forking each process.

3.3.3 Additional Parameters

As the experiment progressed it became clear that not enough performance information was being included and that some of the information was not at a fine enough granularity. This caused the following parameters to be added to the list in addition to or to replace previous parameters.

Floating point operations vary significantly in number of cycles to complete. By reducing the granularity of this measurement to differentiate between each floating point operation more accurate performance information can be inferred.

Floating Point Adds: This parameter and the following parameters replace the Number of Floating Point Operations parameter described previously. They provide more accurate information by isolating each type of floating point operation.

Floating Point Subtracts

Floating Point Multiplies

Floating Point Divides

Floating Point Sqrt: This parameter and the next measure are added to extend the standard floating point operations to include a couple of common operations. The number of square roots parameterizes performance time due to calculating square roots of floating point numbers.

Floating Point Exp: The number of exponentiations parameterizes performance time due to calculating exponentials of floating point numbers.

Similarly, the parallel model was adjusted to include more accurate performance information about floating point operations.

Maximum Floating Point Adds in Parallel: This parameter and the following parameters replace the number of floating point operations in parallel.

Maximum Floating Point Subtracts in Parallel

Maximum Floating Point Multiplies in Parallel

Maximum Floating Point Divides in Parallel

Maximum Floating Point Sqrt in Parallel: The maximum number of square roots on a single processor is proportional to the contribution to the performance caused by the execution of an operation to calculate the square root of a floating point number.

Maximum Floating Point Exp in Parallel: The maximum number exponentials on a single processor is proportional to the contribution to the performance caused by the execution of an operation to calculate the exponent of floating point point numbers.

The previous information on memory accesses in parallel loops was not always able to account for all occasions of memory contention between processors. Invariant accesses were known to cause memory contention but stride-1 and non-stride-1 did not provide enough information to determine memory access patterns between processors. Furthermore, read and write contention have very different performance effects. Read contention may cause very little performance hit as multiple copies can exist at the same time. In the case of write contention, however, only one copy can exist and there is overhead associated with managing this copy.

Total Parallel Read Contention: The total parallel read contention is a measure of the number of memory fetches that will access the same location across iterations of the same parallel loop. This number is proportional to the overhead caused by multiple processors loading the value or the cache line into their local memory (register or cache).

Total Parallel Write Contention: Total parallel write contention is a measure of memory stores that will write to the same location from different iterations of the parallel loop. This number is proportional to the overhead caused by managing consistency of the global memory space.

In our final model we include ten parameters for the sequential model and sixteen parameters for the parallel model. Processor performance is measured by the number of floating point adds, subtracts, multiplies, and divides as well as the number of exponentiations and square roots. In the parallel model these measures are adjusted per processor. Memory accesses and contention is measured by the total memory accesses as well as the number of stride one, non-stride one, and invariant memory accesses in the inner loop. The parallel model includes these measurements in the inner loop as well as the parallel loop. Additional measurements include memory read and write contention. Loop overhead is measured by number of loop iterations. Process overhead is measured by the number of parallel forks.

Chapter 4

Experiment

Our goal is to predict the execution time of a program by using a performance model based on the parameters discussed in Chapter 3. We count these parameters for a series of different programs and different structures of each program. We then measure the actual execution time of each program and perform a least squares fit and extract the coefficients and the intercept that will provide a linear fit between the parameters and the execution time. If the least squares fit was good and the loops used in the analysis were representative, then these results form a model that can be used to predict the execution time for other scientific code.

4.1 Linear Regression

The performance model was a result of multiple regression analysis. Regression analysis is a data analysis technique used to investigate the relationship between a dependent variable Y and one or more independent (predictor) variables $X_1, X_2, ..., X_n$. The regression model can be written as:

$$Y_i = B_0 + B_1 X_{1i} + B_2 X_{2i} + ... + B_n X_{ni} + e_i \tag{4.1}$$

Y_i is the measured execution time for program i and the X's are the measured parameters for that program B_0 is the intercept for the line, $B_1..B_n$ are the partial regression slope coefficients, and e_i is the residual term.

This equation corresponds to a system of equations of the form

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_p \end{pmatrix} = \begin{pmatrix} B_0 + B_1 X_{11} + \ldots + B_n X_{n1} + e_1 \\ B_0 + B_1 X_{12} + \ldots + B_n X_{n2} + e_2 \\ \\ B_0 + B_1 X_{1p} + \ldots + B_n X_{np} + e_p \end{pmatrix} \tag{4.2}$$

In matrix notation this system of equations can be written as

$$\begin{pmatrix} Y_0 \\ Y_1 \\ \vdots \\ Y_p \end{pmatrix} = \begin{pmatrix} 1 & X_{11} & X_{21} & \ldots & X_{n1} \\ 1 & X_{12} & X_{22} & \ldots & X_{n2} \\ \vdots & & & & \\ 1 & X_{1p} & X_{2p} & \ldots & X_{np} \end{pmatrix} \begin{pmatrix} B_0 \\ B_1 \\ \vdots \\ B_n \end{pmatrix} + \begin{pmatrix} e_0 \\ e_1 \\ \vdots \\ e_p \end{pmatrix} \tag{4.3}$$

where

Y = 1 by p column vector of observations on the independent variable Y

X = $n+1$ by p matrix that results from p observations on n independent variables $X_1, ..., X_n$, where the first column of 1's represent the intercept term B_0; that is, $X_0 \equiv 1$

B = 1 by $n+1$ column vector of unknown parameters $B_0, B_1, ..., B_n$, where B_1 is the intercept term, and $B_1, ..., B_n$ are called the partial regression coefficients or weights

e = 1 by p column vector of p residuals e_i

The performance model was created by collecting measurements of execution time along with a number of parameters. The execution times are the dependent, Y_i values and the other measured parameters become the independent variables, X_{ij} in the model. There is an assumption that these parameters are related linearly with the execution time. Violations of this assumption are discussed in the next section. Once the least squares fit is performed, the model can be used to predict the execution times of code segments that were not part of the model. This can be done by solving the equation

$$\hat{Y} = B_0 + B_1 X_1 + B_2 X_2 + ... + B_n X_n \tag{4.4}$$

Where $\hat{Y}$ is the predicted value for the execution time of the code segment with parameters $X_1, X_2, ..., X_n$.

4.1.1 Goodness of Fit

The multiple coefficient of determination, R^2 (also known as the squared multiple correlation coefficient) is used to measure how good a fit we achieved in creating the model. This statistic represents the the proportion of total variability in the execution times that is accounted for in the regression model. R^2 is only a measure of how well the model fits the data that we used in our regression analysis, not how well the model can be used to predict performance on new data.

$$R^2 = 1 - \frac{\sum_{i=0}^{n}(Y_i - \hat{Y}_i)^2}{\sum_{i=0}^{n}(Y_i - \bar{Y})^2} = \frac{\textit{total variability} - \textit{left-over variability}}{\textit{total variability}} \tag{4.5}$$

Y_i is the real execution time, $\hat{Y}_i$ is the predicted execution time, and $\bar{Y}$ is the mean of the real execution times over all programs included in the model. R^2 represents the total variability in the response variable Y that is accounted for by the regression model. R^2 is restricted to the interval $0 <= R^2 <= 1$. A value near 1 is indicative of a good fit. By inspection, it is clear that a value of 1 would occur when the predicted data is identical to the predicted data.

4.1.2 Residuals

Another measure of how good the linear fit was on the data is to look at the difference between the real value of Y for each data point and the value of Y predicted by the model. The residual can be calculated as follows:

$$residual_i = Y_i - \hat{Y}_i \tag{4.6}$$

Each data point represents a particular combination of characteristics of a program. The value of the residual is important because it shows how well the model represents that particular collection of characteristics. The performance of any program with similar counts for the parameters would likely be inaccurately predicted by the model. A positive residual means that the model is overestimating the performance of that particular program. A negative residual means that the model is underestimating the performance of that program. How poorly the model is representing the data point can be determined by comparing the residual with the performance value. The magnitude of the residual should be a small proportion of the value.

Residuals can also be used to measure how good a job the model did to predict the performance time of a program. In this case, after predicting the performance time the program is run and the actual performance is measured. This is the value used to compute the residual.

Residual versus performance value plots are used in the results section to measure how well the model represents the prediction set of programs and to measure the effectiveness of the model on the experimental set of programs.

Another use of residual plots is to determine if the model is incomplete. If the residuals form a linear pattern, there may be a linear relationship that is not included in the model. The addition of another parameter may help to reduce the residuals and provide a better model. On the other hand, a curvature in the residuals may represent some non-linear factor that is not represented by the model. This would disprove the assumption we made previously that a linear relationship exists between the parameters and the execution time.

4.2 Prediction Set

The programs in the prediction set included a number of programs with different versions of the same program representing a different loop structure, problem size, or number of processors. The criteria for selecting the set of programs used to create the performance model included a number of considerations. The set should include characteristics of programs that are commonly used in the scientific community. This is due to the fact that the model will be built around the parameters measured from these programs. Programs that include characteristics that are not included in the base set might have parameters that are not well represented in the model and thus the performance prediction for these programs will not be accurate.

For example, in the simple case where the base set of programs include no floating point divides, the resulting model will not include any information about floating point divides and the coefficient for that parameter would be zero. As a result, if a performance prediction is performed on a program that contains floating point divides in the inner loop, that prediction is likely to underestimate the performance time. In this example, the model does not include this relationship between number of floating point divides and performance. Thus the resulting performance prediction will be inaccurate.

Another example is memory contention. The relationship between memory contention and performance was discussed earlier. If the base set does not include a program that has this characteristic, then the model will not contain the necessary information to accurately predict the performance of programs that contain memory contention. In fact, this is true for any of the parameters that we use in our model. They must all be well represented.

