Accommodating Polymorphic Data Decompositions in Explicitly Parallel Programs *

Calvin Lin Lawrence Snyder

Dept. of Computer Science and Engr., FR-35
University of Washington
Seattle, WA 98195

Abstract

Explicitly parallel programs have the potential for greater performance than their implicitly parallel counterparts. However, this benefit can be accompanied by additional programming difficulties. This paper addresses one particular problem that has implications for both scalability and portability: the need for programs to accommodate diverse data decompositions. We explain why programs with explicit communication have difficulties in handling changes in data decomposition, and we present our solution to this problem which involves the notions of derivative functions *and* configuration parameters. *We illustrate our technique by using three different data decompositions to solve the Modified Gram-Schmidt method on four parallel machines.*

1 Introduction

Implicitly parallel languages are favored for their programming convenience. By contrast, explicitly parallel languages, particularly those with explicit communication, tend to provide both superior performance and additional programming pitfalls. The performance benefit stems from the *ability* to control details of communication and granularity [8, 11]. The programming difficulties come from the *need* to specify such "low-level details." In message passing languages, the programmer controls both data decomposition and communication. Since these two aspects of a program are intimately related, it would seem that message passing programs must be significantly rewritten when a different data decomposition is desired. Hence, programs with explicit communication appear to be less flexible, and thus less scalable and portable. This paper refutes this claim by showing how the Phase Abstractions programming model provides flexibility with respect to data decompositions.

We first show that different execution environments require different data decompositions in order to achieve scalability and portability. Next, we explain why it is hard for explicitly parallel programs to handle changes in data decompositions. We then describe our solution in the context of the Phase Abstractions programming model, which is to specify a general program that is customized at loadtime. In our solution flexibility comes from two constructs: *derivative functions* and *configuration parameters*. We show how derivative functions provide the illusion that boundary conditions do not exist, thus isolating the source code from the details of the data decomposition. Configuration parameters help eliminate the overhead of using a general solution. We illustrate these concepts by using the Modified Gram-Schmidt method (MGS) as an example.

2 The Need for Polymorphism

To choose the best data decomposition, many factors should be considered, including characteristics of the problem size, the hardware and the application. Supporting changes to these factors requires that programs support different, or *polymorphic*, data decompositions.

Issues of Scale. Consider, for example, linear algebra algorithms that manipulate 2D matrices. These matrices can be partitioned across one dimension (strips) or two (blocks). Schreiber shows that for sparse Cholesky factorization of $N \times N$ matrices, column decompositions are not as scalable as block decompositions because as P grows the algorithm is lim-

*This research was supported in part by ONR Grant N00014-89-J-1368, ARPA Grant N00014-92-J-1824 and NSF Grant CDA-9211095

0-8186-5602-6/94

ited to $O(N)$ parallelism rather than $O(N^2)$ parallelism [14]. This same argument applies to other matrix computations, including the MGS method.

Hardware Factors. Machines with high message startup cost favor a small number of messages; those with a large per-byte communication cost favor a small communication volume. In general, the former situation suggests a strips decomposition while the latter suggests blocks: Strips have fewer neighbors and thus fewer messages, while blocks have lower perimeter-to-area ratios, and hence less overall data volume.

Software Factors. Applications are typically composed of multiple phases. For example, MGS is just one phase of the Car-Parrinello (CP) molecular dynamics simulation. As we show later, MGS typically performs better with a block decomposition, but other computationally more expensive phases of the CP algorithm prefer strips, so it is more efficient for MGS to adopt a 1D decomposition rather than force data conversion between phases.

The implication of the above points is that code must be flexible if it is to be portable and reusable. The code must accommodate changes in scale, and the code must be able to adapt to different hardware and software contexts.

3 Problems with Polymorphism

There are many ways to specify data decompositions. Parallelizing compilers perform this task automatically, though at times with suboptimal performance. With more recent approaches such as HPF [6] and Vienna Fortran [4], the programmer describes the data decomposition but not other aspects of parallelism, such as communication. However, explicit communication plays a fundamental role in many parallel algorithms. For example, communication is the distinguishing feature of Batcher's sort and various matrix multiplication algorithms. Since compilers cannot be expected to infer high level communication abstractions from low level data dependencies, these algorithms must be specified with explicit communication. In recognition of this shortcoming, MetaMP [13] provides a set of canned communication operations that includes global combining and matrix rotation. Our approach goes one step further and allows fully general communication as specified by the programmer.

