Hypertasking:

Automatic Data Parallel Domain Decomposition

on the Intel Parallel Supercomputer

Marc Baber
Dept. of Compter Science and Engineering
Oregon Graduate Institute of Science and Technology
Intel Scientific Computers
Email: marc@isc.intel.com

ABSTRACT

Many nearest neighbor data parallel problems share a similar domain decomposition strategy for hypercube architectures. That strategy is to map sub-blocks, or grains, of each array variable to the available nodes in a manner that matches the hypercube topology to the geometry of the underlying problem, maximizing locality of reference and minimizing message passing overhead. Each node then applies the same algorithm used in the original sequential code to its subset of the data, synchronizing with its neighbors as necessary.

This paper describes a domain decomposition tool that accepts a sequential C program with comment directives as input, and produces a new SPMD (Single Program Multiple Data) C program which can be run on a hypercube of any size. Directives allow the user to mark certain arrays to become distributed memory virtual arrays, to limit loops to indices that refer to the local sub-block of an array, and to cause exchanges of boundary values to occur between nodes.

The preprocessor approach maintains code portability and is similar to some techniques used on shared memory machines.

Introduction

Data parallel applications are often considered the easiest to implement on distributed memory architectures. Their decomposition is relatively straightforward and good speed-ups are usually obtained. Almost anywhere problems are approached by grid-point approximations, finite difference methods or PDEs, hypercubes can be put to good use. Other problems involving matrix mathematics can also utilize distributed memory machines well.

The mechanics of multidimensional array decomposition onto hypercube or mesh topologies of computing nodes is relatively intuitive and straightforward. Various languages and language extensions have been proposed [CK 88] and implemented [KM 87, KMV 87, Meh 89, RP 89, RS 89, Tse 89] to address the issue of automatic and semi-automatic domain decomposition, but most of them require the programmer to rewrite code in new languages, or to add new statements that can compromise the code's portability.

The domain decomposition tool described in this paper operates on comment-directives inserted into ordinary C source code. The approach is equally applicable to FORTRAN programs. The tool is called *hypertasking* because its user interface was influenced by the design of Cray Research's microtasking parallelization tool [Cra 85], and because it provides loop-level parallelism for hypercube distributed memory architectures. It is not concerned with operating systems or scheduling multiple processes or threads, as the name might imply to some readers. Hypertasking is intended to make it easy for software developers to port their existing data parallel applications to the hypercube without making their code hardware specific.

In hypertasking, arrays can be decomposed in any and all dimensions, but the number of nodes allocated in any given dimension is controlled by the underlying libraries, and is always a power of two to preserve locality of reference within the logical node mesh. All arrays are decomposed into regular rectangular sub-blocks. Guard rings [Fox+88] for each sub-block are provided. The term "guard ring" tends to imply a 2-D problem decomposed in both dimensions, but the concept is extended in this implementation to multiple dimensions. This paper proposes *guard wrapper* as a general term encompassing guard rings in 2-D decompositions, guard shells in 3-D, guard hypershells in 4-D, and so on. A *guard wrapper* could be one array element thick for 2-D 5-point stencils, or two elements thick for 2-D 9-point stencils, for example. Guard wrappers can be as thick as the user wishes, but not larger than the sub-block itself. For example, if each node's sub-block of a two-dimensional array is two elements by six elements, then the wrapper thickness cannot be more than two or it would contain the entire neighboring sub-block, plus elements owned by a third node.

Each element is stored on one or more nodes, though it is only owned by one node. Any node can read or write any element in the distributed virtual array, but communication costs make non-local reads and writes expensive. Application algorithms should exhibit good locality of reference to make hypertasking worthwhile.

To optimize performance for non-local operations, several efficiency features, discussed in the section on domain decomposition, have been implemented, including the combination of local sub-blocks with their guard wrappers in the same oversized array, and array base-shifting. Two modes of implicit message passing are currently offered, including a global exchange of all boundary values to update guard wrappers, and probes for single array elements when non-local elements are referenced.

This paper will begin by discussing the advantages and disadvantages of a preprocessor approach to parallel programming compared to other approaches. Further sections will describe the hypertasking user interface with examples and outline hypertasking's internal algorithms for decomposing arrays. Finally, performance benchmarks will be discussed, along with plans for future research and development.

Rationale and Related Work

It seems inevitable that distributed memory computers will be based on mass-produced general purpose microprocessors for the foreseeable future. With general purpose processors come general purpose compilers. It is unlikely that compiler vendors will support distributed memory extensions while distributed memory computers represent a small fraction of their market. Since it is desirable to leverage the general purpose compilers, rather than write completely new compilers, the remaining choices are to enhance the general purpose compiler or to use a preprocessor or pre-compiler. Enhancement requires compiler source code (often expensive), and learning time to understand the internals of the compiler. Other problems with the compiler extension approach include how to merge new releases of the compiler source with the distributed memory extensions efficiently and correctly, and how to port the extensions to compilers targeted for different processors and architectures.

The most practical approaches, preprocessors and pre-compilers, are both built upon existing sequential compilers. This paper proposes *paracompiler* as a term for the class of source code transformers that accept either a parallel language or an augmented sequential language as input and produce sequential source code (typically C or Fortran) as output. The term suggests both the parallel nature of these software tools and the fact that they are not true compilers in the same sense that "paramedics" are not true doctors. A *paracompiler* may be a simple preprocessor [Cra 85], performing only direct textual substitutions, or it may be a pre-compiler [KM 87, KMV 87, QHJ 88, ZBG 88, Meh 89, RP 89, RS 89, Tse 89], utilizing an intermediate structure of tokens to represent the program during the transformation process. Pre-compilers may also be interactive-- suggesting transformations to the user and soliciting approval or guidance.

One advantage of hypertasking is that it produces executables that can be run on any size hypercube from one node up to any power of two. Some other paracompilers [CK 88, RS 89] produce cube size dependent and/or fixed grain size executables. As hypercubes are used more in production environments, users may often not know at compile time what size subcube will be available to them at run time, so cube size flexibility may grow in importance.