Another factor that must be considered is the proportion in which each parameter is represented. If a characteristic is included, but it is under-represented in the base set of programs then it may result in becoming an outlier in the fit. The R^2 term which we use to measure the fit does not always expose the fact that these outliers exist. If they are in very small numbers, the R^2 term will not be significantly affected. For this reason, care was taken to represent each characteristic in as even a manner as possible. As an example, instead of including only one version of a program that contains memory contention, multiple versions were included. This reduces the likelihood that that one program would become an outlier in the model and memory contention would not be represented in the model. Due to the dimensionality of the problem it is not possible to graphically represent the problem space in order to inspect the fit for outliers. However, a graph of the residuals after the fit is completed is a good measure of whether there were data points that were not close to the line.

Which programs to use in the prediction set is only one issue. Another is what form of the program to use. Different loop orderings were included for the programs in order to include data with different memory access patterns but the same instructions. Loop transformations were performed using Tiny, a loop restructuring research tool. For example, one algorithm for Cholesky decomposition of a matrix follows:

```
      subroutine chol(n, a)
      integer n
      real a(n,n)
      do 10 k = 1,n
        a(k,k) = sqrt(a(k,k))
        do 20 i = k+1,n
          a(i,k) = a(i,k)/a(k,k)
          do 30 j = k+1,i
            a(i,j) = a(i,j)-a(i,k)*a(j,k)
30        continue
20      continue
10    continue
      end
```

By performing loop distributions and loop interchanges on this code segment and by parallelizing different levels of the nested loops, it is possible to create at least fifteen different versions of this code. A listing of these versions can be found in Appendix A. Each version is characterized by a label of the type K(I(J)). In this case K represents the outer loop which contains I, the next level loop, which in turn contains J, the innermost loop. Parallel loops use a "p" suffix, as K(I(Jp)).

Finally, performance depends on which loop level was run in parallel and how many processors it was run on. The Sequent does not allow nested parallel loops so each allowable level of parallelization was included in the model. The loop transformations mentioned above change the dependences within the loop which determine which loop levels are allowed to be parallelized. Each version of parallel loop was run on differing numbers of processors in order to include data for each.

The base set of programs that was used for the final performance models includes the following:

Cholesky Decomposition A specialization of LU decomposition for symmetric, positive-definite matrices. The sequential model includes eight versions of the program and problem size n = 100, 200, 300, and 400. The parallel model includes seven versions of the program run with number of processors p = 2, 4, 6, 8, and 10 and array size n = 50, 100, 200, 300, and 500.

Matrix Multiply Multiplication of two matrixes. The sequential model includes four versions of the program and problem size n = 100, 200, 300, and 400. The parallel model includes nine versions of the program run with number of processors p = 2, 4, 6, 8, and 10 and array size n = 50, 100, 200, 300, and 500.

Other Programs created by hand in order to represent characteristics not found in the rest of the base set. Each of these has one sequential and one parallel version. The sequential program is executed with problem size n = 100, 200, 300, and 500. The parallel version is executed with problem size n = 100, 200, 300, and 500 and number of processors p = 2, 4, 6, 8, and 10.

The code for each of these programs is included in Appendix B.

4.3 Experimental Set

The experimental set includes both programs that were used in creating the model and programs that were not used in the model. Programs were chosen that were representative of programs used in the scientific community.

LU Decomposition Decomposes a matrix into its upper and lower triangular parts. There are four sequential and six parallel versions of this program. The sequential versions are executed with varying problem size n = 100, 200, 300, and 400. The parallel versions are run with varying problem size n = 100, 200, 300, and 400 and varying number of processors p = 2, 4, 6, 8, 10.

Back-Substitution The back-substitution part of Gaussian elimination. There are two sequential and two parallel versions of this program. The sequential versions are executed with varying problem size n = 300, 400, 500, and 600. The parallel versions are run with varying problem size n = 300, 400, 500, and 600 and varying number of processors p = 4, 6, 8.

The code for LU Decomposition and back-substitution are included in Appendix C.

Chapter 5

Tools

A number of tools were used in the data collection and analysis phase of this project. The Tiny program was used to analyze programs and collect counts for the parameters. It was also used to perform loop transformations on the program. Splus, a data analysis environment, was used to create the linear model from the data. Other tools assisted in running the programs and collecting data.

5.1 Tiny

Tiny is a program restructuring tool for a simple imperative language [Wol91b]. The main objective of Tiny is loop restructuring. Tiny has a menu based user interface, which allows users to restructure loops interactively. During an interactive session, illegal restructuring (i.e. those which may change the semantics of the original program) are detected by its built-in data dependence analysis, and users are informed automatically. After this session, Tiny can convert the program to an executable C or Fortran program with parallel execution directives.

Unlike Fortran, the Tiny language is not designed to write big scientific programs. We can easily convert a well-formed loop structure as would be written by Fortran DO statements to a Tiny program and can restructure the program to get more parallelism, although it does not have several important features to write real application programs, like subroutine calls, I/O, or intrinsic math libraries.

5.2 Loop Restructuring Transformation

Using direction vectors and/or other results of the dependence analysis, Tiny can restructure loops without changing the semantics of the original program. Currently Tiny supports the following eight loop restructuring transformations.

Parallelization Try to parallelize a loop. If there is no loop carried dependence for the loop, this transformation succeeds.

Vectorization Try to vectorize a loop. If the loop is the innermost and it has no loop carried dependence cycle, this transformation succeeds.

Distribution Try to distribute a multi-statement loop. All dependence cycles should be kept in a single loop.

Interchange Try to interchange a loop with its immediate outer loop. If there is no $(<,>)$ dependence relation concerning the (outer, inner) loops, this succeeds. Non-tightly nested loops can also be interchanged.

Circulation Try to move the innermost (or outermost) loop to the outermost (or innermost) position in a single step.

Skewing Add (or subtract) an outer loop index to the lower and the upper limits for an inner loop. This is always legal and can change its direction vectors, allowing interchanging or other restructuring.

Reversal Try to reverse the execution order of a loop. If the loop carries no dependencies this succeeds.

More discussion on these transformations can be found in [Wol91a].

5.3 Additions to Tiny

At the time this research was begun Tiny kept limited statistics on a program, such as the number of floating point operations, memory operations, stride-1 operations, non-stride-1 operations, invariant operations, and parallel forks. This is done by visiting each statement in the intermediate representation and keeping a symbolic count of the operations. Operations inside loops are counted in terms of the loop limits. We expanded the statistics collected to include the counts we needed to extract or calculate each of our parameters. This involved making counts of more operations such as floating point adds, etc., and also more specific cases such as operations inside parallel and/or inner loops.

In addition to displaying this information on the screen and writing it to a file, we added the ability in Tiny to write this information as a Fortran procedure. The procedure contained a series of write statements, each containing a string describing one of the statistics. The program written in Tiny was also converted into Fortran and was instrumented to collect timing information. The Fortran program was executed using a script and the timing information and statistics were written to a file.

5.4 Splus

Splus is a statistical language and environment [BCW88]. This was the medium used to process and analyze the data. Splus has a built-in least-squares fit function which uses the QR decomposition of the x matrix. The LINPACK library contains subroutines that use the QR decomposition of x along with the y values to estimate the coefficients, residuals, and other statistics. The Splus linear fitting function takes advantage of these LINPACK subroutines and provides these results and statistics. The residuals are used to calculate the R^2 value and to create the residuals plots used for data analysis. The coefficients and intercept returned by the linear fit function defined the performance models from data collected for each parameter for each program in the experiment set. Splus was also used to calculate prediction times using the measured parameters for the programs we wish to predict and the coefficients and intercept of the linear performance model.

5.5 Other Tools

Data was collected by running each program with the output procedure added by Tiny. Each program was run multiple times while timing information was collected. The average execution times were kept. A script ran each program with differing values for the number of loop iterations (also the size of the arrays) and number of processors. In writing the output file the script parsed and evaluated the symbolic counts substituting reals values for the loop limits and extracted the performance time and the information needed for the model parameters. The data from each program was collected and written in an easy to understand, English format.

An additional script extracted the data from the output file and wrote a data file, packing data in a format readable by Splus. This script allowed the output file to contain any amount of data which might not be used in the model. The information required for the current model could be extracted from the output file. If the model changed and required new information there was no need to re-run the programs; changes could be made in this script to include a different set of data and the data could be extracted from the same output file.

5.6 Execution

Programs were run on the Sequent Symmetry with twenty processors. Execution of programs run in parallel was limited to ten or less processors in order to ensure that no other users were sharing the processors. Each processor is a 16 MHz 386 with 32 KB cache and a 16 B cache line. The operating system is Dynix version 3.2.0. The Fortran programs were compiled using the ATS FORTRAN compiler and the parallel programming libraries. In order to minimize confounding effects from the compiler, optimization was not turned on. The effects of the compiler on the model will be discussed later. Each Fortran program included measured the execution time of the subroutine being measured by running it five times, discarding the first time in order to ensure a warm cache and averaging the rest. Time was measured by the SECNDS() function call that takes a time in seconds as an argument and returns the number of seconds that has elapsed since that time. In this way the number of seconds that elapsed from right before the subroutine call to right after the call completed was calculated.

Chapter 6

Results

Performing the least squares fit on the parameters collected from the prediction set of programs and the performance times for those programs creates the coefficients that represent the performance model. In the following sections we will discuss the fit of each model, sequential and parallel, and the results of performance prediction using these models.

6.1 Single Processor Results

There are ten parameters for the sequential model. These parameters were discussed in detail in the previous chapter. Table 6.1 contains the coefficient for each of the parameters. There are three categories of coefficients: zero coefficients, positive coefficients, and negative coefficients.

A parameter with a zero coefficient can be removed from the model without any loss of information. In this case both the number of invariant operations in the inner loop and the number of loop iterations adds no information to the model. It does not mean that these parameters are not contributing to the performance time of the programs. It means that they are making no further contribution to the model. That is, any variance in the model that is due to these parameters has already been removed by other parameters. These parameters are not independent from the rest of the parameters.

