The Good News About Dynamic Object-Oriented Parallel Processing

Andrew S. Grimshaw,W. Timothy Strayer,Padmini Narayan

Technical Report No. CS-92-41
December 17, 1992

This work is partially funded by NSF grant ASC-9201822.

The Good News About Dynamic Object-Oriented Parallel Processing[1]

Andrew S. Grimshaw
W. Timothy Strayer
Padmini Narayan

Abstract

Mentat [1] is an object-oriented parallel processing system that addresses the need for easy-to-use tools for the construction of portable medium-grain parallel software. Three of the key features of the Mentat approach are that it is object-oriented, that it uses a layered virtual machine, and that it dynamically constructs data dependence program graphs and dynamically manages system resources and communication. It has been argued that the overhead of these three features would be so large that the performance gains over sequential execution would be negligible. We have evidence to the contrary. In this paper we show that these features do not inherently imply poor performance. We introduce Mentat, its object model, and those features that most strongly affect performance. We then examine the performance of four applications under Mentat on two different platforms. The four applications were selected because they span the spectrum of communication and synchronization requirements. The results show that a dynamic, object-oriented parallel programming environment can offer both a hospitable programming environment as well as good performance for a wide range of applications.

1. Introduction

Writing software for parallel processing systems has proven to be more difficult than writing sequential software. In the early 1980's it was believed that automatic parallelization tools could be constructed that would automate the process of turning "dusty deck" sequential programs, usually written in FORTRAN, into parallel codes [2]. These tools have been somewhat successful when targeted to tightly coupled shared memory systems, yet the results are disappointing for the greater challenge of the distributed memory systems that have come to dominate the field of parallel processing. The reasons for this are complex, but a critical factor is their inability to detect medium-grain parallelism[2] as demanded by distributed memory architectures. As a result, the bulk of "production" parallel programs are written by hand, typically using send and receive primitives as language extensions.

Writing distributed memory parallel programs by hand is difficult. The programmer must manage communication, synchronization, and scheduling of tens to thousands of independent processes. The burden of *correctly* managing the environment often overwhelms programmers, and requires a

1. This work is partially funded by NSF grant ASC-9201822.
2. They can detect medium-grain, loop-level data-parallelism, but then the data distribution problem arises.

considerable investment of time and energy. This has led to the perception in the parallel processing community that programming parallel machines is subject to a “no pain, no gain” rule: to reap the benefits of parallelism one must first suffer the pain of hand coding the application.

A second, and related, problem is software portability. Once implemented on a particular MIMD architecture, the resulting codes are often not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. As a consequence, considerable effort must be re-invested to port the application to a new architecture. Given the plethora of new architectures and the rapid obsolescence of existing ones, this discourages users from parallelizing their code in the first place. Until these problems are solved, parallel architectures will remain outside of the mainstream computing community. This is unfortunate since parallel machines offer the promise of substantially improved performance over their sequential counterparts.

The object-oriented paradigm has proven to be a powerful software engineering tool for sequential software. Attributes such as programming in the large, encapsulation, polymorphism, fault containment, and software reuse have all made the task of constructing complex sequential software more tractable. The same attributes can help manage the complexity of parallel software. Yet for parallel processing, the key issue is performance: how do the properties of the paradigm, particularly polymorphism and encapsulation, affect performance? Is the cost of supporting the paradigm so great as to negate the benefits of parallelism, defeating the high-performance objective of parallel processing systems? We believe that the “no pain, no gain” rule stems from the tools that have been used, and is not intrinsic to parallel processing. By combining the object-oriented paradigm with parallel processing compiler and run-time system technology, the task of writing high-performance parallel software can be simplified.

Mentat [1] is an object-oriented parallel processing system that addresses the need for easy-to-use tools that facilitate the construction of portable medium-grain parallel software. Mentat extends an existing object-oriented language, C++, with mechanisms that allow the programmer to indicate which application classes are sufficiently computationally complex to warrant parallel execution. The compiler and run-time system then take over and *dynamically* manage scheduling, communication, and synchronization for the programmer, freeing the programmer from the need to manage the difficult aspects of parallel computation.

Mentat achieves portability by providing a virtual machine interface to the compiler. The virtual machine implementation isolates architecture dependencies, facilitating migration to other platforms.

It has been argued that the overhead for using an object-oriented paradigm over a layered virtual machine that dynamically constructs data dependence program graphs and dynamically manages system resources and communication would substantially diminish performance. We argue that this is not true. While meeting our goals of ease of use and applications portability, we have achieved very good performance for a wide variety of applications on MIMD platforms that range from loosely coupled networks of workstations to tightly coupled multicomputers.