Together, a program's data decomposition and data dependencies define how processes communicate. When the data decomposition changes, the communication pattern also changes. For example, for processes whose dependencies are with their four nearest neighbors, decomposition by rows implies communication with North and South neighbors, decomposition by columns implies East and West neighbors, and decomposition across both rows and columns implies communication with all four neighbors. Because of boundary conditions, different processes may have different neighbors, e.g., processes on the top edge of the process array have no northern neighbors, while those in the interior have four neighbors. (See Figure 1).

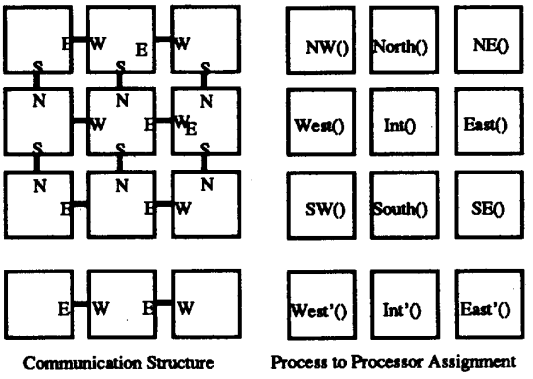

Figure 1: Proliferation of Special Cases When Decomposition Changes.

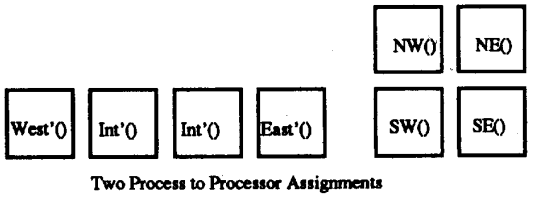

Figure 2: Data Decomposition Affects Assignment of Processes to Processors.

Without language support there are two ways to handle this variable number of neighbors. One solution would have the source code explicitly test for neighbors before transmitting messages. This leads to cluttered code that can severely hinder readability [12]. A second solution is to write different programs for each process. For example, a process on the top edge would assume no northern neighbors, while those in the interior would assume all four neighbors exist. This has two problems. First, there is a large number of programs to write. Figure 1 shows that 12 different codes are needed to scale from 3 to 9 processes. Second, when the data decomposition changes the mapping of programs to processes must also change. Figure 2 shows that different decompositions can require

different code-to-processor assignments even when the number of processes do not change. In the *ideal solution* programmers would write an SPMD program that deals only with the "common case" and works for any data decomposition. Such a solution would help decouple data decomposition from communication while yielding the performance benefits of explicit communication. The next section explains how our model supports such a programming style.

4 Our Solution

Our solution is based on the The Phase Abstractions programming model [1, 5, 15] and the Orca C language [9, 10]. Figure 8 sketches the structure of an Orca C program with an emphasis on the *configuration parameters*. These parameters are computed once at load time to adapt to different architectural or input conditions. Once computed, these values are constants in the remainder of the program.

The Phase Abstractions model consists of two major components: the XYZ programming levels and the ensembles. The XYZ levels are a means of structuring a parallel computation:

> The X, or *process*, level is the composition of instructions into processes
>
> the Y, or *phase*, level is the composition of processes into phases
>
> the Z, or *problem*, level is the composition of phases into a problem solving program.

Further layers of composition could be added, but they seem to be conceptually unnecessary.

The X level provides the primitive units from which concurrent activity is defined. Processes encapsulate units of computation that can execute concurrently. This allows grain size to be parametrically controlled, which is critical for portability and scalability.

The Y level corresponds to our informal notion of a parallel algorithm. A phase is most easily thought of as a graph, with vertices representing processes and edges indicating interprocess communication. The processes execute concurrently to collectively accomplish a single computation such as an FFT or matrix multiplication. A phase describes the scalable concurrency of a parallel algorithm: Additional concurrency manifests itself as additional nodes in the graph.