Positive and negative coefficients are both contributions to the performance model. It is important to keep in mind that these coefficients do not represent the contribution

Parameter	*Coefficient*
Intercept	-0.97
Floating Point Adds	-1.27e-5
Floating Point Subtracts	-1.14e-5
Floating Point Multiplies	-1.43e-5
Floating Point Divides	-1.25e-5
Floating Point Exp	4.81e-6
Floating Point Sqrt	4.87e-6
Memory Operations	9.65e-6
Stride-1 Operations in Inner Loop	-6.92e-6
Non-Stride-1 Operations in Inner Loop	1.61e-7
Invariant Operations in Inner Loop	0.00
Loop Iterations	0.00

Table 6.1: Coefficients and Intercept for Sequential Performance Model

to the performance time from each parameter but the amount of variance in the model that is associated with this parameter. The coefficients, along with the parameters and the intercept point represent the best fit line in the linear model. With this in mind it is not surprising that some of the parameters are negative. The interdependence among the parameters and the intercept point in the model all have influence on the coefficients.

The R^2 value for the sequential model is 0.995. The residual plot is shown in Figure 6.1. The plot shows increasing magnitude of residuals as performance time increases. Since there is a linear relationship present rather than randomly scattered residuals, there are still some linear variables unaccounted for. Possible omissions are integer operations or operating system overhead that is independent of program parameters. The only points in the residual plot that have a significant residual value (more than 10% of the performance time) have very small performance times. In these cases any operating system overhead is a significant portion of the performance time. For this reason the model loses accuracy for programs with small execution times. We will limit the model to programs with performance times greater then one second. This is not actually much of a limitation because we are modeling scientific code which in general has performance times much greater than one second.

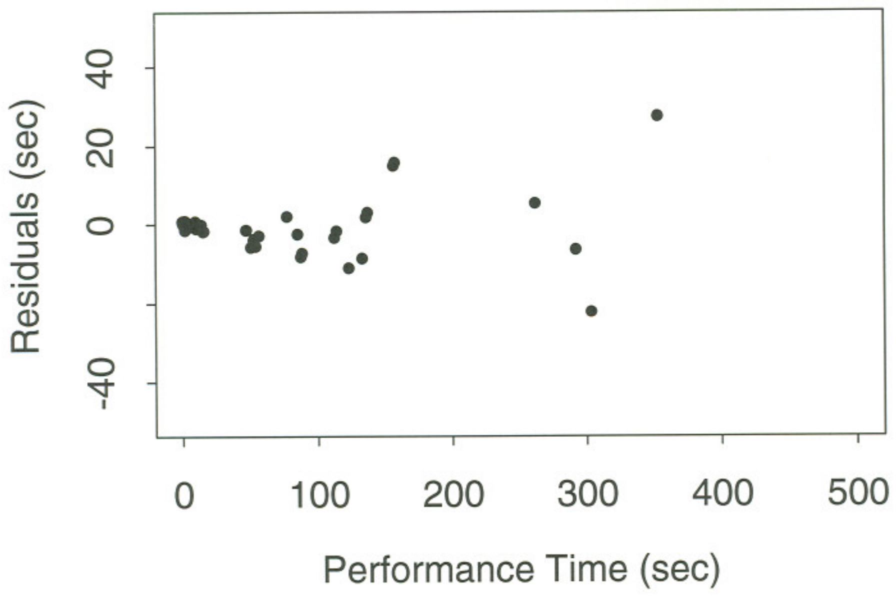

Figure 6.1: Sequential Performance Model

Parameter	*Coefficient*
Intercept	-0.48
Max Stride-1 Operations in Inner Loop	3.38e-6
Max Non-stride-1 Operations in Inner Loop	4.23e-6
Max Invariant Operations in Inner Loop	3.70e-6
Max Floating Point Adds	2.74e-6
Max Floating Point Subtracts	3.83e-6
Max Floating Point Multiplies	-6.92e-6
Max Floating Point Divides	1.62e-5
Max Floating Point Exp	2.72e-5
Max Floating Point Sqrt	3.80e-5
Memory Operations	0.00
Par Stride-1 Operations in Inner Loop	-1.36e-6
Par Non-stride-1 Operations in Inner Loop	-1.99e-6
Par Invariant Operations in Inner Loop	3.16e-6
Par Read	9.53e-7
Par Write	0.00
Parallel Forks	1.70e-4
Loop Iterations	0.00

Table 6.2: Coefficients and Intercept for Parallel Performance Model

6.2 Multiprocessor Results

The parallel performance model has a number of parameters in addition to those used in the sequential model. Table 6.2 shows the parameters and coefficients for the parallel model. Once again we find that the number of memory operations is not necessary in the parallel model. The same is true for the number of loop iterations. In addition, the number of writes to the same memory location did not contribute to the model. It was found that this parameter did not occur in any of the experiment set of programs. As discussed previously, multiple writes to the same memory location in a parallel loop violates data dependency relations. Other parameters that are specific to multi-processor execution were found to contribute to the model. For instance, the overhead of creating new processes to run the parallel loop executions does contribute to the model. Memory read contention was present in the experimental programs and did contribute to the model. Similarly, the stride in the inner loop of the parallel execution was found to be

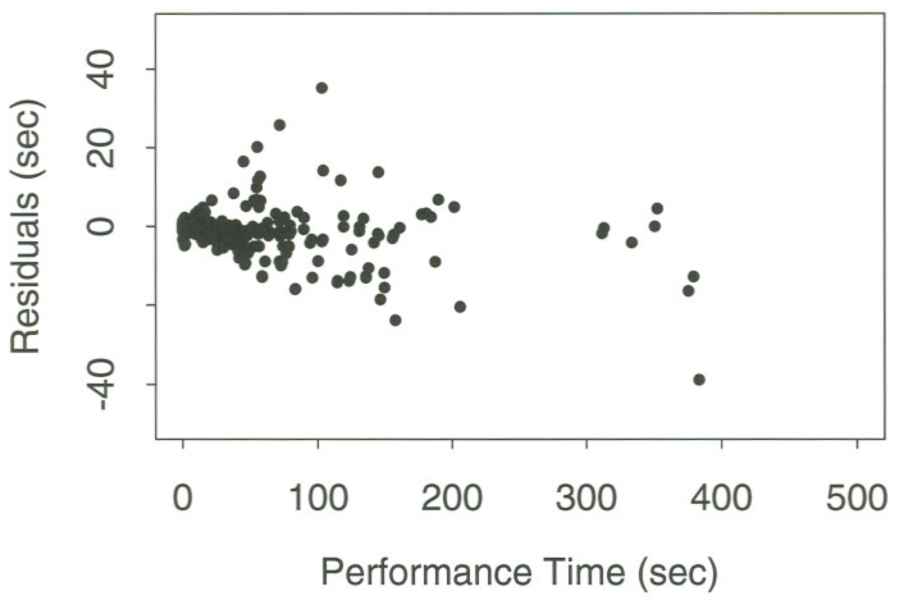

Figure 6.2: Parallel Performance Model

significant.

The R^2 value for this model is 0.985. This fit has lost some accuracy over relative to sequential model. The residual plot for the parallel model can be found in Figure 6.2. Once again we see a relationship between the magnitude of the residuals and the performance time. As mentioned previously, the most likely effect missing in the model is due to operating system overhead that is not covered by existing parameters. Residuals are again significant for programs with performance time less than one second. We will restrict the model to programs with performance time greater than one second.

The R^2 value for the parallel model shows some loss of accuracy over the sequential model. This is also evident in the residual plot. There are a small number of points with large residuals in the model that do not fall into the range of small performance times. Some residuals are up to 30% of the performance time for the program. Since this was not true in the sequential model it leads us to believe that an addition parameter is necessary to include more information about the parallel execution. Adding the number of processors as a parameter, however, was not found to improve the model. The parameter removes no additional variance and the fit remains the same in both the R^2 value and in the residuals. It does, however, partially mask the variance from other parameters, thus changing the coefficients. For this reason, the parameter was not included in the final model. The fact that the model is not improved with the additional parameter, suggests that if the variance is associated with the number of processors, the missing information does not have a linear relationship with respect to the number of processors.

In an early version of the model we found that by removing the number of processors as a variable would increase the accuracy of the model. This is done by creating a independent model for each number of processors. However, as the general model became more accurate by adjusting the parameters, there was no longer any noticeable advantage of separate models over one single model for all values of p. We built both the general model and the models for each value of P to compare. As mentioned before, the general model has a R^2 value of 0.985. The model for p=4 has a R^2 value of 0.978 and the R^2 value for the model with p=8 is 0.994. Figure 6.3 compares the residuals for the p=4 model with the residuals for the data points from executions with four processors in the general model. In this case both the R^2 value and the residual plot show a slight improvement over the general case. However, not all the models show improvement, Figure 6.4 compares the residuals for the p=8 model with the residuals for the data points with execution on 8 processors in the general model. Here both the R^2 and the residual plot show a slightly worse fit than in the general case. These two examples represent the two ends of the range of R^2 values. The models for the rest of the p values have R^2 values that fall within this range. Overall the models were very similar in fit to the general model. No information was lost and none was gained.