The object-oriented paradigm, a virtual machine, and dynamic management of parallelism do not necessarily lead to poor performance. We show this by introducing Mentat and its underlying object model, and we identify those features that most strongly affect performance. We then examine the performance of four applications under Mentat on two different platform, a network of Sun IPC SparcStations, and an Intel iPSC/2. The four applications are: (1) DNA and protein sequence comparison, (2) image convolution, (3) Gaussian elimination with partial pivoting on dense matrices, and (4) matrix-vector multiply on sparse matrices. These four applications were selected because they span the spectrum of communication and synchronization requirements, illustrating how Mentat performs on applications with various computation to communication ratios. The first application, DNA and protein sequence comparison, is essentially a data-parallel application. This is a highly data-parallel application with no communication between the multiple identical workers. The image convolution problem has more communication between workers, but the computation ratio is still very high. The Gauss solver requires frequent communication to select and distribute the next pivot. Finally, the sparse matrix-vector multiply requires tremendous amounts of communication and illustrates a problem with the object-oriented paradigm when applied to parallel computing—that encapsulation can be a hinderance because it can increase the amount of communication, driving computation granularity down.

2. Mentat Overview

The three primary design objectives of Mentat are to provide: (1) high performance via parallel execution, (2) easy-to-use parallelism, and (3) applications portability across a wide range of platforms.

The realization of these three objectives has been guided by our philosophy that there are some things that people do better than machines, and some things that machines do better than people. We want to exploit the comparative advantages of each. Specifically, given current compiler technology it is difficult for a compiler to take a “big picture” view of an application and decide what is computationally significant and which data structures and elements are tightly coupled. This difficulty follows from the graph representations of programs that compilers manipulate. To try to infer from the program graph what the program does, and to map computations and data structures to processors, are difficult tasks for a compiler. However, compilers and run-time systems are very good at enforcing rules, particularly communication and synchronization requirements that can be readily extracted from programs.

People, on the other hand, know their application, its data structures, and what is computationally expensive, i.e., they know the problem domain. Unfortunately, people are not good managers of the communication and synchronization requirements of parallel programs. As a response to the complexity of the environment programmers restrict what they attempt to do in parallel to very regular computations that synchronously replicate the same activity on many processors, essentially implementing coarse-grain SIMD. Consequently, many opportunities for parallelism are lost.

The Mentat philosophy is to recognize the comparative advantages of both compilers and people. The programmer specifies the computationally expensive components and their associated data structures, while the compiler and run-time system (RTS) manage the communication, synchronization, and scheduling. This permits programmers to concentrate on algorithmic and design aspects of the code, releasing them from the cognitive burden of managing the parallel environment.

2.1 Mentat Programming Language (MPL)

The Mentat philosophy is reflected in the Mentat Programming Language, MPL. Rather than invent a new language, MPL is an extension of the popular object-oriented language C++. The extensions are designed to facilitate communication from the programmer to the compiler and RTS. The extensions are how we encode programmer domain knowledge so that the compiler and RTS can make better decisions.

```
1  mentat class bar {
2  // private member functions and variables
3  public:
4     int function1(int);
5     int function2(int, int);
6  };
```

Figure 1—A Mentat class definition

The most important extension is the keyword `mentat` as a prefix to class definitions, as shown on line 1 of Figure 1. The keyword `mentat` indicates to the compiler that the member functions of the class are computationally expensive enough to be worth doing in parallel. Mentat classes are further defined to be either *regular* or *persistent*. `Regular` Mentat classes are stateless, and their member functions can be thought of as pure functions in the sense that they maintain no state information between invocations. As a consequence, the run-time system may instantiate a new instance of a `regular` Mentat object to service each invocation of a member function from that class.

Persistent Mentat classes, on the other hand, do maintain state information between member function invocation. Since state must be maintained, each member function invocation on a persistent Mentat object is served by the same instance of the object. This instance has a system-wide unique name and an independent thread of control. Member functions for each instance are executed one at a time, in a monitor-like fashion.

Member function invocation on Mentat objects is syntactically the same as for C++ objects, but semantically there are three important differences. First, Mentat member functions are always call-by-value, since the model does not assume shared memory. Second, Mentat member function invocations are non-blocking, providing for the parallel execution of member functions whenever data dependencies permit. This is transparent to the user and is called inter-object parallelism encapsulation. Finally, each invocation of a `regular` mentat object member function causes the instantiation of a new object to service the request, further promoting inter-object parallelism.