The guard wrapper provided by hypertasking is ideal for most data parallel applications, and single element polling provides access to elements not stored locally, but intermediate granularities, such as non-local row references, or optimized combined references to the same non-local node [KMV 87] are not yet implemented.

New or extended programming languages, implemented as pre-compilers [QHJ 88, ZBG 88, Meh 89, RP 89, RS 89], produce a high-level language as their target, and can approach stand-alone compilers in sophistication of their analysis and optimization heuristics. Unfortunately, the parallel language can become a barrier for programmers if it is too unfamiliar or cumbersome. Pre-compilers, based on sequential languages, but providing a small and powerful set of extensions, may be the best overall solution. They require minimal retraining for the programmer; they can perform quite sophisticated code transformations, and users can easily hand tune the generated code.

Paracompilers which have the same language as their source input and target output [Cra 85, ACK 87] are usually easy to learn and use. This approach is best for preserving portability.

Hypertasking is currently implemented with a preprocessor. Future versions will employ pre-compilers, perform dependency analysis and handle more sophisticated implicit message passing while requiring fewer directives from the user.

The Hypertasking User Interface

Hypertasking directives appear as comments in C, so the source can be compiled with or without hypertasking, depending on whether or not the source is preprocessed by **hype**. The un-hypertasked program can only run on one node, of course, but this feature allows hypertasking to be added to a program in a non-intrusive manner, preserving portability while making speed-up measurements relatively easy.

Hypertasking Directives

The hypertasking technique relies on the user to insert comment directives into C code which is then rewritten by a preprocessor. The three main directives are `ARRAY, BRACKET` and `EXCHANGE`.

The ARRAY Directive

```
. . .
/* iSC ARRAY 1 Y Y N */
float pixels[1000][1000][2];
. . .
new = 0 ; old = 1 ;
for ( i = 0 ; i < smooth_iterations ; i++ ) {
   /* iSC EXCHANGE pixels */
   /* iSC BRACKET pixels 1*/
   for ( x = 0 ; x < 1000 ; x++ ) {
      /* iSC BRACKET pixels 2 */
      for ( y = 0 ; y < 1000 ; y++ ) {
         pixels[x][y][new] = weighted_average(pixels[x+1][y-1][old],
                                              pixels[x+1][y  ][old],
                                              pixels[x+1][y+1][old],
                                              pixels[x  ][y-1][old],
                                              pixels[x  ][y  ][old],
                                              pixels[x  ][y+1][old],
                                              pixels[x-1][y-1][old],
                                              pixels[x-1][y  ][old],
                                              pixels[x-1][y+1][old]);
      }
   }
new = (new+1)%2; old =(old+1)%2;
}
```

Example 1: An image processing application

The ARRAY directive is inserted in the user's code before a standard array declaration. Arguments for the ARRAY directive specify guard wrapper thickness, and which dimensions of the array are to be distributed.

The general ARRAY directive syntax is:

```
/* iSC ARRAY <thick> <Y|N> ... */
```

"Y" and "N" are used as distribution flags and indicate, for each dimension, whether or not the array may be distributed in that dimension. The ARRAY directive in Example 1 sets the guard wrapper thickness to one, and distributes a 3-D array in the first two dimensions.

Guard wrappers store values of array elements that are adjacent to local elements in the virtual array, but are owned by a neighbor node. One or more elements outside of the local sub-block's boundaries are stored in every direction, including diagonals. The thickness of the guard wrapper is controlled by the <*thick*> argument to the ARRAY directive. Guard wrapper updating is controlled by the EXCHANGE directive, discussed in the next section.

The EXCHANGE Directive

Guard wrappers of the specified array are updated between neighboring nodes each time the code from an EXCHANGE directive is executed.

The general EXCHANGE directive syntax is:

```
/* iSC EXCHANGE <array-name> */
```

Generally, the EXCHANGE directive is used at the beginning of the outermost loop in an array processing section , as in the pixel processing code shown in Example 1. If multiple arrays are modified in the loop, multiple EXCHANGE directives must be used.

The user is responsible for inserting the EXCHANGE directive at appropriate points in his or her program to make sure the algorithm uses current values.

The BRACKET Directive

The `BRACKET` directive syntax is:

```
/* iSC BRACKET <array> <dim> */
```

The `BRACKET` directive has two arguments, array name and dimension number. It precedes a C for-loop whose local iterations are to be limited to values that are valid local indices for the specified array in the specified dimension.

The user is responsible for determining whether the iterations of a loop are independent. Fortunately, for many problems such as CFD and reservoir modelling, iterations are inherently parallel, and the problem reduces to detecting implementations that impose restrictions in the algorithm that do not exist in the actual physical problem.

Example 1 demonstrates a technique used in Jacobi algorithms to avoid copying the new array into the old array at the end of each iteration. Two variable indices to the third dimension are toggled between 0 and 1 and, in effect, swap old and new values. Only the first two dimensions are distributed and `BRACKET`ed. By not distributing the third dimension of the pixels array, it is guaranteed that each old element will be stored on the same node as its corresponding new element.

The LOCAL Directive

Array references and assignments can be either global or local. Global references are the default generated by hypertasking, and represent the general case for distributed memory virtual arrays. A global array reference can access any element anywhere in the hypercube. Global references and assignments are implemented with conditional expressions that are resolved without subroutine calls for elements in the local sub-block, but which require a subroutine call, and usually message passing, to return or update the values of elements owned by other nodes.

If the user knows that a given array reference or assignment can be satisfied locally, it can be declared local by inserting the string, `/*LOCAL*/` between the array name and its indices, as shown in the example in the next section. A local array reference saves the cost of evaluating the conditional expression to determine if the element is local, and should only be used if the user knows that the algorithm will only reference values that are local or at least are contained in the local guard wrapper and are current.