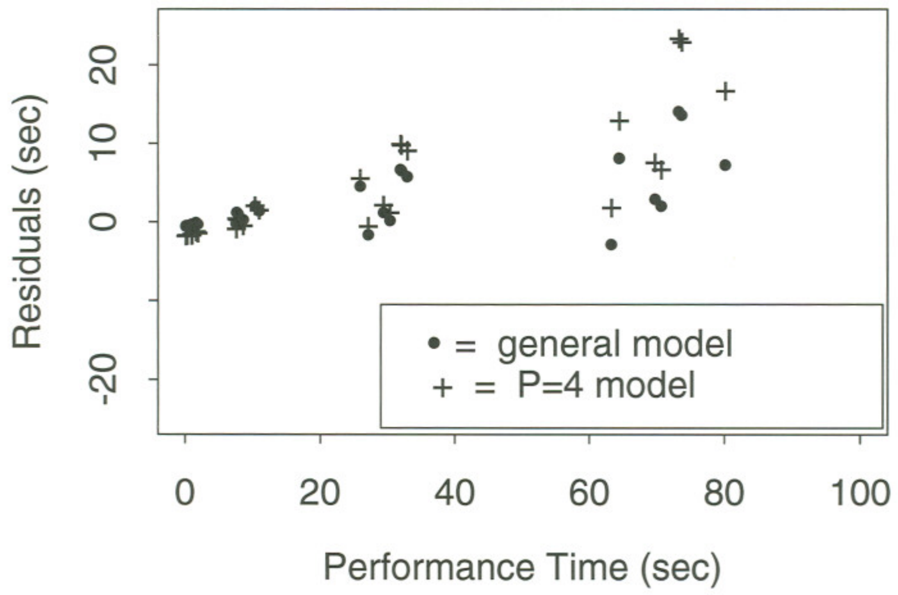

Figure 6.3: P=4 data for both General and P=4 Model

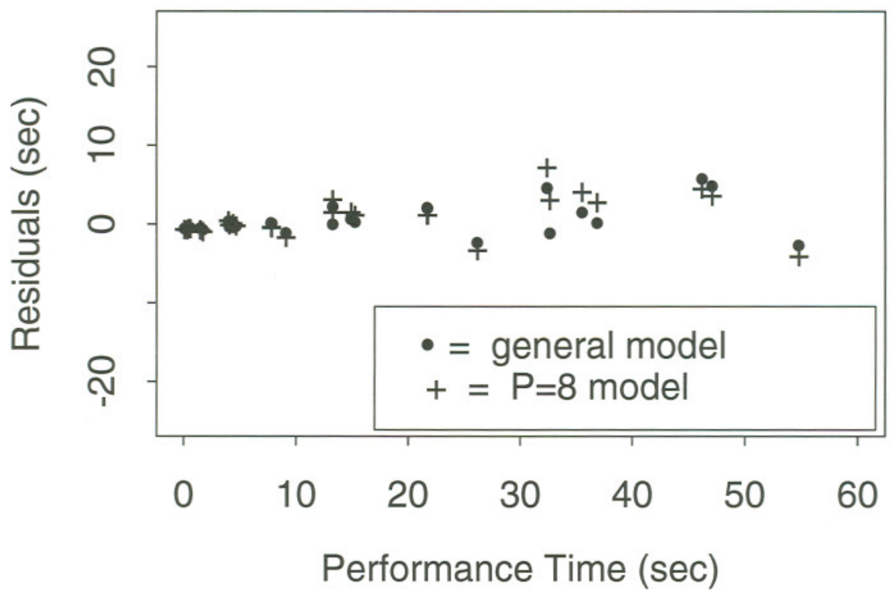

Figure 6.4: P=8 data for both General and P=8 Model

Program Version	*Performance Time*	*Predicted Performance*
K(IJ(I))	95.15	96.71
K(II(J))	105.09	119.61
K(I(J))	105.22	119.61
I(K(J))	104.77	119.69

Table 6.3: Performance Prediction Times for Sequential LU Decomposition, n = 300

6.3 Prediction Using Sequential Model

We applied the sequential performance model to the parameters measured for different versions of LU decomposition. Table 6.3 contains true performance times and predicted performance times for each version with constant problem size, n = 300. Only one version has a significantly different performance time. This is the case where memory accesses are following the cache line. All other versions have very similar performance times. The performance model was able to predict the optimal version of the program and the prediction time for this version was within 2% of the true performance time. The other versions were predicted less accurately (within 15%). The model does, however, accurately predict that the times for these versions are very close.

A more global picture, for all five versions and problem sizes (n = 50, 100, 200, 300, 400) of the program, can been seen in the residual plot, Figure 6.5. We disregard points with very low performance times because of limitations of the model mentioned above. The remaining points are each within 15% of the true performance time.

We use the same model to predict performance of two versions of our back-substitution program. The versions are listed in appendix C. It is clear from the residual plot, Figure 6.6, that the predicted performance values are not as accurate as in the LU decomposition program. In this case the performance times were predicted within 10 to 60 percent of the true performance times. Once again the smaller execution times had worse prediction results. In this case there is no clear optimal version of the program. Both the true and the predicted values are very similar for both versions. Table 6.4 shows the true performance times and the predicted times for both versions of the program and all values of n.

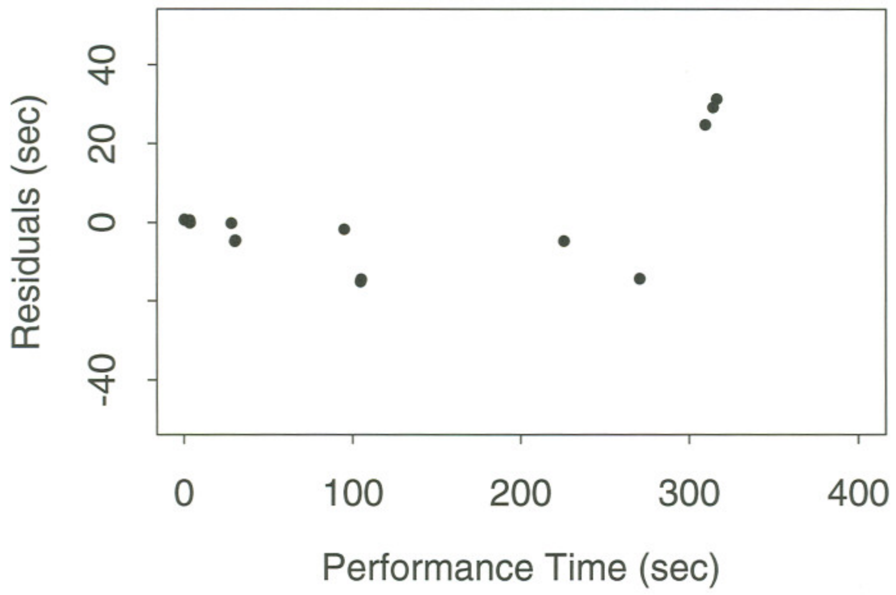

Figure 6.5: Sequential LU Decomposition Performance Prediction

Program Version	n	*Performance Time*	*Predicted Performance*
J(I)I(J)	300	2.01	0.75
J(I)J(I)	300	2.02	1.02
J(I)I(J)	400	3.62	2.09
J(I)J(I)	400	3.65	2.56
J(I)I(J)	500	5.63	3.80
J(I)J(I)	500	5.63	4.54
J(I)I(J)	600	8.21	5.90
J(I)J(I)	600	8.17	6.96

Table 6.4: Performance Prediction Times for Sequential Back-substitution

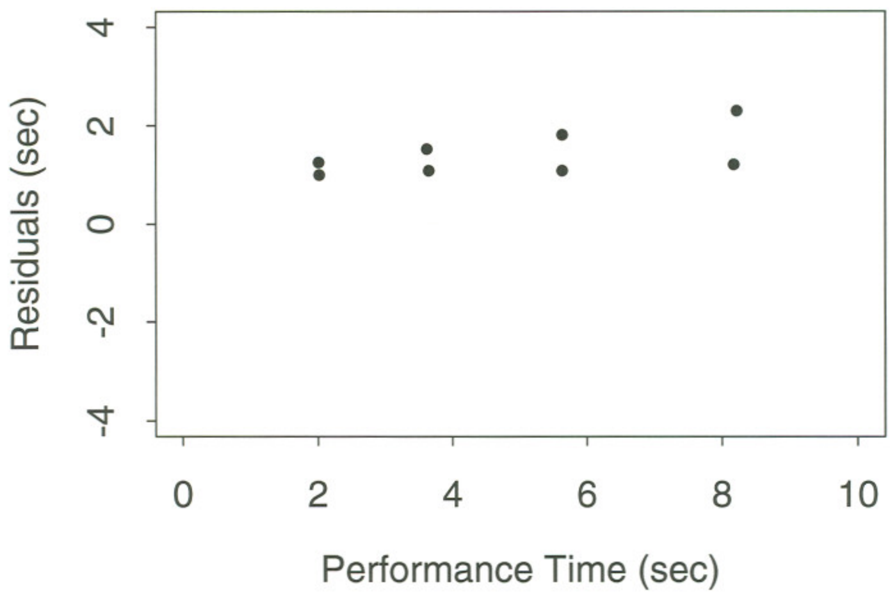

Figure 6.6: Sequential Back-substitution Performance Prediction

Program Version	*Performance Time*	*Predicted Performance*
K(IpJp(I))	63.26	60.42
K(Ip(J))	64.44	72.66
K(IpIp(J))	69.74	72.72
K(II(Jp))	73.25	87.31
K(I(Jp))	73.69	87.31
I(K(Jp))	80.16	87.49

Table 6.5: Performance Prediction Times for Parallel LU Decomposition, n=400, p=4

Program Version	*Performance Time*	*Predicted Performance*
K(IpJp(I))	121.49	118.12
K(Ip(J))	130.33	143.92
K(IpIp(J))	138.34	143.99
K(II(Jp))	140.94	158.04
K(I(Jp))	141.37	158.04
I(K(Jp))	141.63	158.22

Table 6.6: Performance Prediction Times for Parallel LU Decomposition, n=400, p=2

6.4 Prediction Using Parallel Model

The same program, LU decomposition, also has a number of parallel versions. The performance for each of these is predicted with the parallel performance model.

LU decomposition on multiple processors has seven versions with slightly more variation in performance times than the sequential versions. Table 6.5 contains true performance times and predicted performance times for a subset of these programs where the problem size and number of processors is held constant at p=4 and n=400. The predicted performance times correctly ranks the programs versions in order of performance. The predicted times are grouped into three sets with very similar times. The first set contains the version 5, the optimal version of the program. The second set contain versions 2, 4, and 6 with predicted performance times of about 72 seconds. The third set contains versions 1, 3, and 7 with performance times of about 87 seconds. Within the second and third set the actual performance times varied much more than the predicted times. The performance model has difficulty distinguishing between the performance of these versions.