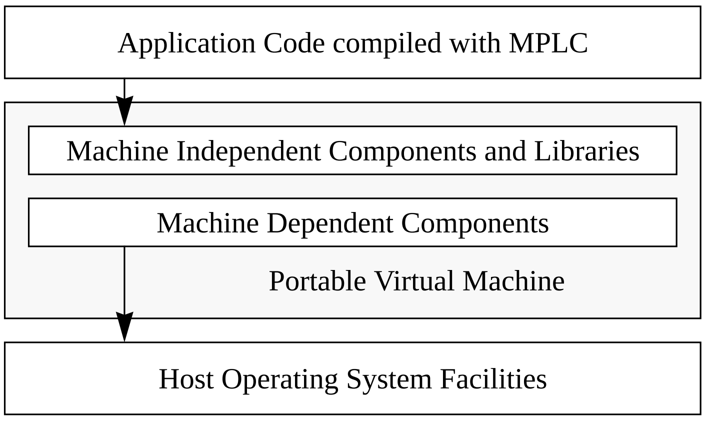

Application Code compiled with MPLC
Machine Independent Components and Libraries
Machine Dependent Components
Portable Virtual Machine
Host Operating System Facilities

Figure 2—Mentat virtual machine model

2.2 Sources of Overhead

Two of the key features of the Mentat implementation are that Mentat dynamically detects and manages parallelism and that Mentat uses a virtual machine model to achieve portability. These features relieve the programmer of the burdensome task of finding the parallelism, employing the language extensions to achieve the parallelism, and then translating from this parallel language dialect to another when porting the application. While Mentat's features help ease the programmer's job, they are not without cost. In general, there are four sources of overhead during the execution of a Mentat application: program graph construction, argument marshalling and transportation, argument matching, and object scheduling and instantiation.

Mentat uses a layered virtual machine abstraction to achieve portability, as shown in Figure 2. The virtual machine provides services for program graph construction, argument matching, Mentat object scheduling, Mentat object instantiation, and inter-object communication. The object model supported by the virtual machine supports the notion of processes with distinct address spaces and a single thread of control, where communication is via messages. Mentat objects map one-to-one onto processes in the virtual machine. The MPL compiler generates code that uses virtual machine features rather than native operating system process/thread and IPC facilities.

```
1  int w, x, y, z;
2  w = 0;
3  bar A, B, C; // bar is a Mentat class, A,B,and C are Mentat objects
4  x = A.function1(5); // x is an L-value
5  y = B.function1(x); // x’s value is needed on the right
6  if (w ==0) {
7     z = C.function2(x, y);
8  }
```

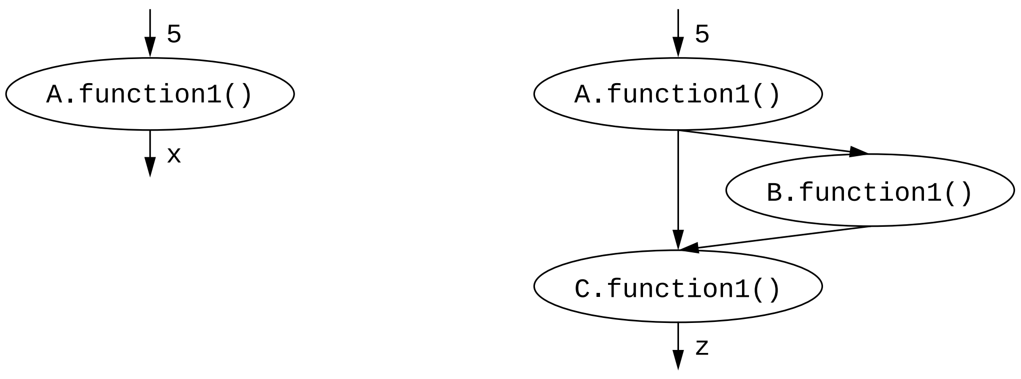
Figure 3—Code invoking a member function of a Mentat object

The virtual machine provides a mapping between the services required by the application and the services provided by the underlying operating system. Sometimes the mapping is simple and does little more than rearrange arguments. When the underlying operating system does not provide the needed service, the mapping is more complex. In any event, some overhead is introduced that often could have been avoided by having the compiler generate code specific to the host environment.

To perform dynamic data dependence detection, the MPL compiler emits calls to the RTS to monitor the use of *result variables*. A result variable is a variable that occurs on the left-hand side of a *Mentat expression*. A Mentat expression is an assignment statement in which the right-hand side is a member function invocation on Mentat object. For example, in the code fragment in Figure 3, line 4, `A` is a Mentat object, and `x` is a result variable. The RTS monitors all uses of `x` both as an L-value and as an R-value. When line 4 is executed the RTS constructs a program graph node corresponding to the member function invocation, as shown in Figure 4 (a). The node has a single input arc corresponding to the single argument, and a message containing the value 5 is associated with the input arc. When `x` is later used on the right-hand side as an argument to a Mentat object member function invocation, as on line 5, a new graph node is constructed and data dependence arcs are added. Similarly, if the expression on line 6 is true, a new graph node is constructed and new arcs are added. The program graph after the execution of line 7 is shown in Figure 4 (b). In this manner the program dynamically unfolds with the control flow functions that are being executed in the local address space determining the program graph structure. Furthermore, the arguments are marshalled and delivered to the appropriate functions as needed, based on this data dependency graph. This process is completely transparent to the programmer. Once the graph is constructed, its execution is managed by the RTS.