Example: Gauss-Siedel/Jacobi Problem

Example 2 is an implementation of a Gauss-Siedel/Jacobi hybrid algorithm for thermal problems. This particular code solves a two-dimensional 32-by-32 floating point array representing a square thermal plane with four constant edge temperatures. The convergence test and output of the final results did not convert cleanly, so conditional code was added.

To run the above program with hypertasking, the user runs the source through the **hype** preprocessor, which changes array references to macros, modifies **for** loops, and adds calls to hypertasking library routines. The resulting C program is then compiled and linked with the hypertasking libraries.

Once the program is compiled and linked, the user need only allocate a cube of any size, and load the executable.

Domain Decomposition

This section describes the algorithm used by hypertasking to determine how a given array is decomposed onto the available nodes. An initialization routine **(ht_init)** is called at the beginning of program execution to determine what section of the array is local. Each node uses the current cube dimension, its own unique node identifier, the global dimensions of the virtual array and the user's directives (specifying which dimensions to decompose) to calculate the size of one sub-block and which sub-block it owns.

Example 2: Gauss-Seidel Thermal Problem

```
#define SIZE 32
#include <stdio.h>
/* iSC ARRAY 1 Y Y */
float ii[SIZE][SIZE];
/* iSC ROUTINE */
main() {
int i,j,k,l,m,n,o,p;
float q,r,s,t,u,v;
char density[11];

/* iSC INIT */
strcpy(density,” .:!-=*%#@E”);
/* initialization of constant temperatures */
/* iSC BRACKET ii 1 */
for ( j = 0 ; j <= SIZE-1 ; j++ ) {
   /* iSC BRACKET ii 2 */
   for ( k = 0 ; k <= SIZE-1 ; k++ ) {
   ii/*LOCAL*/[j][k] = (((k==SIZE-1)&&(j>0)&&(j<SIZE-1))*100.0 +
     (j==SIZE-1)*1000.0 + (j==0)*500.0);
   };
};

r = (float)(SIZE*SIZE+1);
i = 0;
o = mclock();

/* repeat until convergence criterion is met */

while ( r > (SIZE) ) {

i++;
r = 0;
/* iSC EXCHANGE ii */
/* iSC BRACKET ii 1 */
for ( k = 1 ; k <= (SIZE-2) ; k++ ) {
   * iSC BRACKET ii 2 */
   for ( l = 1 ; l <= (SIZE-2) ; l++ ) {
      q = ii/*LOCAL*/[k][l];
      ii[k][l]/*LOCAL*/= (ii/*LOCAL*/[k-1][l] + ii/*LOCAL*/[k+1][l] +
                         ii/*LOCAL*/[k][l-1] + ii/*LOCAL*/[k][l+1])/4.0;
      q = q - ii/*LOCAL*/[k][l];
      if ( q < 0 ) { q = -q; };
      r = r + q;
   }
}
#ifdef hypertasking
gssum(&r,1,&s);
#endif

} ;
p = mclock() - o;
#ifdef hypertasking
if ( mynode() == 0 ) {
```

Example 2 (Continued):

```
#endif
   for (k=0;k<SIZE;k++) {
      for (l=0;l<SIZE;l++) {
         n = (int)(ii[k][l])/100;
         putchar(density[n]);
      };
      putchar('\n');
   };
printf("SIZE=%d NUMNODES=%d\n",SIZE,numnodes());
printf("TIME=%d milliseconds\n",p);
printf("ITERATIONS=%d\n",i);
o = (SIZE-2)*(SIZE-2)*4*i;
t = (float)o/(float)p;
u = t/(float)numnodes();
printf("OPERATIONS=%d\n",o);
printf("Kflops/sec=%f\n",t);
printf("Kflops/sec/processor=%f\n",u);
#ifdef hypertasking
};
gsync();
#endif
}
```

Mapping the Hypercube to Array Dimensions

First, **ht_init** divides the cube dimensions among the array dimensions which are eligible for decomposition, according to the user's directives. Undecomposed dimensions of the array are assigned zero.

The goal of cube dimension mapping is to minimize the ratio of exterior elements to interior elements, without sacrificing locality of reference. The mapping algorithm assigns each cube dimension to the array dimension that will result in the smallest increase in cross-sectional area, taking the previous cube dimension assignments into account.

Another way to look at this problem is that each assignment of a cube dimension to an array dimension halves the size of every sub-block. In this view, the goal is to reduce the maximum boundary value message size. Since all messages can be sent in all dimensions/directions nearly simultaneously in hypertasking, the largest boundary value message is the critical limitation to speed-up. An algorithm, based on this view, finds the largest "side" (edge in 2-D, face in 3-D, etc.) and assigns the next cube dimension in an orthogonal array dimension to split the largest "side" of the sub-block.

Both goals are achieved by the same algorithm, demonstrated below, which is based on the former abstraction, and is simpler.

As an example, consider a 3-D array A(2,4,16) decomposed on a 6-D hypercube (64 nodes) (Fig. 1). The cost of dividing A in each of its three dimensions is initially:

$$c = 2^n * s/k$$

where:

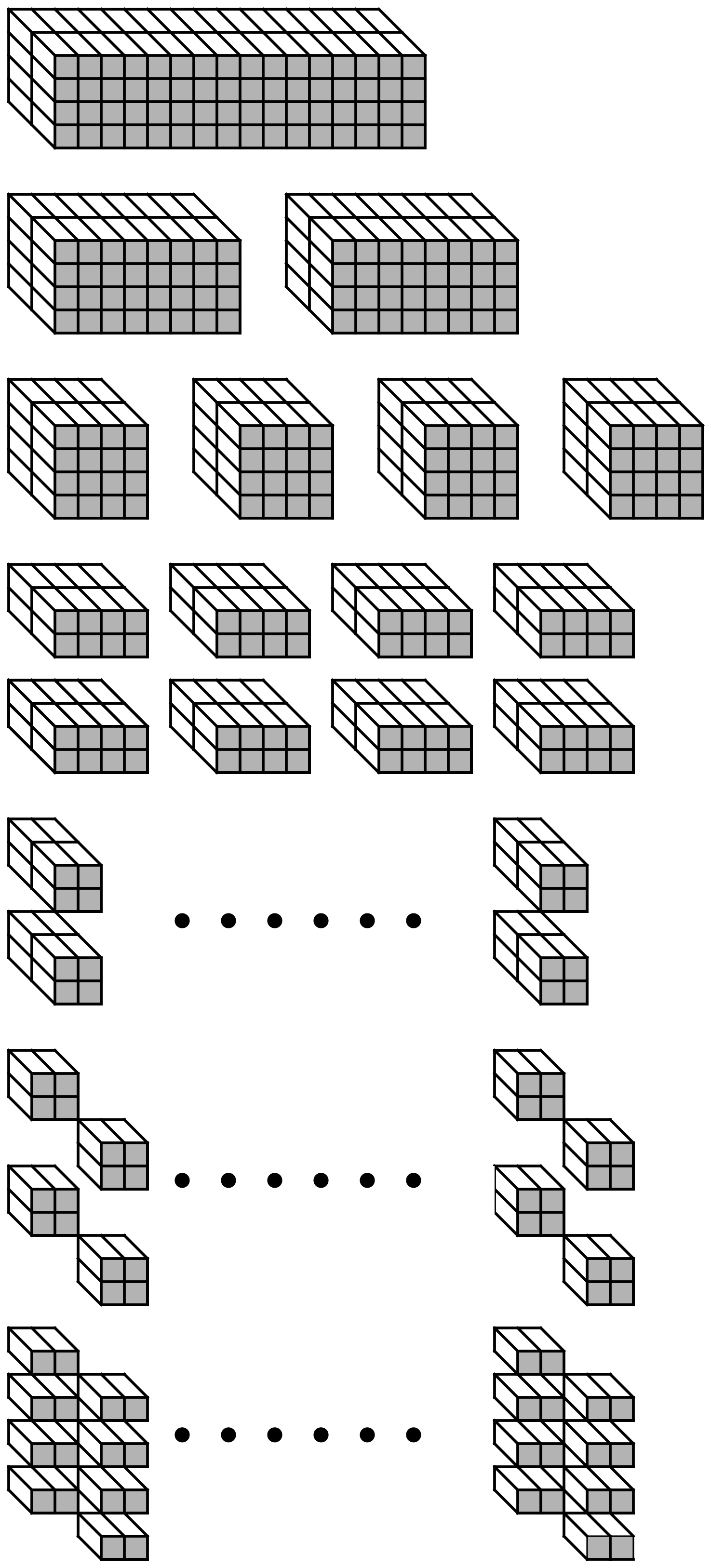
c = cost of further decomposition of the current dimension

n = how many cube dimensions have already been assigned to the current dimension

s = size of the array (total number of elements)

k = declared array size in the current dimension

Figure 1: Stepwise 3-D domain decomposition



A(2,4,16)

Step 1:
Divide third dimension.
Cost = 1 x 8 = 8

Step 2:
Divide third dimension.
Cost = 2 x 8 = 16

Step 3:
Divide second dimension.
Cost = 1 x 32 = 32

Step 4:
Divide third dimension.
Cost = 4 x 8 = 32

Step 5:
Divide first dimension.
Cost = 1 x 64 = 64

Step 6:
Divide second dimension.
Cost = 2 x 32 = 64

$c_1 = 2^0 * 128/2 = 64$
$c_2 = 2^0 * 128/4 = 32$
$c_3 = 2^0 * 128/16 = 8$

The first dimension of the cube is mapped to dimension 3 of the array, incurring the minimum cost of 8 elements of the array being exposed to inter-nodal communication overhead by “slicing” the array once. Actually, 16 elements are exposed, since two surfaces are created, but since the exchange can happen in parallel and the extra factor of two would only complicate matters unnecessarily, we will count only the cross-sectional area, or eight units, in this case. One dimension of the available cube has now been assigned and five remain. The cost of further decomposition of dimension three doubles and the relative costs of assigning the next cube dimension become:

$c_1 = 2^0 * 128/2 = 64$
$c_2 = 2^0 * 128/4 = 32$
$c_3 = 2^1 * 128/16 = 16$

The second dimension of the cube is also assigned to dimension 3 of the array, for a cost of 16 units. On the next iteration the updated costs are:

$c_1 = 2^0 * 128/2 = 64$
$c_2 = 2^0 * 128/4 = 32$
$c_3 = 2^2 * 128/16 = 32$

The algorithm arbitrarily selects the lower dimension number when costs are equal. The third cube dimension is assigned to dimension two of the array, and its cost is updated to 64. The fourth assignment goes to dimension three again, and its cost doubles to 64 also. The fifth and sixth assignments would go to dimensions one and two, respectively, leaving the final cost list as follows:

$c_1 = 2^1 * 128/2 = 128$
$c_2 = 2^2 * 128/4 = 128$
$c_3 = 2^3 * 128/16 = 64$

Notice that the number of cube dimensions mapped to array dimensions one and three differ by two. In extreme cases, say `B(2,2,128)`, all the cube dimensions may map to a single array dimension, even if all the array dimensions were eligible for decomposition, according to the user’s directives.

Through out the rest of this paper, the notation `A.dims(n)` will be used to refer to the number of cube dimensions that were mapped to the `n`th dimension of an array `A`.

Interpreting Node ID Bits

For each decomposed array, hypertasking maps or “unfolds” the hypercube into an n-dimensional logical mesh, where n is either the dimension of the hypercube or the number of array dimensions decomposed, whichever is the lesser.

The **ht_init** routine partitions the Node ID into bit strings with lengths corresponding to the values of `A.dims(k)` where `k` is a series `1..n` for an n-dimensional array. Undecomposed dimensions get zero-length bit strings, and are, in effect, ignored for the purposes of defining the logical node mesh.