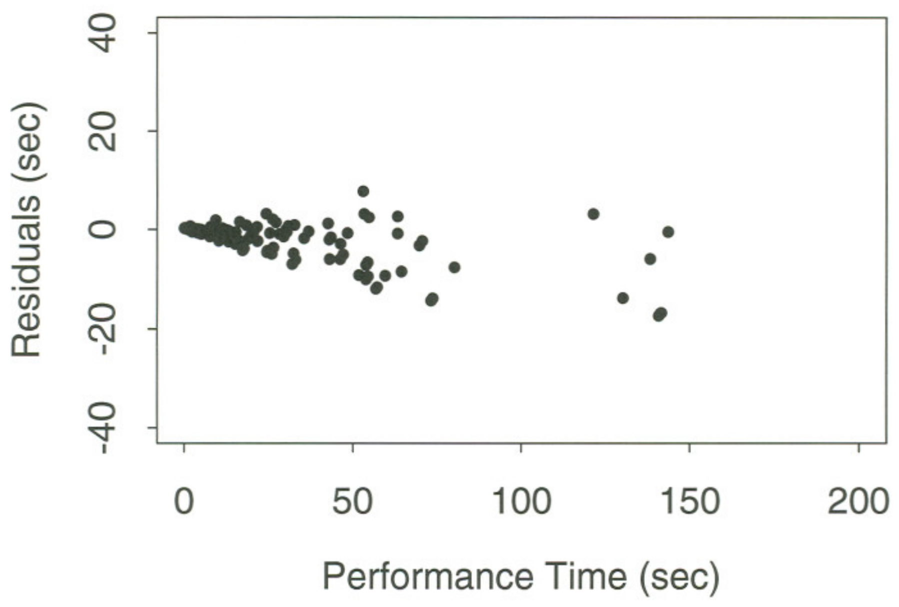

Figure 6.7: Parallel LU Decomposition Performance Prediction

Not all subsets of the data show as good results in distinguishing the best performance from different program versions. Table 6.6 extracts the actual performance times and the predicted performance times for the programs with p=2 and n=400. Here, the optimal program version is still predicted correctly. However, the predicted times on the rest of the versions is not accurate enough to correctly rank the performance of the program versions.

The performance model does a good job of predicting the decrease in performance time for increasing numbers of processors. Figure 6.8 shows the speedup of one version of LU decomposition when the number of processors increases. Also shown are the predicted decrease in performance. The performance values are very similar and shows the same trend in performance over differing numbers of processors.

The residual graph for all versions of LU decomposition can be found in Figure 6.7.

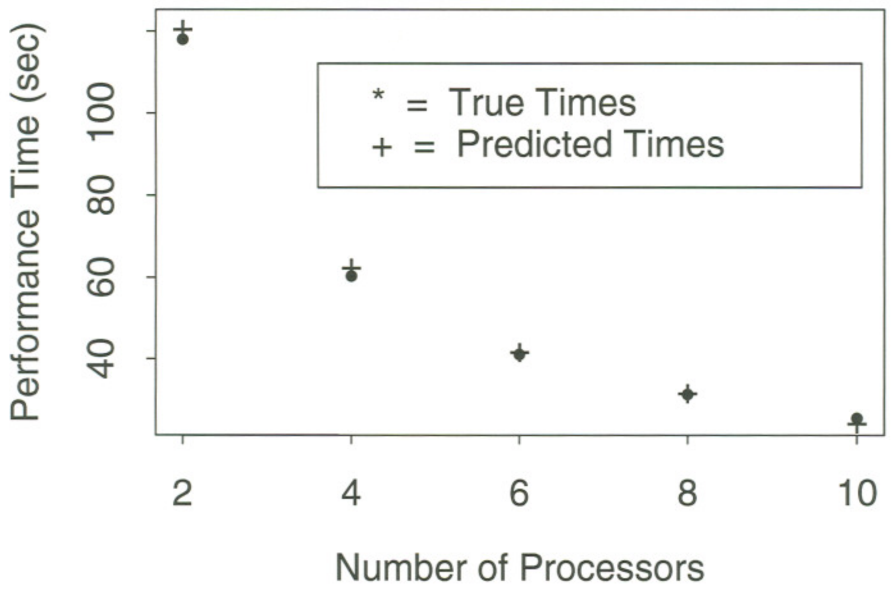

Figure 6.8: LU Decomposition Performance version K(I(Jp)) with Number of Processors (n = 500)

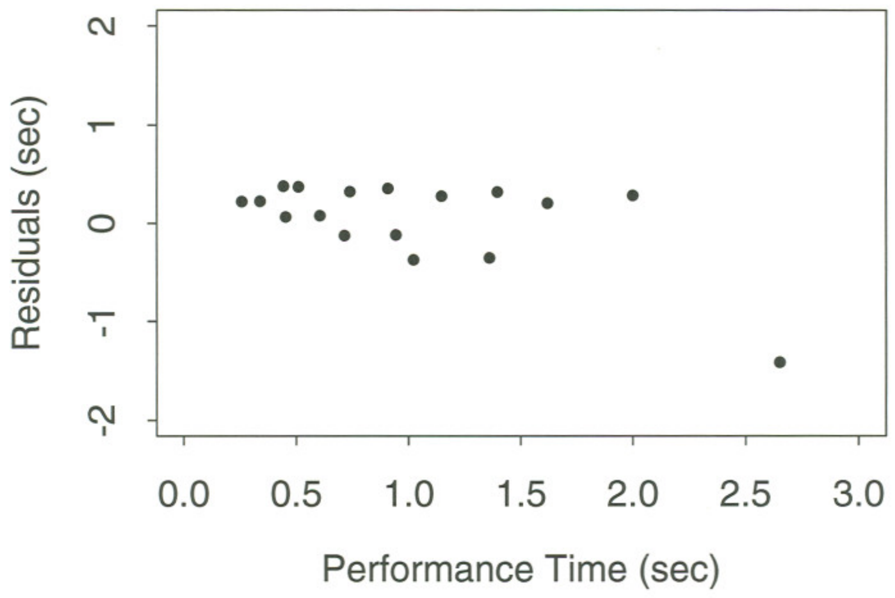

Figure 6.9: Parallel Back-substitution Performance Prediction

Again disregarding points with very low performance times, in general residuals are small (within 15%) of the performance times. However, as seen in the above examples, the granularity of the performance prediction is not always accurate enough to accurately differentiate between performance of closely related versions of the same program.

The back-substitution program has quite a bit more inconsistent prediction times when compared to true performance times. In Figure 6.9 the residuals are a greater percent of the performance time. In worst case, more than 100%. Part of the problem in this case is that the performance times were very small when the program was run on multiple processors. However, one point that sticks out in the graph has a performance time of greater than 2:30 secs. and a corresponding residual of about 1:30 secs. The times for all of the versions of the program along with all values of n and p can be found in Table 6.7. The Jp(I)Jp(I) version of the program has better performance than the

Program Version	*p*	*n*	*Performance Time*	*Predicted Performance*
Jp(I)Jp(I)	4	300	0.50	0.08
Jp(I)Ip(J)	4	300	0.67	0.08
Jp(I)Jp(I)	6	300	0.34	0.11
Jp(I)Ip(J)	6	300	0.51	0.13
Jp(I)Jp(I)	8	300	0.26	0.03
Jp(I)Ip(J)	8	300	0.44	0.06
Jp(I)Jp(I)	4	400	0.91	1.12
Jp(I)Ip(J)	4	400	1.18	1.12
Jp(I)Jp(I)	6	400	0.61	0.52
Jp(I)Ip(J)	6	400	0.91	0.54
Jp(I)Jp(I)	8	400	0.45	0.38
Jp(I)Ip(J)	8	400	0.74	0.41
Jp(I)Jp(I)	4	500	1.41	2.47
Jp(I)Ip(J)	4	500	1.85	2.44
Jp(I)Jp(I)	6	500	0.95	1.05
Jp(I)Ip(J)	6	500	1.40	1.07
Jp(I)Jp(I)	8	500	0.71	0.83
Jp(I)Ip(J)	8	500	1.15	0.86
Jp(I)Jp(I)	4	600	2.04	4.10
Jp(I)Ip(J)	4	600	2.65	4.06
Jp(I)Jp(I)	6	600	1.36	1.70
Jp(I)Ip(J)	6	600	2.00	1.71
Jp(I)Jp(I)	8	600	1.02	1.38
Jp(I)Ip(J)	8	600	1.62	1.41

Table 6.7: Performance Prediction Times for Parallel Back-substitution

Jp(I)Ip(J) version. Although the predicted performance times are not accurate in many cases, they are consistently less than the times for the Jp(I)Jp(I) version.

A complete listing of performance times and predictions times for all versions of programs in the experimental set over all values of n and p are included in Appendix D.

Chapter 7

Discussion

A number of simplifying assumptions were made that restrict the domain of the performance model. These restrictions were made because of limitations in resources and in order to focus the scope of the research project. The scope includes scientific code executed in an environment that is not restricted because of multiple users sharing cycle time.

7.1 Effects of System Load

One restriction in our model is due to the fact that processes created in a parallel loop construct on the Sequent Symmetry are not necessarily scheduled on different processors. The Sequent creates processes which will then be scheduled on available processors. If enough processors are available, each process will each be scheduled on a different processor and will be executed in parallel. The Sequent Symmetry is a time sharing system and thus processes must compete with processes scheduled by other applications and other users. The operating system overhead of context switch time, time spent in the wait queue, overhead from being swapped out of memory, etc., is indeterminate and could cause performance times to be significantly worse than predicted times. In the realm of this study the load on the system was small and thus each process was always scheduled to its own processor. This greatly increases the predictability of performance for a given code segment.