Overhead is also incurred in the invoked objects. When messages containing the arguments arrive on the input arcs they must be matched with messages that have already arrived to determine if all of the arguments are available. Input guards must also be evaluated against the messages that have arrived. Once a complete set of messages is available, the specified member function is executed and the return results are forwarded (in messages) along all of the out-going arcs. In the case of `regular` objects, message matching is performed by the underlying RTS outside of any object's address space.

Thus, the sources of overhead encountered in a code fragment such as that in Figure 3 are the monitoring of result variables, the creation of graph nodes and arcs, and the marshalling and transportation of arguments. With the exception of the transportation of arguments, the cost of which depends on the size of the argument, the cost of these overhead operations is independent of the granularity of the Mentat member functions invoked. This implies that a performance slow-down will result if the invoked member functions do not take longer to execute than the sum of the overhead. On the other hand, as the computation complexity of the member functions increases, performance will improve if there is enough parallelism in the program graph. The cost of using Mentat, therefore, depends on the nature of the



(a) Graph after execution of line 4

(b) Graph after execution of line 8

Figure 4—Data dependence graphs for code in Figure 3

application. Mentat works well on applications with sufficient medium-grain parallelism to help amortize these overhead costs.

3. Performance Results

Our objective is to show that the use of the object-oriented paradigm, a virtual machine, and the dynamic management of parallelism does not necessarily lead to poor performance. We have chosen as the vehicle of our discussion four applications on two different parallel platforms. The applications are DNA and protein sequence comparison, image convolution, Gaussian elimination with partial pivoting on dense matrices, and matrix-vector multiply on sparse matrices; they span the granularity and synchronization spectrum. The platforms are a 32-node Intel iPSC/2 and a network of Sun IPC SparcStations. The platforms were chosen because they have different topologies, they have different optimal grain sizes, they are each an archetype of a class of parallel machines, and they are familiar to many readers.[3] We begin our discussion by first describing our performance metric and test apparatus in detail. Then we briefly outline the problem that each application is intended to solve, and how we have implemented the application within Mentat. We also make note of the important attributes of the application, especially with respect to how they help form the solution. We then offer performance results and observations.

3.1 Overview and Testing Methodology

Our performance metric is the dependent variable *speedup*. The independent variables in the experiments are the number of workers (or processors) applied to the problem and the size of the problem. The interplay of these three variables highlights the importance of granularity and clearly shows the trade-off.

The way speedup is calculated varies widely. Two common equations are:

$$\text{Speedup} = \frac{T_1}{T_p} \quad (a) \qquad \text{Speedup} = \frac{T_{\text{sequential}}}{T_p} \quad (b) \qquad \text{(Eq 1)}$$

3. Mentat has been ported to several other platforms as well, including the iPSC/860, other networks of workstations, and the TMC CM-5.

The difference is in the numerator: T_1 is the time to execute the parallel program on one processor, while $T_{\text{sequential}}$ is the time to execute an equivalent sequential program on one processor. The denominator, T_p is the time to execute the parallel version of the program on p processors. The speed of the processor is held constant. Because the competition to parallel computing is sequential computing, we feel that it is unrealistic to use Equation 1 (a) since T_1 includes all of the overhead of parallel computing and none of the benefit. Equation 1 (b) represents the payoff for choosing a parallel implementation over a sequential one, which is more in line with what a potential user of parallel computing will consider. In our analysis we use this second definition of speedup.

To find $T_{\text{sequential}}$ we execute an equivalent C or C++ program (no parallel constructs at all) on an idle target platform and measure the *wall clock time* once the program has been loaded and the arguments parsed. We include in the wall clock time any necessary I/O for the program. The parallel Mentat execution times are measured in the same manner. Note that this implies that the Mentat objects that actually do the work have not been loaded, nor their data delivered, when the timers are started. Thus, all overhead is included. In the same vein, we have been very careful to use the same level of hand optimization of inner loops, and the same level of compiler optimization for both the C++ and Mentat versions. In this way we make the fairest possible comparison.

To calculate the speedup for each application and architecture we used the best times and not the averages. This is done to demonstrate the capabilities of Mentat. For the hypercube, where the whole machine is devoted to the Mentat application, this makes little difference. The network of workstations, however, cannot be dedicated in that way, so we try to factor out transient interference; the best times reflect more closely a dedicated system.

Some of the irregularities seen in the graphs are due to the algorithm used by the Mentat scheduler [3]. This scheduler is simplistic and does not perform well at high loads or when the number of available processors is less than the number of processes resulting from the decomposition of the application.