For mesh-interconnected architectures, hypertasking would use each bit string as a simple binary number, indicating which sub-block, in the given dimension, the current node owns. For example, consider a 2-D array `B(0..79,0..79)` decomposed in both dimensions on a 64-node system, connected as an eight-by-eight 2-D mesh (Fig 2a). Node 29 would have `011-101` as its binary node identifier (divided into two three-bit strings). Thus, node 29 would be in the third sub-block in the first dimension, the fifth sub-block in the second dimension, and would contain `B(20..29,40..49)`.

The simple use of bit strings as binary counters is unsatisfactory for hypercube architecures. In the above example, element `B(0,19)` would map to node id `000-001`,

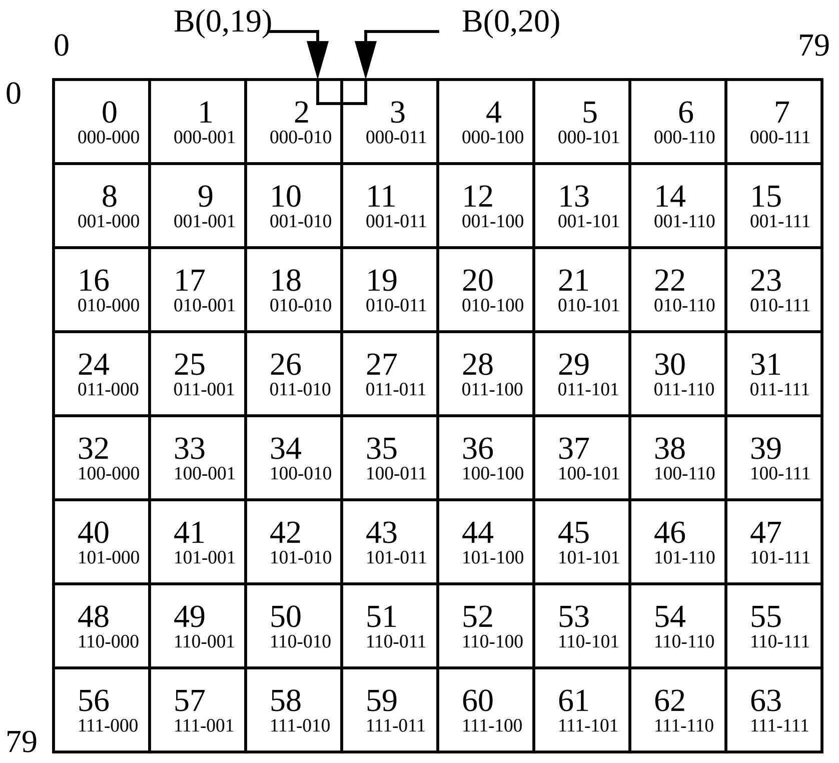


Figure 2a: Simple Array Decomposition for Mesh Architectures

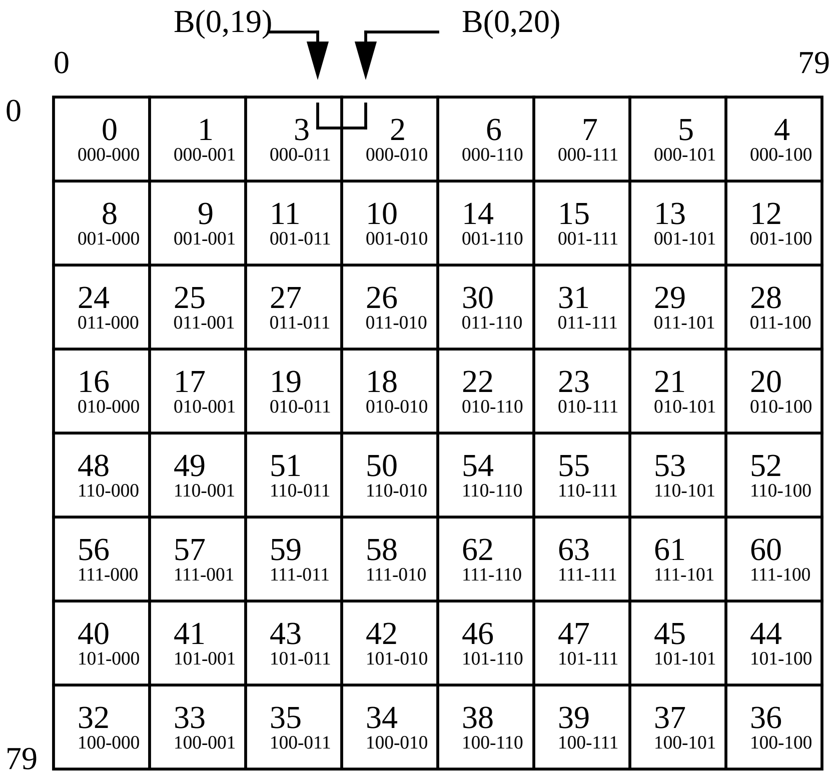


Figure 2b: Array Decomposition with BRGCs for Hypercube Architectures

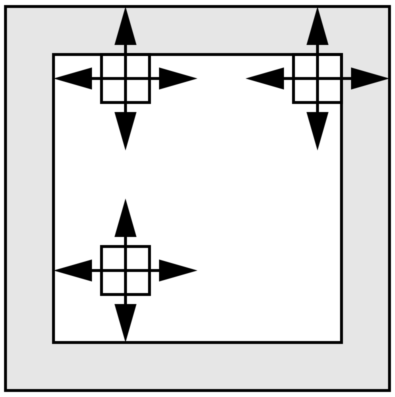


Figure 3:
Since sub-blocks and guard wrappers are stored in the same array, every element in the local sub-block calculates the addresses of its neighbors identically.

whereas `B(0,20)` would map to node id `000-010`. Since the two node ids differ by two bits, the two nodes are not directly connected.