In most cases this simplification should have no effect on the use of the performance prediction tool as long as the load on the system is consistent during the use of the tool. The use of performance prediction to evaluate the optimal loop transformation will not depend on the load of the system. A loop formulation that is superior to an alternate formulation in most cases will remain optimal even with the additional overhead added by a more heavily loaded system. The reason that this is true is based on the fact that course grained parallelism is optimal on even a lightly loaded system because of the overhead included in forking a parallel process. A heavily loaded system would cause more overhead per process and thus provide better performance for coarsely grained parallel constructs. Since this is already the case, it would not effect the evaluation of optimal loop construct. If the load on the system is not consistent while predicting transformations of a loop then one loop formulation might artificially perform better or worse than another while the difference is caused only by a decrease in parallelism due to processes spending time on the wait queue.

7.2 Conditionals

There has been little discussion of the effect of conditional branches on the performance prediction model. In this model, conditionals of the form if/then/else were analyzed including both paths. This could clearly cause an overestimate of the performance time. A better algorithm would be to predict the branch taken. Clark and Levy [CL82] measured simple conditional branches were taken about 50% of the time. Hennessy and Patterson [PH90] measured branches taken 53% of the time. Clearly on the average, predicting the branch taken is as good as you can do without a more elaborate branch prediction scheme. The number of executions of a loop body was known in this analysis. Thus loop branches did not need to analyzed. The number of executions of a loop can often be determined at compile time with analysis such as constant propagation.

7.3 Coefficients

It is important to understand what the coefficients in the linear performance model represent. It is tempting to interpret them as weights for the parameters. In this way a parameter with a higher coefficient would have more contribution to the performance time. If the parameters were completely independent, this would be the case. However, the parameters are not at all independent. For example, the number of memory accesses is not independent from the number of floating point operations because the floating point results are stored in memory and the operands are read from memory. Indeed, the number of floating point multiplies is not necessarily independent from the number of floating point adds. If a program contains a loop that includes two multiplies and one add, the total number of multiplies and adds are a multiple of the number of times the loop is executed. In this case the number of multiplies will also be twice the number of adds. This is certainly not independent.

If two parameters share some variance, that is, if they are dependent on each other, the coefficient for one parameter might carry this shared variance. Thus the second parameter will have a smaller coefficient than it would if the first had not been present. Much of the relationship between the parameter and the performance time is included in other factors. In the extreme case a parameter adds no additional information to the model and the coefficient is zero. Although a relationship may exist between the parameter and the performance time, that relationship adds no information that is not already included in other factors.

7.4 Compilers

We intend for this model to be used in a compiler independent way. Of course, realistically, different compilers and even different optimizations performed by the same compiler can effect the performance time of the program. The parameters are measured before the code is compiled so any optimization done by the compiler can influence the performance time relative to these parameters. In these models we held these variables constant by using the same compiler with the same optimizations active.

To account for differences of compilers and optimizations, some information on the compiler optimizations that will be used by the compiler must be included. This information can be included as additional parameters in the model or different versions of the model could be built to handle different compiler optimizations. Alternatively, the parameters could be measured by the compiler after optimizations have taken place.

7.5 Tiny

The Tiny language is a reduced language from more typical imperative languages. Tiny was used to analyze the programs and collect data for the parameters. The programs included were originally written in Tiny and translated by Tiny into Fortran. This restricted the programs to those that Tiny could understand. One restriction imposed by Tiny is that program size must be limited to small programs. Tiny also doesn't understand subroutine or function calls, so our programs were limited to those that did computation without calling other functions.

7.6 Other Systems

Although we restricted this work to the shared memory Sequent system, it is equally applicable to other shared memory systems as well as vector processors and sequential machines. The parameters used to create this model should work as well on these other systems. The only way that they are specific to an architecture is that the parameters were chosen to cover all architectural features that effect performance of a program. If, for example, we wished to model a vector processor, other parameters would need to be included to cover that feature such as the number of vector operations and the stride of vector loads and stores. If the architectural features are similar, the only requirement to port the performance model is to collect the data for the experimental set of programs on each machine and perform the linear fit to create the model.

In addition, the method is easily adapted to distributed memory machines. Additional parameters may be needed however, to include information on data distribution and communication.

Chapter 8

Future Work

In the previous section some restrictions to the scope of this research were discussed. With these restrictions in mind, this section discusses possible ways to expand this research in order to overcome these limitations.

8.1 Increase Scope of Programs

One of the underlying restriction was the fact that programs had to be read by the Tiny tool which both counted the parameters and performed loop transformations. If this analysis could be performed using a tool with a larger scope, such as a compiler, more programs could be considered for inclusion in the performance and prediction set of programs.

Additional questions will arise by incorporating the work in a compiler. For instance, how will subroutine and function calls be treated in the performance parameters? Should the compiler provide feedback to the user about which compiler optimizations were performed in order to disambiguate the performance gain do to compiler optimizations over loop restructuring? Also, the Tiny tool would not be available to perform loop restructuring. It will have to be done by hand unless the compiler has that ability built in.

With a larger pool of programs to choose from, it should be unnecessary to include the programs in the Other category of the prediction set. These programs were included because real programs could not be found to cover all the characteristics necessary to cover a wide variety of programs. Since these Other programs were created in order to artificially contribute to the range of programs that are represented by the model, they may have affected the coefficients in a way that might not be representative of real programs. It should be possible to choose programs from various benchmark sets that well represent most scientific programs without included any programs created by hand.

8.2 Multiple Machines

This research was restricted to the Sequent Symmetry, a shared memory multiprocessing system. Further work should include both other shared memory systems and systems with different performance characteristics, such as vector processors and distributed memory systems. The model itself will not port to different systems, however, the methodology should port easily. If the new system does not have any additional performance characteristics, the same parameters may apply to the new model. The prediction set of programs must be run on the new system and the linear model recalculated. As mentioned previously, if the system includes characteristics that affect the performance time that were not present in this work, then additional parameters must be added to include information about the effect of the system on the performance of programs. For example, if the system includes vector processors, the stride of the vector loads and stores may be included as well as the number of vector operations. These programs must be reanalyzed to include all the parameters and the run on the system in order to build the new linear model.

8.3 Other Studies

A study might be carried out in order to measure the effects of operating system overhead on performance on a time-shared system. It would also be beneficial to include some information in the model in order to parameterize these effects. However, these effects can not be measured statically.

Another study might investigate the effect of conditionals on performance prediction.

It may be found that scientific code does not follow the 50% taken rule mentioned above. It is possible that conditional branches are often never taken because they check for error conditions that rarely occur.

At the least, a simple branch prediction scheme should be implemented in order to measure parameters more accurately for code that contains branches. Possibly a more accurate prediction scheme can be determined from the study.

Appendix A

Cholesky Decomposition

In this section we will demonstrate multiple versions Cholesky Decomposition and how they are derived from the base version (KIJ form). These transformations were performed using the Tiny tool.

Version 1: K(I(J)) form.
The original program is a Cholesky Decomposition.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  for i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
    for j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 2: K(I(Jp)) form.
Parallelize the J loop.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  for i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
    doall j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 3: K(II(J)) form.
Go back to serial form. Distribute I loop.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  for i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
  endfor
  for i = k+1,n do
    for j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 4: K(IpI(Jp)) form.
Parallelize both the first I loop and the inner J loop.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  doall i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
  endfor
  for i = k+1,n do
    doall for j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 5: K(IpIp(J)) form.
Parallelize both I loops.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  doall i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
  endfor
  doall i = k+1,n do
    for j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 6: K(IJ(I)) form.
Go back to serial form. Interchange J and I loops.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  for i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
  endfor
  for j = k+1,n do
    for i = max(k+1,j),n do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 7: K(IpJ(Ip)) form.
Parallelize both I loops.

```
real a(n,n)
integer n
for k = 1,n do
  a(k,k) = sqrt(a(k,k))
  doall i = k+1,n do
    a(i,k) = a(i,k)/a(k,k)
  endfor
  for j = k+1,n do
    doall i = max(k+1,j),n do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
endfor
```

Version 8: J(K(I)I) form.
Interchange J and K loops. Notice that J and K are NOT tightly nested!

```
real a(n,n)
integer n
for j = 1,n do
  for k = 1,j-1 do
    for i = max(k+1,j),n do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(j,j) = sqrt(a(j,j))
  for i = j+1,n do
    a(i,j) = a(i,j)/a(j,j)
  endfor
endfor
```

Version 9: J(K(Ip)Ip) form.
Parallelize both I loops.

```
real a(n,n)
integer n
for j = 1,n do
  for k = 1,j-1 do
    doall i = max(k+1,j),n do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(j,j) = sqrt(a(j,j))
  doall i = j+1,n do
    a(i,j) = a(i,j)/a(j,j)
  endfor
endfor
```

Version 10: J(I(K)I) form.
Go back to serial version. Interchange I and K loops.

```
real a(n,n)
integer n
for j = 1,n do
  for i = max(1+1,j),n do
    for k = 1,min(j-1,i-1) do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(j,j) = sqrt(a(j,j))
  for i = j+1,n do
    a(i,j) = a(i,j)/a(j,j)
  endfor
endfor
```

Version 11: J(Ip(K)Ip) form.
Parallelize both I loops.

```
real a(n,n)
integer n
for j = 1,n do
  doall i = max(1+1,j),n do
    for k = 1,min(j-1,i-1) do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(j,j) = sqrt(a(j,j))
  doall i = j+1,n do
    a(i,j) = a(i,j)/a(j,j)
  endfor
endfor
```

Version 12: I(K(J)) form.
Go back to original program: Interchange I and K loops (again, they are NOT tightly nested).

```
real a(n,n)
integer n
for i = 1,n do
  for k = 1,i-1 do
    a(i,k) = a(i,k)/a(k,k)
    for j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(i,i) = sqrt(a(i,i))
endfor
```