Only computer scientists are interested in isolated algorithms such as the FFT. Sophisticated applications such as weather prediction and seismic analysis require the combined effect of many parallel algorithms. Accordingly, the Z level specifies the control logic that invokes parallel algorithms to solve the user's problem. For example, the high level logic of a program might be to invoke a phase that reads data, invoke an FFT phase on the data, etc.

The programming constructs for the X and Z levels are conceptually straightforward, but the explicit parallelism of the Y level requires new facilities known as *ensembles*. An ensemble is a set with a partitioning, with each partition known as a *section*. Three types of ensembles are used to define a phase:

> A *data ensemble* is a data structure with a partitioning;
>
> A *code ensemble* is set of process instances with a partitioning;
>
> A *port ensemble* is a graph with a partitioning.

Ensembles that conform, i.e., have the same partitioning, can form a phase. The corresponding sections from the data, code and port ensembles are associated so that the process assigned to a section (by the code ensemble) operates on the data assigned to that section (by the data ensembles) and communicates with the neighboring sections as defined by the graph (from the port ensemble). Each section is assigned to a processor for execution.

Figure 3 shows a 6×6 matrix, P, and its ensemble for a 3×3 array of sections. Graphical depictions of code and port ensembles are shown in Figure 1, where each rectangle represents a section. The code ensemble assigns a process name, such as North(), to each section. In the port ensemble figure, heavy lines indicate logical communication between sections and names at the ends of heavy lines represent ports. Port names are used at the X level where messages are transmitted through named ports.

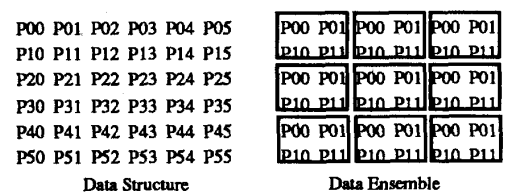

Figure 3: A 6×6 Matrix and its Ensemble

4.1 Derivative Functions

Section 3 discussed problems with polymorphic data decompositions. The Phase Abstractions model

solves this problem by separating common case code from boundary condition code through the use of *derivative functions*: This allows the port ensembles to scale, provides a clean SPMD style, and handles boundary conditions at the Y level where they naturally belong. Essentially, derivative functions insulate the X level code from changes in the data decomposition by allowing the X level code to behave as if all ports exist. For each boundary condition the programmer writes a derivative function and binds it to the appropriate ports. These functions execute locally in each process to simulate the behavior of ports. For example, a simple derivative function might return the value of a variable whenever a process receives a message from that port (representing a boundary condition that is a reflection). As another example, sends to an otherwise unconnected port can invoke a no-op derivative function. More complicated derivative functions are possible [1, 9], but the key point is that *from the perspective of the process code, all ports exist and there are no special cases.*

The following Y level declaration shows how derivative functions might be bound to unconnected ports. `Phase1` is the name of a phase; `i` and `j` specify the section's coordinates; `N` and `S` are port names; and `no_op()` is a user-written stub that simply returns. Section 5.3 shows how this declaration scales automatically as the number of rows and columns changes.

```
Phase1[i][j].port.N  send  <->  no_op()  where i=0
Phase1[i][j].port.S  send  <->  no_op()  where i=rows-1
```

5 MGS Example

This section illustrates our ideas using the MGS application. We describe the sequential algorithm, sketch three parallel algorithms, and compare their performance on various machines. After showing how our model allows a single program to implement all three algorithms, we measure the overhead of our solution and show that this overhead can be removed with the help of configuration parameters.