Therefore, for hypercubes, each bit string is seen as the binary reflected gray code (BRGC) of the sub-block's position in a given dimension (Fig. 2b). The algorithm applies a BRGC inverse function to the bit string to determine its node's actual position in the logical node mesh. For mesh-interconnected architectures, an element's home node would be determined by concatenating bit strings which are calculated by dividing the element's index in the corresponding dimension by the size of the sub-block in that dimension and truncating. To use BRGC in the logical node mesh, an additional step of converting each bit string to its gray code is performed before concatenation.

This method guarantees that locality of reference within an array is preserved within the logical node mesh, and that global boundary value exchanges can occur simultaneously for all nodes in all dimensions without any contention for channels, except in the case of diagonal boundary-values.

Guard Wrappers

After the **ht_init** routine determines the size of one sub-block, it allocates a continuous segment of memory large enough to contain the sub-block plus a *guard wrapper* of a user-specified thickness, usually one or two elements. The sub-block is managed in memory like a multidimensional C array that encompasses both the sub-block and the guard wrapper. As a result, both internal and external elements access their neighbors, both local (in the sub-block) and non-local (in the guard wrapper) using the same addressing (Fig 3). Thus, if no guard wrapper updating is

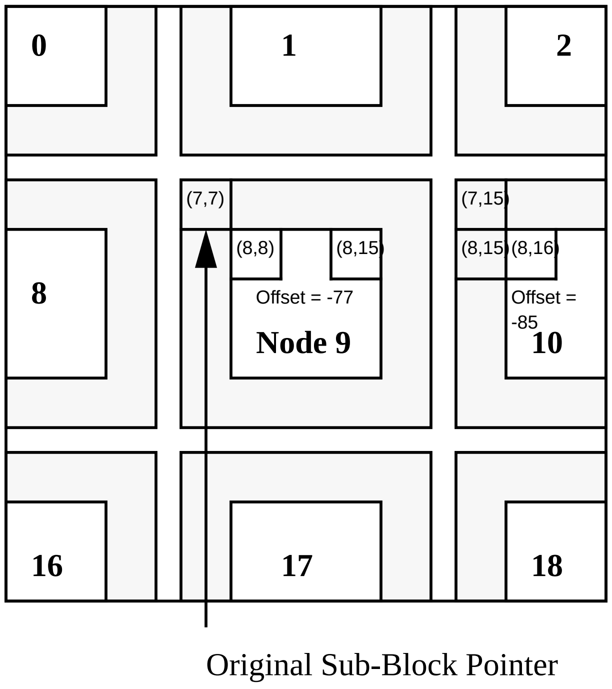


Figure 4:
The pointer to the sub-block on node 9 is off-set by (7 x 10 + 7) = 77, to simplify (speed up) address calculations. Note that with offsets, the address for element (8,15) is 95 on both nodes 9 and 10.

necessary during an iteration, processing of the local sub-block can be vectorized as a single loop (or nested loop set); the loop(s) would not have to be split to accommodate different array addressing modes.

Array Base Shifting

In order to avoid the cost of subtracting the lowest local indices from the indices of a desired array element each time an element is referenced, **ht_init** subtracts from the sub-block pointer a value equal to the calculated relative address of the local element with minimum indices in all dimensions.

For example, consider a 2-D 64-by-64 array decomposed on a 64-node (6-D) hypercube mapped into an eight-by-eight logical node mesh (Fig. 4). In this example, we will ignore gray codes for simplicity, and assume that nodes in the same dimension are numbered sequentially. Node nine (`001-001` node identifier) would contain the sub-block `A(8..15,8..15)`, or `A(7..16,7..16)` counting the guard wrapper. With the guard wrapper one element thick, the local sub-block is a ten-by-ten array. Originally, the sub-block pointer points to `A(7,7)`. To get `A(9,9)`, you would have to calculate the offset as follows:

$$(9-7)*10 + (9-7)*1 = 22$$

The offset for `A(7,7)` is initially zero. If we did not subtract the local minimums from the indices, the address of `A(7,7)` would be

$$7*10+7 = 77$$

ht_init, in this case, would subtract 77 from the address stored in the sub-block pointer, so that all offsets could be calculated without subtracting. As one might expect, no array bounds checking is provided.

A major advantage of *base shifting* is that a given element in the global virtual array has the same offset on its home node as it does on every node where it appears in a guard wrapper.

Performance Results

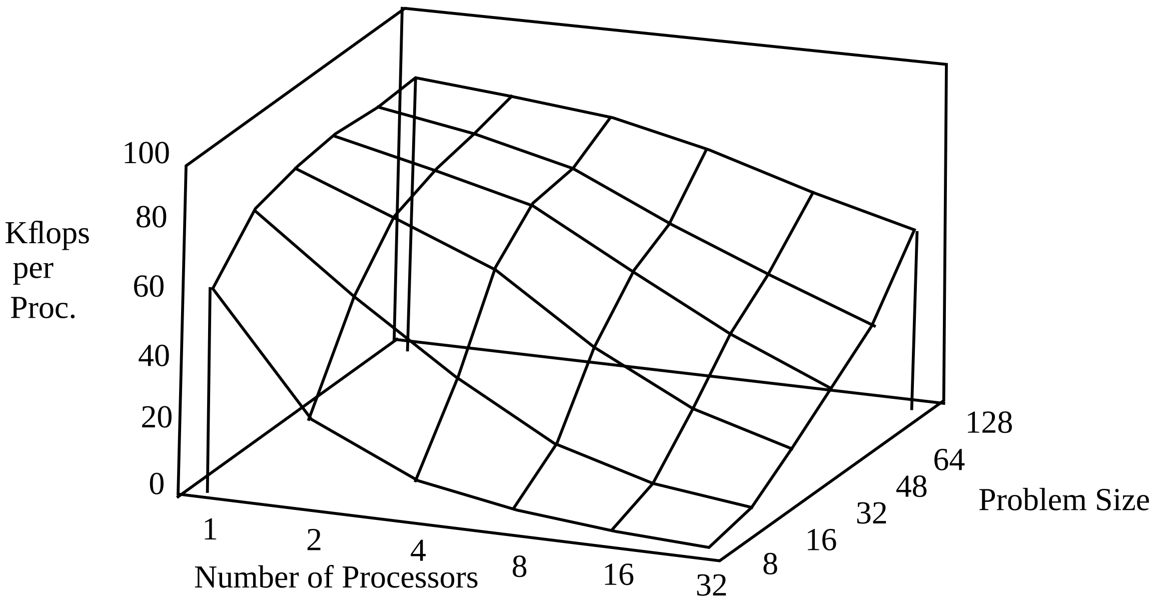
Figure 5a is a 3-D graph showing the relations between hypercube size (or number of processors), problem size and individual processor efficiency for a hypertasked program running on an iPSC/2 witout SX acceleration. Figure 5b shows the same information for the iPSC/860. Efficiency is not sacrificed significantly (linear speed-up is approached) for large problems or small cube sizes, but the combination of large numbers of processors and small problem sizes yields very poor efficiency.