Version 13: I(K(Jp)) form.
Parallelize J loop.

```
real a(n,n)
integer n
for i = 1,n do
  for k = 1,i-1 do
    a(i,k) = a(i,k)/a(k,k)
    doall j = k+1,i do
      a(i,j) = a(i,j)-a(i,k)*a(j,k)
    endfor
  endfor
  a(i,i) = sqrt(a(i,i))
endfor
```

Version 14: I(J(K)) form.
Interchange J and K loops (again, non-tightly nested).

```
real a(n,n)
integer n
for i = 1,n do
  for j = 1,i-1 do
    a(i,j) = a(i,j)/a(j,j)
    for k = 1,j do
      a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
    endfor
  endfor
  a(i,i) = sqrt(a(i,i))
endfor
```

Version 15: J(I(K)) form.
Finally, interchange J and I and get J(I(K)) form slightly different from J(I(K)I).

```
real a(n,n)
integer n
for j = 1,n do
  a(j,j) = sqrt(a(j,j))
  for i = j+1,n do
    a(i,j) = a(i,j)/a(j,j)
    for k = 1,j do
      a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
    endfor
  endfor
endfor
```

Appendix B

Prediction Set

The following Fortran programs are given in their basic forms. Analysis similar to that done in Appendix A was performed in order to form multiple transformations of the programs. Each transformation was also performed with varying values for n ranging from n= 50 to n = 500. In addition, parallel loops were also performed with varying numbers of processors. P varied from 2 to 10.

Cholesky Decomposition (**K(I(J)) form**):

```
       subroutine chol(n,a)
       real a(n, n)
       do 10 k = 1,n
        a(k,k) = sqrt(a(k,k))
        do 20 i = k+1,n
         a(i,k) = a(i,k)/a(k,k)
         do 30 j = k+1,i
          a(i,j) = a(i,j)-a(i,k)*a(j,k)
 30      continue
 20     continue
 10    continue
       end
```

Cholesky Decomposition (K(Ip(J)) form):

```
       subroutine chol(n,a)
       real a(n, n)
       do 10 k = 1,n
        a(k,k) = sqrt(a(k,k))
        do 20 i = k+1,n
         a(i,k) = a(i,k)/a(k,k)
c$doacross
         do 30 j = k+1,i
          a(i,j) = a(i,j)-a(i,k)*a(j,k)
   30    continue
   20   continue
   10  continue
       end
```

Matrix Multiply (I(J(K)) form):

```
       subroutine mm(n,c,b,a)
       real a(n,n)
       real b(n,n)
       real c(n,n)
       integer n
       do 10 i = 1,n
        do 20 j = 1,n
         do 30 k = 1,n
          c(i,j) = c(i,j)+a(i,k)*b(k,j)
   30    continue
   20   continue
   10  continue
       end
```

Matrix Multiply (Ip(J(K)) form):

```
      subroutine mm(n,c,b,a)
      real a(n,n)
      real b(n,n)
      real c(n,n)
      integer n
c$doacross
      do 10 i = 1,n
       do 20 j = 1,n
        do 30 k = 1,n
         c(i,j) = c(i,j)+a(i,k)*b(k,j)
   30   continue
   20  continue
   10 continue
      end
```

Addition (I(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)+a(i,j)
   20  continue
   10 continue
      end
```

Addition (Ip(J) form):

```
       subroutine other2(n,a)
       integer n
       real a(n,n)
c$doacross
       do 10 i = 1,n
        do 20 j = 1,n
         a(i,j) = a(i,j)+a(i,j)
   20   continue
   10  continue
       end
```

Subtraction (I(J) form):

```
       subroutine other1(n,a)
       integer n
       real a(n,n)
       do 10 i = 1,n
        do 20 j = 1,n
         a(i,j) = a(i,j)-a(i,j)
   20   continue
   10  continue
       end
```

Subtraction (Ip(J) form):

```
       subroutine other2(n,a)
       integer n
       real a(n,n)
c$doacross
       do 10 i = 1,n
        do 20 j = 1,n
         a(i,j) = a(i,j)-a(i,j)
   20   continue
   10  continue
       end
```

Multiply (I(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)*a(i,i)
   20  continue
   10 continue
      end
```

Multiply (Ip(J) form):

```
      subroutine other2(n,a)
      integer n
      real a(n,n)
c$doacross
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)*a(i,i)
   20  continue
   10 continue
      end
```

Divide (I(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)/a(i,i)
   20  continue
   10 continue
      end
```

Divide (Ip(J) form):

```
      subroutine other2(n,a)
      integer n
      real a(n,n)
c$doacross
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)/a(i,i)
   20  continue
   10 continue
      end
```

Exponent (I(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)**0.5
   20  continue
   10 continue
      end
```

Exponent (Ip(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
c$doacross
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = a(i,j)**0.5
   20  continue
   10 continue
      end
```

Square Root (I(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = sqrt(a(i,j))
   20  continue
   10 continue
      end
```

Square Root (Ip(J) form):

```
      subroutine other1(n,a)
      integer n
      real a(n,n)
c$doacross
      do 10 i = 1,n
       do 20 j = 1,n
        a(i,j) = sqrt(a(i,j))
   20  continue
   10 continue
      end
```

Appendix C

Experimental Set

The following programs are given in their basic forms. Analysis similar to that done in Appendix A was performed in order to form multiple transformations of the programs. Each transformation was also performed with varying values for n ranging from n= 50 to n = 500. In addition, parallel loops were also performed with varying numbers of processors. P varied from 2 to 10.

LU Decomposition (**K(I(J))** form):

```
      subroutine lu(n,a)
      real a(n,n)
      integer n
      do 10 k = 1,n
       do 20 i = k+1,n
        a(i,k) = a(i,k)/a(k,k)
        do 30 j = k+1,n
         a(i,j) = a(i,j)-a(k,j)*a(i,k)
 30     continue
 20    continue
 10   continue
      end
```

LU Decomposition (K(I(Jp)) form):

```
      subroutine lu(n,a)
      real a(n,n)
      integer n
      do 10 k = 1,n
       do 20 i = k+1,n
        a(i,k) = a(i,k)/a(k,k)
c$doacross
        do 30 j = k+1,n
         a(i,j) = a(i,j)-a(k,j)*a(i,k)
   30   continue
   20  continue
   10 continue
      end
```

Back-substitution (**J(I)J(I)** form):

```
      subroutine bksub1(u,v,b,w,x,n)
      real u(n,n)
      real v(n,n)
      real b(n)
      real w(n)
      real x(n)
      integer n
      do 10 j = 1,n
       if ( w(j).gt.0 ) then
        do 20 i = 1,n
         x(j) = x(j)+u(i,j)*b(i)
20      continue
       end if
10    continue
      do 30 j = 1,n
       do 40 i = 1,n
        x(j) = x(j)+v(j,i)*x(j)
40     continue
30    continue
      end
```

Back-substitution (**Jp(I)Jp(I)** form):

```
      subroutine bksub1(u,v,b,w,x,n)
      real u(n,n)
      real v(n,n)
      real b(n)
      real w(n)
      real x(n)
      integer n
c$doacross
      do 10 j = 1,n
       if ( w(j).gt.0 ) then
        do 20 i = 1,n
         x(j) = x(j)+u(i,j)*b(i)
   20   continue
       end if
   10 continue
c$doacross
      do 30 j = 1,n
       do 40 i = 1,n
        x(j) = x(j)+v(j,i)*x(j)
   40  continue
   30 continue
      end
```

Appendix D

Prediction Data

LU Decomposition			
Program Version	n	*Performance Time*	*Predicted Performance*
K(I(J))	100	3.68	3.51
K(I(J))	200	30.40	34.77
K(I(J))	300	105.22	119.61
K(I(J))	400	314.22	284.79
I(K(J))	100	3.70	3.52
I(K(J))	200	30.20	34.81
I(K(J))	300	104.77	119.69
I(K(J))	400	270.73	284.94
K(IJ(I))	100	3.42	2.66
K(IJ(I))	200	28.01	28.00
K(IJ(I))	300	95.15	96.70
K(IJ(I))	400	225.93	230.44
K(II(J))	100	3.70	3.51
K(II(J))	200	30.49	34.77
K(II(J))	300	105.09	119.61
K(II(J))	400	309.70	284.79

Table D.1: Sequential Performance Times (1 of 2)

Back-substitution			
Program Version	n	*Performance Time*	*Predicted Performance*
K(I(J))	100	3.68	3.51
J(I)J(I)	300	2.02	1.02
J(I)J(I)	400	3.65	2.56
J(I)J(I)	500	5.63	4.54
J(I)J(I)	600	8.17	6.96
J(I)I(J)	300	2.01	0.75
J(I)I(J)	400	3.62	2.09
J(I)I(J)	500	5.63	3.80
J(I)I(J)	600	8.21	5.90

Table D.2: Sequential Performance Times (2 of 2)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
K(I(Jp))	2	100	2.42	2.70
K(I(Jp))	2	200	17.31	21.22
K(I(Jp))	2	300	57.15	68.52
K(I(Jp))	2	400	141.37	158.04
K(I(Jp))	4	100	1.67	1.61
K(I(Jp))	4	200	10.34	12.42
K(I(Jp))	4	300	32.06	38.72
K(I(Jp))	4	400	73.69	87.31
K(I(Jp))	6	100	1.49	1.24
K(I(Jp))	6	200	8.42	9.48
K(I(Jp))	6	300	24.63	28.79
K(I(Jp))	6	400	54.54	63.74
K(I(Jp))	8	100	1.49	1.07
K(I(Jp))	8	200	7.82	8.01
K(I(Jp))	8	300	21.73	23.82
K(I(Jp))	8	400	47.10	51.95
K(I(Jp))	10	100	1.62	0.95
K(I(Jp))	10	200	7.93	7.13
K(I(Jp))	10	300	21.49	20.84
K(I(Jp))	10	400	43.50	44.88