3.2 Platforms

We conducted experiments on two different platforms, a network of sixteen Sun IPC SparcStations and a 32-node Intel iPSC/2 hypercube. These two platforms are distinguished by their processor computational capabilities and their interconnection network characteristics. The sixteen Sun IPC's are connected by thin Ethernet and are served by a Sun 4/260 file server. Each IPC has 16 Mbytes of memory and a local disk for swap space.

The Intel iPSC/2 is configured with 32 nodes (five dimensions). Each node has 4 Mbytes of physical memory and an 80387 math co-processor. The hypercube also has a 1.5 Gbyte Concurrent File System with four I/O nodes. The NX/2 operating system provided with the iPSC/2 does not support virtual memory; this, coupled with the amount of memory dedicated to the operating system, limits the size of the problems that can be run on the iPSC/2.

3.3 DNA and Protein Sequence Comparison

Our first application, DNA and protein sequence comparison, is trivially parallel and requires very little communication. The problem can be easily decomposed into separate functions where a master object can start each function and wait for the results.

With the advances in DNA cloning and sequencing technologies, biologists today can determine the sequence of amino acids that make up a protein more easily than they can determine its three-dimensional structure, and hence its function. The current technique used for determining the structure of new proteins is to compare their sequences with those of known proteins. DNA and protein sequence comparison involves comparing a single query sequence against a library of sequences to determine its relationship to known proteins or DNA. For the computer scientist, the basic problem is simple: DNA and protein sequences can be represented as strings of characters that must be compared. Biologists want to know the degree of relationship of two sequences. Two given sequences are compared and, using one of several algorithms, a score is generated reflecting commonality. Three popular algorithms are Smith-Waterman [4], FASTA [5], and Blast [6]. The latter two algorithms are heuristics; the quality of the score is traded for speed. Smith-Waterman is the benchmark algorithm, generating the most reliable scores although at considerable time expense. FASTA is less accurate but is twenty to fifty times faster than Smith-Waterman.

```
regular mentat class sw_worker {
   // private member data and functions
public:
   result_list *compare(sequence, libstruct, paramstruct);
   // Compares sequence against a subset of the library. Returns
   // a list of results (sequnce id, score).
}
```

Figure 5—Class definition for scanlib worker.

An important attribute of the comparison algorithms is that all comparisons are independent of one another and, if many sequences are to be compared, they can be compared in any order. This natural data-parallelism is easy to exploit and results in very little overhead.

A common operation is to compare a single sequence against an entire database of sequences. This is called the *scanlib* problem. In scanlib, the source sequence is compared against each target sequence in the sequence library. A sorted list of scores is generated and the sequence names of the top *n*, usually 20, sequences and a score histogram are generated for the user.

The Mentat implementation of scanlib uses `regular` Mentat class workers to perform the comparisons. The class definition for the Smith-Waterman worker is given in Figure 5. The private member variables have been omitted for clarity. The single member function, `compare()`, takes three parameters: `sequence`, the source sequence to compare, `libstruct`, the structure containing information defining a subrange of the target library, and `paramstruct`, a parameter structure containing algorithm-specific initialization information. The member function `compare()` compares the source sequence against every sequence in its library subrange and returns a list of result structures. Each result structure has a score and the library offset of the corresponding sequence.

The important features of the main program are shown in Figure 6. Note that we only had to declare one worker, and that the code from lines 7-10 looks as though the single worker is being forced to do its work sequentially. Recall, however, that since the worker is a `regular` Mentat class, each invocation instantiates a separate copy of the worker object, so each copy is actually doing the comparisons in parallel.

Performance on the two platforms is shown in Figure 7. A 3.8-Mbyte target library containing 9633 sequences was used. CCHU, RSMD, LCBO, and RNBY3L are four different source sequences. They are 104, 229, 229, and1490 bytes long respectively. Comparison time of Smith-Waterman is linear in the sequence lengths.

Overhead Analysis

The program graph for this application is a simple fan-out/fan-in graph with three edges, each corresponding to a worker argument, from the main program (master) to each of the workers, and a single edge from each worker to the main program. The sources of overhead are building the program graph, sending the arguments to the workers, and scheduling and instantiating the workers. Each of these overhead costs is a one time cost in this application. The total overhead cost is very small when compared to the time to perform the comparisons for all but the smallest libraries.