For comparison, a flat surface level with the single node unhypertasked performance would indicate perfect, linear speed-ups for any number of processors. A binary logarithmic decay (running down from left to right) would define a break even surface, where a problem of a given size would run in the same wall clock time on any size hypercube.

The benchmark problem was a Gauss-Seidel/Jacobi hybrid algorithm finding a thermal equilibrium for a 2-D, homogeneous, square material with constant temperatures at the edges, using a five point stencil and simply averaging neighbor values to obtain each element's value. The algorithm is a hybrid because boundary values are not updated during each iteration, so at the edges of each sub-block the algorithm uses some old values from the previous iteration (Jacobi) whereas a pure Gauss-Seidel algorithm would always use current iteration values for the north and west neighbors. In the interior of each sub-block, the algorithm is entirely Gauss-Seidel.

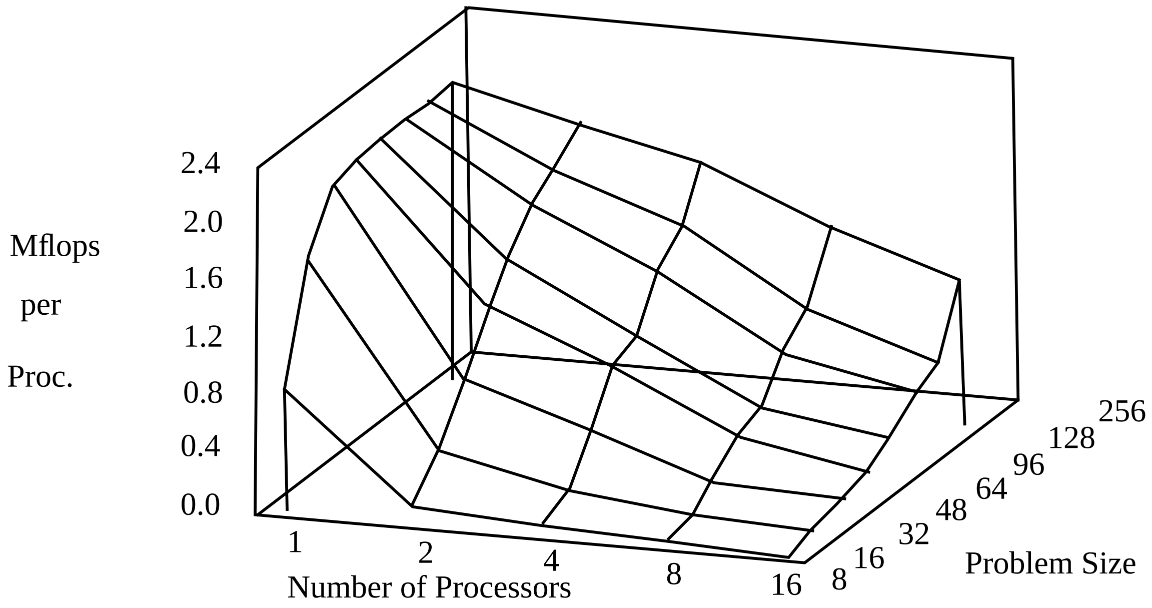
The performance graphs in Figure 5 represent typical speed-ups for parallel algorithms, and do not reflect the slight degradation of the Gauss-Seidel/Jacobi hybrid algorithm as more processors are used. If a pure Jacobi algorithm were used, no such degradation would occur, since the Jacobi algorithm is completely parallelizable. The efficiency graph would then exactly match the graphs in Figure 5.

Figure 5a: Performance Results for the iPSC/2



86.4 Kflops = 100% efficiency

Figure 5b: Performance Results for the iPSC/860



2.65 Mflops = 100% efficiency

The following table shows the number of iterations required to converge for various problem sizes and algorithms (G-S indicates Gauss-Seidel and G-S/J-n. is the G-S/Jacobi hybrid running on an n-node cube). The percentage of degradation from pure Gauss-Seidel performance is shown in parentheses. Interestingly, the degradation is minimal for larger problems and small cube sizes-- its main impact is on cube and problem size combinations that would not yield good speed-ups anyway.

Algorithm	Problem Size			
	16^2	32^2	64^2	128^2
G-S	118(0)	431(0)	1495(0)	4929(0)
G-S/J-2	123(4)	440(2)	1509(1)	4948(0)
G-S/J-4	129(9)	450(4)	1525(2)	4972(1)
G-S/J-8	135(14)	461(7)	1544(3)	5001(1)
G-S/J-16	140(19)	471(9)	1561(4)	5028(2)
G-S/J-32	151(28)	491(14)	1594(7)	5080(3)
Jacobi	200(69)	720(67)	2420(62)	7569(54)

The temperatures were stored as C **float** types (single precision). The hypertasking libraries and the hypertasked version of the main program were compiled with the Greenhill C compiler using the -OLM optimization switches, without any attempts to use vectorization.