5.1 The Modified Gram-Schmidt Method

The MGS method is one way to perform QR factorization, a computation that factors an $M \times N$ matrix into two matrices, Q and R, such that Q is orthonormal ($Q^TQ = 1$) and R is upper triangular. Figure 4 shows the sequential MGS algorithm [16]. Initially, `a` contains the input matrix. Upon completion, `a` contains Q and `r` contains R. The algorithm processes one column of the `a` matrix at a time. For each column, the `FindMax` phase computes the pivot: the minimum of all elements of the `d` array whose index is $\geq$ `k`. `SwapCols` then swaps the pivot column with the k^{th} column for `a`, `r` and `d`. `Orthogonalize` divides elements of the k^{th} column of `a` by the pivot. Finally, the last phase normalizes `a` by computing the inner product of the `k`th column with all remaining columns of `a`, and updating all remaining columns of `a` with the appropriate inner product.

```
GramSchmidt()
{
   for (j=0; j<N; j++) {
      d[j] = 0;
      for (i=0; i<M; i++)              /* InitD Phase*/
         d[j] += a[i][j] * a[i][j];
   }

   for (k=0; k<N; k++) {
      p = k;
      for (i=k+1; i<N; i++)            /* FindMax Phase*/
         if (d[p] < d[i])
            p = i;

      if (d[p]==0) {
         rank = k;
         break;
      } else {
         swap (d[k], d[p]);
         for (i=0; i<M; i++) {         /* SwapCols Phase*/
            swap (a[i][k], a[i][p]);
            swap (r[i][k], r[i][p]);
         }
      }
      r[k][k] = sqrt(d[k]);            /* Compute Pivot */

      for (i=0; i<M; i++)
         a[i][k] = a[i][k]/r[k][k]; /* Orthogonalize */

      for (j=k+1; j<N; j++) {          /* Normalize Phase*/
         r[k][j] = 0;
         for (i=0; i<M; i++)
            r[k][j] += a[i][k]*a[i][j];

         d[j] = d[j] - r[k][j]*r[k][j];

         for (i=0; i<M; i++)
            a[i][j] = a[i][j] - a[i][k]*r[k][j];
      }
   }
}
```

Figure 4: The Modified Gram Schmidt Method

5.2 Parallel MGS Algorithms

Our parallel algorithms follow the structure of Figure 4. We describe three parallel algorithms that decompose the `a` and `r` matrices in three different ways.

Cached Rows. The Cached Rows algorithm uses a blocked rows decomposition (see Figure 5a). This decomposition induces communication in only two phases, `InitD` and `Normalize`, where values of the `a` matrix must be summed and broadcast along a single column (see Figure 6a). Each element of the `d` array corresponds to one column of the `a` matrix, so distributing the `d` array would lead to communication in the `FindMax` and `SwapCols` phases. Thus we replicate the `d` array at the small cost of additional storage.

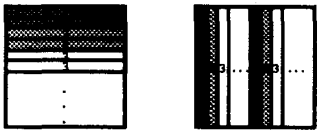

Figure 5: Data Decomposition: (a) Blocked Rows (b) Interleaved Columns

In a naive parallelization of the sequential code, a row decomposition requires a global sum and broadcast for each iteration of the `Normalize` phase. We move these operations outside the inner loop so that only O(M) communication operations are needed; partial values must be cached to perform this optimization.

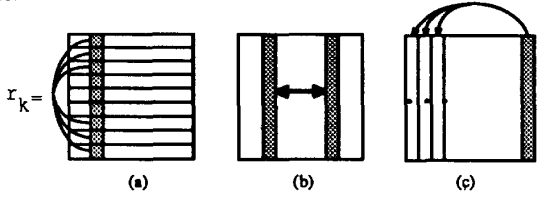

Figure 6: (a) Global Sum & Broadcast across Rows (b) Swap Columns (c) Broadcast for Columns

Interleaved Columns. Blocked columns would lead to severe load imbalance, so we interleave columns as shown in Figure 5b. This decomposition induces communication when swapping columns (`SwapCols`) and when broadcasting the pivot (`Normalize`) (see Figures 6b and 6c). In the `Orthogonalize` phase all processes are idle except the owner of the k^{th} column. We distribute the `d` array, which results in additional communication in the `FindMax` phase but avoids costly global sums and broadcasts of `d[j]` inside the inner loop of the `Normalize` phase. This algorithm has $O(N)$ broadcasts and swaps.

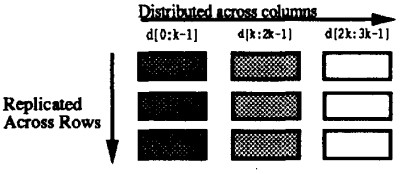

Figure 7: Data Decomposition for 2D Algorithms

2D CRIC. A two-dimensional decomposition induces communication across both rows and columns. The best 2D algorithm for MGS is a combination of Cached Rows and Interleaved Columns. We call this algorithm the Cached Rows/Interleaved Columns (CRIC) algorithm [7]. Columns of `a` and `r` are interleaved and rows are blocked (see Figure 7). As with Interleaved Columns, the `d` array is distributed across columns; as with Cached Rows, the columns of the `d` array are replicated across rows.