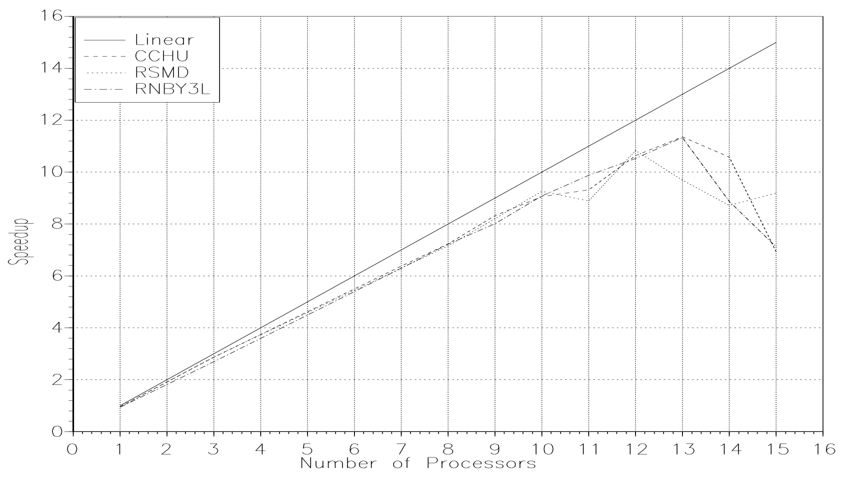
Performance

Performance on this application clearly demonstrates that for naturally data-parallel applications with no inter-worker communication and little worker-master communication, neither the object-oriented paradigm nor dynamic management of parallelism seriously affect performance. Indeed, in [7] we show

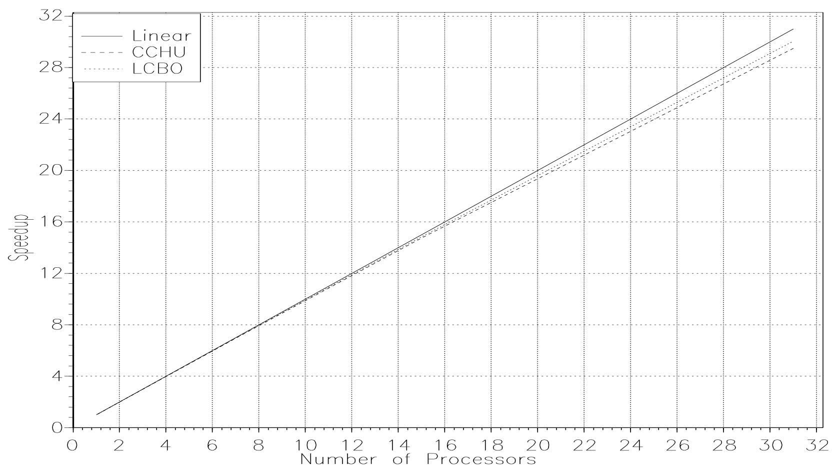
```
// master program routine -- initialization details omitted
// -- get the number of workers
// -- divide the library into a partition for each worker

sw_worker worker;
// invoke the comparison for each partition of the library
for (i = 0; i < num_workers; i++) {
   // compute library parcel boundaries
   results[i] = worker.compare(the_seq, libparcelinfo, param_rec);
}
// for each partition’s result, and for each comparison
// within the partition, compute statistics
for (i = 0; i < num_workers; i++) {
   for (j = 0; j < results[i]->get_count(); j++) {
      // for each result
      // record mean, stdev, and histogram information
   }
}
```

Figure 6—Code for the master program routine for scanlib.



(a) Smith-Waterman on the Sparc IPC network



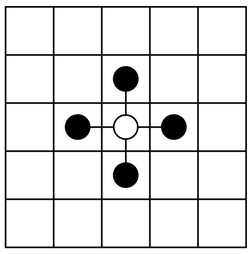
(b) Smith-Waterman on the Intel iPSC/2

Figure 7—Speedup for DNA sequence comparison.

that the performance of the Mentat version is always within 10% of a handcoded version of the same application on the Intel iPSC/2, and usually within 5%.

3.4 Image Convolution (a stencil algorithm)

Our second application is image convolution. Image convolution was chosen because it is an intuitive member of a class of algorithms called *stencil algorithms*. Other stencil algorithms include Jacobi iteration and many partial differential equation solvers [8]. In stencil algorithms, the next value of a point in space is a function of its neighboring points. Exactly which neighbors contribute to the next value of a



$$b[i,j] = \frac{a[i,j] + a[i+1,j] + a[i,j-1] + a[i,j+1] + a[i-1,j]}{5}$$

Figure 8—A five point stencil in two dimensions, with an averaging stencil function.

given point is determined by the stencil. A five point stencil on a two dimensional space is shown in Figure 8. This stencil is an average of values in the north, south, east, and west neighbors and the value of the point itself. Image convolution is used to “clean up” an image by the successive application of a set of stencil functions called filters. Distortion is filtered out from images to obtain progressively better images. It is a computationally expensive application with relatively few synchronization points.

In the general image convolution problem we compute a new image *g* from an old image *f* using a filter *h*. If *f* is a *p*×*q* image, and *h* an *m*×*n* filter, then for all *i* from 1 to *p*, and *j* from 1 to *q*, *g* is given by:

$$g[i,j] = \sum_{a=-\frac{m}{2}}^{\frac{m}{2}} \sum_{b=-\frac{n}{2}}^{\frac{n}{2}} \frac{f[i+a, j+b] \times h[a,b]}{m \times n} \qquad \text{(Eq 2)}$$

The computation complexity for square images and filters is $O(n^2p^2)$.