These benchmarks were measured before BRGC node mapping was completed. With BRGC logical node mesh-mapping, there should be no degradation in performance if the problem size and the cube size both grow proportionally.

Future Plans

Since hypertasking can be adapted to mesh-topology architectures as well as hypercubes, it will be a valuable tool for comparing the efficiency and speed of various connection topologies and technologies which may be investigated in the future.

Combining vectorization (pipelined execution) with automatic decomposition on the Intel/860 will probably exhibit aggregate calculation rates in the Gigaflop range or more for sufficiently large problems. To get the same efficiencies found in the previous section, the required problem sizes will likely be larger. I would like to quantify, experimentally, how much larger problems must grow to preserve efficiency.

The ability to deallocate, reallocate, grow and shrink virtual arrays can be added to hypertasking fairly easily. Such a feature would allow the user to easily employ multigrid methods, or to change the decomposition strategy for performance gains in different sections of an algorithm. Dynamic arrays would also require sequential versions of the new routines to be implemented in order to preserve the goal that hypertasked programs should run with essentially the same capabilities as the sequential versions.

A Fortran version of hypertasking would greatly increase the technique's usefulness, since many, if not most, of the applications that would most benefit from hypertasking are written in Fortran. The difficulty of implementing a Fortran version stems from Fortran's inability to support POINTER types which are useful for dynamic memory management. A Fortran version can be implemented easily if POINTER type extensions are provided in the underlying Fortran compiler similar to Cray Research's CFT extension. If not, a Fortran program can be encased in C code above and below it in the calling hierarchy. The new main program, written in C, would allocate space for sub-blocks in the normal way, and then pass the sub-block's dimensions to the Fortran main program, converted to subroutine form. Within the Fortran program, only standard array syntax would be used, but the order of subscripts would be reversed to account for the differences in array allocation in C and Fortran.

A more robust preprocessor, based on **lex** and **yacc**, will allow some directives to be eliminated and provide the programmer greater latitude. Dependency analysis could generate or replace LOCAL directives and determine proper *guard wrapper* thickness.

Ideally, the preprocessor would divide the source program into sequential and parallel sections that would be compiled and run on the cube host and cube nodes respectively, in a manner similar to how Cray microtasking delineates sequential and parallel routines. When the main program, running on the host, reaches a call to a hypertasked routine, it will allocate a cube of an appropriate or available size, load the hypertasked routine onto the nodes, pass parameters and global variables as host-to-node messages, and wait for cube execution to complete. This approach improves cube utilization in a multi-user environment, because cubes are only allocated for parallel work. Sequential sections of other programs can run on the host while the first program waits for its node part to complete, and the hypercube resource itself can go from one parallel code to another, executing a minimum of scalar code.

A more intelligent preprocessor, or pre-compiler, would probably evolve from the current level to an interactive precompiler first, and then, gradually to a non-interactive precompiler, as more heuristics are encompassed within the program.

Conclusions

Hypertasking represents an approach to data parallel programming that requires minimal source code changes in the user's application, and can be expected to deliver optimum performance on either hypercube or mesh topology architectures for a large class of grid point applications. The technique currently requires the user to have a clear understanding of the hypercube architecture and when array elements should be processed in parallel. Future versions may go beyond the current preprocessor front-end and provide dependency analysis and other features to further free the parallel programmer from the details of porting applications to distributed memory architectures.

References

[ACK 87] Allen, R., Callahan, D., Kennedy, K., "Automatic Decomposition of Scientific Programs for Parallel Execution," Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Published by the Association for Computing Machinery, (1987).

[CK 88] Callahan, D., Kennedy, K., "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing, 2,* (1988), 151.

[Cra 85] Cray Research, Inc., 1985, "Multitasking User Guide," CRI internal technical note, SN-0222.

[Fox+88] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D., *Solving Problems on Concurrent Processors,* Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey (1988).

[KM 87] Koelbel, C., Mehrotra, P., "Semi-Automatic Domain Decomposition in BLAZE," in *Proceedings of the 1987 International Conference on Parallel Processing,* Published by the Pennsylvania State University Press, University Park, Penn. (1987).

[KMV 87] Koelbel, C., Mehrotra, P., Van Rosendale, J., "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming, 5,* (1987), 365.

[Meh 89] Mehrotra, P., "Programming Parallel Architectures: The BLAZE Family of Languages," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing,* Published by the Society for Industrial and Applied Mathematics, Philadelphia, Penn. (1989).

[QHJ 88] Quinn, M., Hatcher, P., Jourdenais, K., "Compiling C* Programs for a Hypercube Multicomputer," *Proceedings of the ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems,* New Haven, Connecticut, July 19-21, 1988. Published by the Association for Computing Machinery.

[RP 89] Rogers, A., Pingali, K., "Process Decomposition Through Locality of Reference" in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation,* Published by Association for Computing Machinery Special Interest Group on Programming Languages, Portland, Ore. (1989).

[RS 89] Rosing, M., Schnabel, R., "An Overview of Dino -- A New Language for Numerical Computation on Distributed Memory Multiprocessors," in *.Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing,* Published by the Society for Industrial and Applied Mathematics, Philadelphia, Penn. (1989).

[RSW 88] Rosing, M., Schnabel, R., Weaver, R., “Dino: Summary and Examples,” Unpublished University of Colorado at Boulder technical report CU-CS-386-88, (1988).

[Tse 89] Tseng, P., “A Parallelizing Compiler For Distributed Memory Parallel Computers,” Carnegie Mellon University Ph.D. dissertation (1989).

[ZBG 88] Zima, H., Bast H., Gerndt, M., “SUPERB: A tool for semi-automatic MIMD/SIMD parallelization,” *Parallel Computing, 6*, (1988), 1.