Experimental Results. We implemented the above algorithms by hand compiling the concepts of the Phase Abstractions. Details of the experimental methodology and additional results can be found elsewhere [7]. Figure 10 compare the algorithms on four machines. Our main observation is that even comparing the two 1D decompositions, *neither is best for all machines.* Thus, a portable solution to this problem should have the flexibility to accommodate either decomposition.

5.3 Our MGS Solution

The above results compared separate implementations of the three algorithms. Our proposed approach creates a single implementation of the 2D CRIC algorithm that can degenerate to either Cached Rows or Interleaved Columns by changing load time parameters. The program sketch in Figure 8 shows three configuration parameters – `rows`, `cols` and `Processors`. The user-defined `parse()` function reads the command line arguments and sets the appropriate values of `shape` and `Processors`. The `partition2D()` function computes a 2D array for a given number of processors. This program can now execute on the Butterfly, for example, where Cached Rows is best, by invoking the program as follows:

```
CRIC -r -P16
```

A columns decomposition is invoked as follows:

```
CRIC -c -P16
```

No source code changes are necessary even though the program specifies communication that is superfluous for 1D decompositions. Derivative functions convert

```
#define Rows 1
#define Cols 2
#define TwoD 3

program CRIC (argc, argc)

  /* Configuration Computation Section */
   parse(argc, argv);
   switch (shape) {
      case Rows: rows = Processors;
                 cols = 1;
                 break;
      case Cols: rows = 1;
                 cols = Processors;
                 break;
      case TwoD: partition2D (&rows, &cols, Processors);
                 break;
   }

(rows, cols, Processors): /* Configuration Parameters */

  /* Y level ensemble declarations ... */
  /* X level process definitions ... */
begin
  /* Z level body of code ... */
end
```

Figure 8: Configuration Computation Section.

the extraneous communication invocations into no-op's. As defined in Section 4.1, the binding of derivative functions to ports scales properly because it uses values of `rows` and `cols` from the configuration section, e.g. the South port is bound to `no-op()` whenever the process' row ID is equal to `(rows-1)`, which has the correct value for any of our three data decompositions. Of course, the configuration section may be more sophisticated. For example, the code itself may compute the "best" data decomposition based on input parameters and machine characteristics.

5.4 The Cost of Generality

Cost of Generality (time in msecs)

P	CRIC as rows	Cached Rows	% Difference
4	11197	11140	0.51
8	7186	6820	0.54
12	6387	5714	11.62
16	6183	5218	18.49

Figure 9: Overhead of CRIC Solution for Rows Decomposition on the Intel iPSC/2

General solutions are typically less efficient than customized ones. Figure 9 shows the cost of this generality by comparing the performance of the hand coded Cached Rows implementation against the CRIC implementation that degenerates to a rows decomposition. This overhead is incurred *per process* and cannot be removed through added parallelism. Thus, in an analog to Amdahl's law, this overhead is significant because it limits speedup.

5.5 Partial Evaluation

Partial evaluation (PE) can eliminate the above overhead through techniques that convert a general program to a more efficient, less general one [2, 3]. Due to difficulties with pointers and aliasing, PE has typically been applied to functional rather than imperative languages. A classic problem with PE is determining how much optimization to perform. For example, how many recursive calls should be inlined? We avoid these problems by using PE to remove a very restricted type of overhead. Our basic technique is constant folding, which is aided by the existence of configuration parameters that identify "constants." The overhead we wish to remove often appears in conditionals that test whether communication operations should be invoked based on such values as the process' row number, the number of processes in a given row, and the existence of neighbors. These values do not change during the execution of the program, and these expressions are usually computed directly from configuration parameters.

We have not yet implemented this partial evaluator, but a hand simulation of our algorithm shows that all of the overhead shown in Figure 9 can be removed. Of course, it is possible that the partial evaluator will optimize additional sources of inefficiency.

6 Conclusion

Explicitly parallel languages tend to be performant but not convenient. One aspect of convenience is the ease with which a program can adapt to architectural diversity. Using the MGS method as an example, we have shown that polymorphic data decompositions are an important aspect of portability and scalablity; polymorphism can be built into explicitly parallel programs through *derivative functions*; and supporting polymorphism can increase execution overhead, but this overhead can be removed through the use of *configuration parameters* and partial evaluation.