Parallel implementations of convolution often divide the image up into blocks or strips, as shown in Figure 9, with each processor having responsibility for one piece. When the image is divided, a guard region, sometimes called a boundary region, must be maintained by each worker. The size of the guard region depends on the largest filter that may be applied. If multiple filters are to be used, then each worker must obtain an up-to-date copy of its guard region from its neighbor workers. This exchange of guard regions is the interesting aspect of the problem. The workers must synchronize and communicate between every filter application. There are alternate methods for image convolution that do not require this communication, but instead perform redundant computation. However, those techniques are not applicable to the general stencil problem.

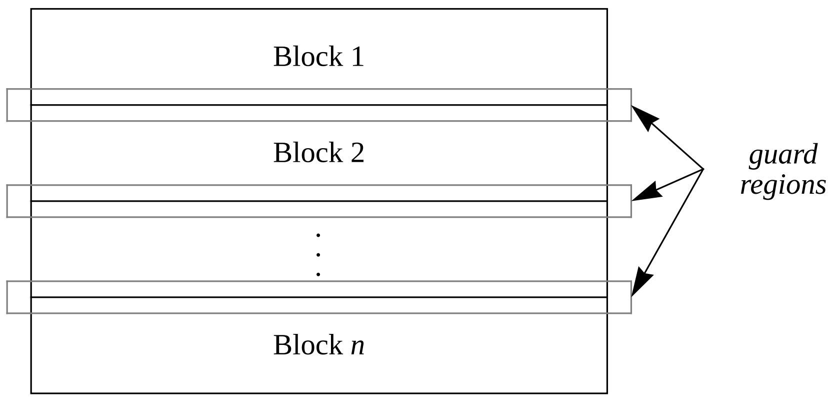


Figure 9—Horizontal decomposition of source image, with boundary regions shown.

The Mentat implementation uses two persistent Mentat classes. Their definitions are given in Figure 10. As before, private members have been omitted for clarity. We have chosen a master-worker

```
persistent mentat class Master_convolver {
// private member variables and functions
public:
   void initialize(string *source_file, string *target_file);
   void add_filter(DD_chararray *filter);
   int convolve();
}

persistent mentat class Worker_convolver {
// private member variables and functions
public:
   int convolve(DD_chararray* filter);    // convolve by filter
   void set_top(DD_chararray *top);       // boundary region top
   void set_bottom(DD_chararray* bottom); // boundary region bottom
   DD_chararray get_top();                // get top boundary region
   DD_chararray get_bottom();             // get boundary region
}

// Master_convolver’s convolve definiton

int Master_convolver::convolve() {
   // for each filter do:
   // -- tell workers to exchange boundaries with neighbors
   // -- tell workers to convolve
   // tell workers to store results in the target file
}
```

Figure 10—Persistent Mentat class definitions for master and workers for convolver.

model in which a master object manages the efforts of several worker objects. The class of the master object is `Master_convolver`. The class of the workers is `Worker_convolver`. Image convolution proceeds as follows. First, the user instantiates an instance of `Master_convolver` class and initializes it with the name of the source image file and the target image file. The master then instantiates the workers and provides each with a region of the image. We have chosen a horizontal decomposition for simplicity. Then, for each filter, the master has the workers convolve their sub-image and exchange guard regions with their neighbor workers. Finally the workers are instructed to write out the file and terminate.

Overhead Analysis

The effect of the interchange in the master object convolution function is to create three sets of short program graphs that pass guard regions from each worker to its neighbors and perform the convolution. This is shown in Figure 11. Note that although there is much communication, the master does not wait for the convolution of one iteration to complete before issuing commands to perform the next set