Table D.3: Parallel Performance Times (1 of 7)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
K(Ip(J))	2	100	1.87	1.89
K(Ip(J))	2	200	15.12	17.84
K(Ip(J))	2	300	51.83	60.73
K(Ip(J))	2	400	130.33	143.92
K(Ip(J))	4	100	0.98	0.76
K(Ip(J))	4	200	7.64	8.89
K(Ip(J))	4	300	25.96	30.63
K(Ip(J))	4	400	64.44	72.66
K(Ip(J))	6	100	0.68	0.38
K(Ip(J))	6	200	5.16	5.92
K(Ip(J))	6	300	17.43	20.60
K(Ip(J))	6	400	43.13	48.91
K(Ip(J))	8	100	0.54	0.20
K(Ip(J))	8	200	3.99	4.43
K(Ip(J))	8	300	13.26	15.58
K(Ip(J))	8	400	32.39	37.03
K(Ip(J))	10	100	0.45	0.05
K(Ip(J))	10	200	3.27	3.53
K(Ip(J))	10	300	10.74	12.57
K(Ip(J))	10	400	26.45	29.90

Table D.4: Parallel Performance Times (2 of 7)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
I(K(Jp))	2	100	2.53	2.72
I(K(Jp))	2	200	17.69	21.27
I(K(Jp))	2	300	59.56	68.62
I(K(Jp))	2	400	141.63	158.22
I(K(Jp))	4	100	1.85	1.62
I(K(Jp))	4	200	10.10	12.46
I(K(Jp))	4	300	32.93	38.82
I(K(Jp))	4	400	80.16	87.49
I(K(Jp))	6	100	1.66	1.26
I(K(Jp))	6	200	10.05	9.52
I(K(Jp))	6	300	28.14	28.89
I(K(Jp))	6	400	63.30	63.91
I(K(Jp))	8	100	1.72	1.07
I(K(Jp))	8	200	9.08	8.06
I(K(Jp))	8	300	26.17	23.92
I(K(Jp))	8	400	54.77	52.13
I(K(Jp))	10	100	1.82	0.97
I(K(Jp))	10	200	9.28	7.18
I(K(Jp))	10	300	24.30	20.94
I(K(Jp))	10	400	53.07	45.05

Table D.5: Parallel Performance Times (3 of 7)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
K(IpJp(I))	2	100	2.02	1.51
K(IpJp(I))	2	200	16.42	14.69
K(IpJp(I))	2	300	53.31	49.93
K(IpJp(I))	2	400	121.49	118.13
K(IpJp(I))	4	100	0.99	0.60
K(IpJp(I))	4	200	7.54	7.44
K(IpJp(I))	4	300	27.13	25.54
K(IpJp(I))	4	400	63.26	60.43
K(IpJp(I))	6	100	0.65	0.30
K(IpJp(I))	6	200	5.13	5.02
K(IpJp(I))	6	300	18.41	17.42
K(IpJp(I))	6	400	42.63	41.19
K(IpJp(I))	8	100	0.49	0.14
K(IpJp(I))	8	200	4.06	3.81
K(IpJp(I))	8	300	13.26	13.35
K(IpJp(I))	8	400	32.65	31.58
K(IpJp(I))	10	100	0.41	0.05
K(IpJp(I))	10	200	3.38	3.08
K(IpJp(I))	10	300	11.33	10.91
K(IpJp(I))	10	400	25.32	25.81

Table D.6: Parallel Performance Times (4 of 7)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
K(IpIp(J))	2	100	1.93	1.90
K(IpIp(J))	2	200	15.96	17.87
K(IpIp(J))	2	300	54.39	60.79
K(IpIp(J))	2	400	138.34	143.99
K(IpIp(J))	4	100	1.00	0.78
K(IpIp(J))	4	200	7.82	8.93
K(IpIp(J))	4	300	29.41	30.69
K(IpIp(J))	4	400	69.74	72.73
K(IpIp(J))	6	100	0.75	0.40
K(IpIp(J))	6	200	5.41	5.95
K(IpIp(J))	6	300	19.54	20.65
K(IpIp(J))	6	400	46.29	48.98
K(IpIp(J))	8	100	0.55	0.21
K(IpIp(J))	8	200	4.38	4.46
K(IpIp(J))	8	300	14.90	15.63
K(IpIp(J))	8	400	35.52	37.10
K(IpIp(J))	10	100	0.48	0.10
K(IpIp(J))	10	200	3.49	3.57
K(IpIp(J))	10	300	11.60	12.63
K(IpIp(J))	10	400	30.09	29.97

Table D.7: Parallel Performance Times (5 of 7)

LU Decomposition				
Program Version	p	n	*Performance Time*	*Predicted Performance*
K(II(Jp))	2	100	2.41	2.70
K(II(Jp))	2	200	17.23	21.22
K(II(Jp))	2	300	56.85	68.52
K(II(Jp))	2	400	140.94	158.04
K(II(Jp))	4	100	1.65	1.61
K(II(Jp))	4	200	10.31	12.42
K(II(Jp))	4	300	31.96	38.72
K(II(Jp))	4	400	73.25	87.31
K(II(Jp))	6	100	1.48	1.25
K(II(Jp))	6	200	8.37	9.48
K(II(Jp))	6	300	24.42	28.79
K(II(Jp))	6	400	53.95	63.74
K(II(Jp))	8	100	1.48	1.06
K(II(Jp))	8	200	7.78	8.01
K(II(Jp))	8	300	21.68	23.82
K(II(Jp))	8	400	46.18	51.95
K(II(Jp))	10	100	1.58	0.96
K(II(Jp))	10	200	7.76	7.13
K(II(Jp))	10	300	20.74	20.84
K(II(Jp))	10	400	43.05	44.88

Table D.8: Parallel Performance Times (6 of 7)

Back-substitution				
Program Version	p	n	*Performance Time*	*Predicted Performance*
Jp(I)Jp(I)	4	300	0.50	0.08
Jp(I)Jp(I)	4	400	0.91	1.12
Jp(I)Jp(I)	4	500	1.41	2.47
Jp(I)Jp(I)	4	600	2.04	4.10
Jp(I)Jp(I)	6	300	0.34	0.11
Jp(I)Jp(I)	6	400	0.61	0.52
Jp(I)Jp(I)	6	500	0.95	1.05
Jp(I)Jp(I)	6	600	1.36	1.70
Jp(I)Jp(I)	8	300	0.26	0.03
Jp(I)Jp(I)	8	400	0.45	0.38
Jp(I)Jp(I)	8	500	0.71	0.83
Jp(I)Jp(I)	8	600	1.02	1.38
Jp(I)Ip(J)	4	300	0.67	0.08
Jp(I)Ip(J)	4	400	1.18	1.12
Jp(I)Ip(J)	4	500	1.85	2.44
Jp(I)Ip(J)	4	600	2.65	4.06
Jp(I)Ip(J)	6	300	0.51	0.13
Jp(I)Ip(J)	6	400	0.91	0.54
Jp(I)Ip(J)	6	500	1.40	1.07
Jp(I)Ip(J)	6	600	2.00	1.71
Jp(I)Ip(J)	8	300	0.44	0.06
Jp(I)Ip(J)	8	400	0.74	0.41
Jp(I)Ip(J)	8	500	1.15	0.86
Jp(I)Ip(J)	8	600	1.62	1.41

Table D.9: Parallel Performance Times (7 of 7)

Bibliography

[BCW88] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics.* Wadsworth and Brooks, Pacific-Grove, California, 1988.

[BR89] Sandra Johnson Baylor and Bharat Deep Rathi. A study of memory refence behavior of engineering/scientific applications in parallel processors. In *International Conference on Parallel Processing*, pages 78–82, 1989.

[CL82] D. W. Clark and H. Levy. Measurement and analysis of instruction set use in the VAX A/780. In *Proceedings of the Ninth Symposium on Computer Architecture*, pages 188–200, April 1982.

[Das89] Subrata Dasgupta. *Computer Architecture: A Modern Synthesis.* John Wiley and Sons, Inc., 1989.

[DG84] William R. Dillon and Matthew Goldstein. *Multivariate Analysis: Methods and Applications.* John Wiley and Sons. Inc., Toronto, Canada, 1984.

[DI88] R. T. Dimpsey and R. K. Iyer. Performance analysis of a shared memory multiprocessor: Case study. In *Proceedings International Conference on Parallel Processing*, pages 174–181, August 1988.

[DI89] R. T. Dimpsey and R. K. Iyer. Multiprogramming performance degradation: Case study on a shared memory multiprocessor. In *International Conference on Parallel Processing*, pages 205–208, 1989.

[GJG88] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformantion. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[KME92] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A methodology for performance analysis of parallel computations with looping constructs. *Journal of Parallel and Distributed Computing*, 14:105–120, 1992.

[Kob78] H. Kobayashi. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1978.

[MB92] Daniel A. Menascé and Luiz André Barroso. A methodology for performance evaluation of parallel applications on multiprocessors. *Journal of Parallel and Distributed Computing*, 14:1–14, 1992.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[PW86] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29:1184–1201, 1986.

[SKB89] Margaret Simmons, Rebecca Koskela, and Ingrid Bucher. *Instrumentation for Future Parallel Computing Systems.* ACM Press, New York, New York, 1989.

[Wol88] Michael Wolfe. Vector optimization vs vectorization. *Journal of Parallel and Distributed Computing*, 5:551–567, 1988.

[Wol91a] Michael Wolfe. Data dependence and program restructuring. *Journal of Supercomputing*, 4(4):321–344, January 1991.

[Wol91b] Michael Wolfe. The Tiny loop restructuring research tool. In *Proceedings 1991 International Conference on Parallel Processing*, volume 2, pages 46–53, August 1991.

[WS90] Debbie Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 137–146, March 1990.