The last two points move us closer to our goal of decoupling data decomposition from communication while retaining the performance benefits of explicit communication. Our partial evaluation technique can also be used to transform SPMD code into MIMD code so that programmers can write with the convenience of the SPMD approach and the efficiency of an MIMD approach. This is an avenue of future research.

References

[1] G. Alverson, W. Griswold, D. Notkin, and L. Snyder. A flexible communication abstraction. In *Proc. of Supercomputing '90*, Nov. 1990.

[2] A. Berlin. Partial evaluation applied to numerical computation. In *Proc. of the 1990 Conference on Lisp and Functional Prog.*, Nice, France, 1990.

[3] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, Dec. 1990.

[4] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran – a Fortran language extension. TR 91-72, ICASE, Sept. 1990.

[5] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proc. of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, SC.

[6] High Performance Fortran Forum. *High Performance Fortran Specification.* Jan. 1993.

[7] C. Lin. *The Portability of Parallel Programs Across MIMD Computers.* PhD thesis, U. Wash., Dept. of Computer Science and Engr., 1992.

[8] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proc. of the Int'l. Conf. on Parallel Processing*, pp. II 163–180, 1990.

[9] C. Lin and L. Snyder. A portable implementation of SIMPLE. *Int'l. Journal of Parallel Programming*, 20(5):363–401, 1991.

[10] C. Lin and L. Snyder. Data ensembles in Orca C. In *5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[11] T. A. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. In *The Scalable High Performance Computing Conference*, 1992.

[12] D. Notkin, *et al.*. Experiences with Poker. In *Symp. on Parallel Programming: Experience with Applications, Languages, and Systems*, Jul. 1988.

[13] S. Otto. MetaMP: A higher level abstraction for message-passing programming. TR CS/E 91-003, Oregon Grad. Inst. of Science and Tech., 1991.

[14] R. Schreiber. The scalability of sparse direct solvers. In J. George, J. Gilbert, and J.Liu, eds., *Sparse Matrix and Graph Algorithms.* IMA, 1992.

[15] L. Snyder. Applications of the "Phase Abstractions." In J. Saltz and P. Mehrotra, eds., *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pp. 79–102. North Holland, 1992.

[16] E. Zapata, J. Lamas, F. Rivera, and G. Plata. Modified Gram-Schmidt QR factorization on hypercube SIMD computers. *Journal of Parallel and Distributed Computing*, 12:60–69, 1991.

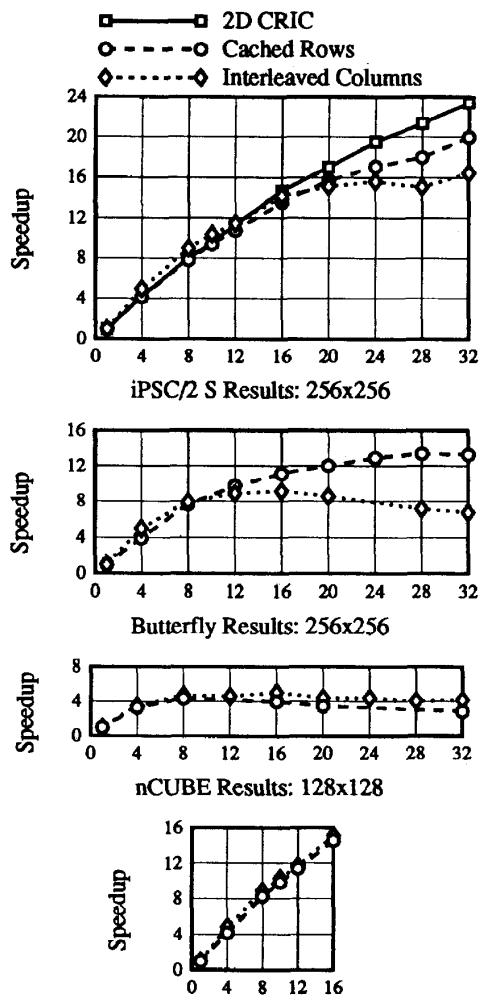

Figure 10: Speedup vs. Number of Processors.