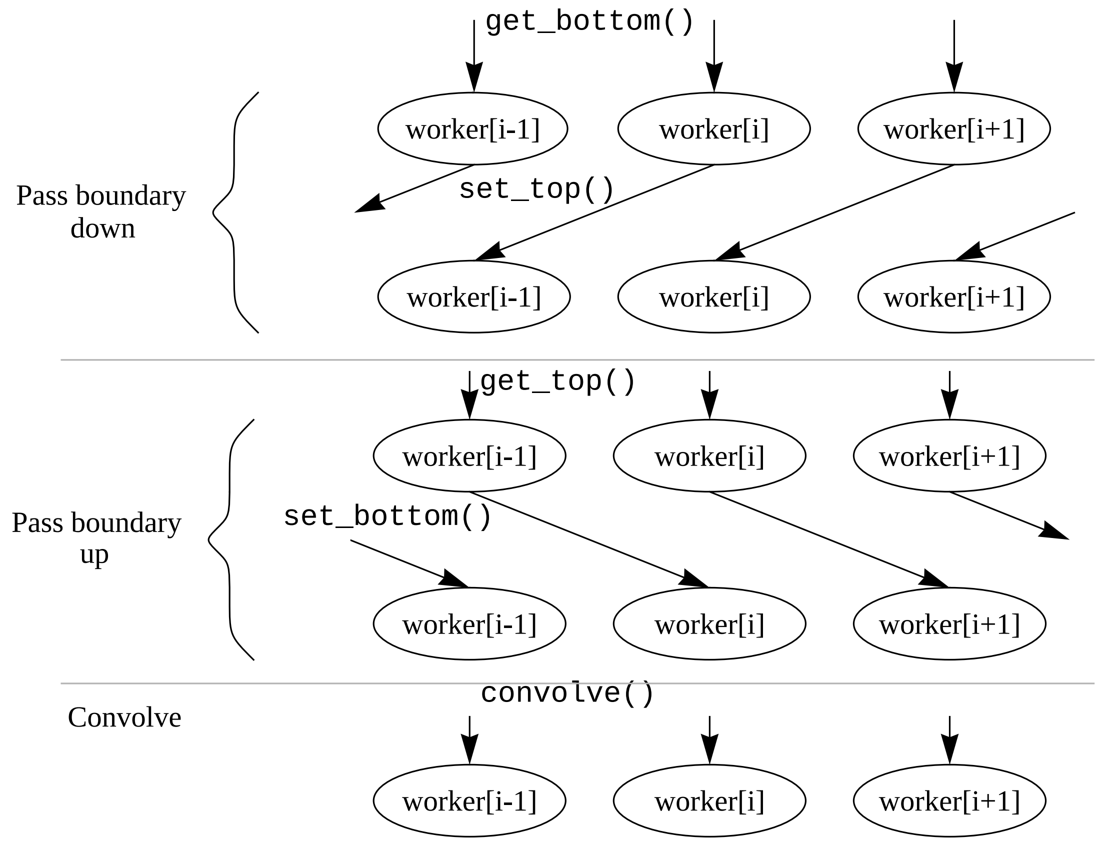


Figure 11—Subgraphs for one iteration of `convolve()`.

of exchanges and the convolution. These subsequent commands are queued up and executed in order. The overhead to accomplish the graph construction and the communication is considerably higher than in scanlib. Although the initialization overhead is comparable, for each filter there are $5k+1$ messages (one message to the master, and four messages to each worker[4]), and $2n\lfloor p/q \rfloor$ bytes transferred between workers.

Performance

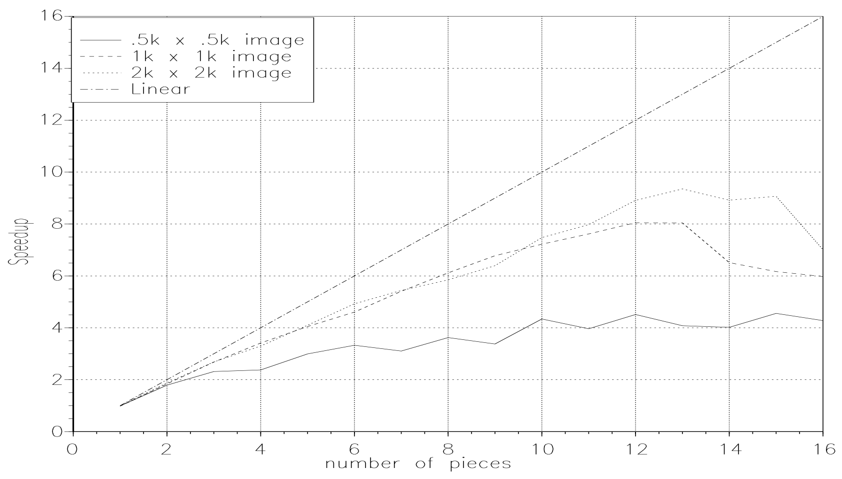
For small images and small filters the overhead of communicating the guard regions is large relative to the amount of work done. However, for realistic size images, e.g., 512×512 and up, we have found this not to be a problem. Figure 12 presents performance data for convolving 512×512, 1024×1024, and 2048×2048 two dimensional electrophoresis gel images obtained from the National Cancer Institute [9]. We cannot positively explain the “bumps” in the iPSC/2 graph. We believe that they are an artifact of placement decisions made by the scheduler. Currently the scheduler does not use application topology information in placing objects. This can result in poor placement decisions.

Performance on the iPSC/2 indicates that overhead on the larger problems is not significant. The results clearly show that the overhead associated with the object-oriented paradigm, dynamic construction of program graphs, and the use of a virtual machine did not significantly affect performance. A comparison with the Sparc IPC times indicates that bandwidth is an issue.

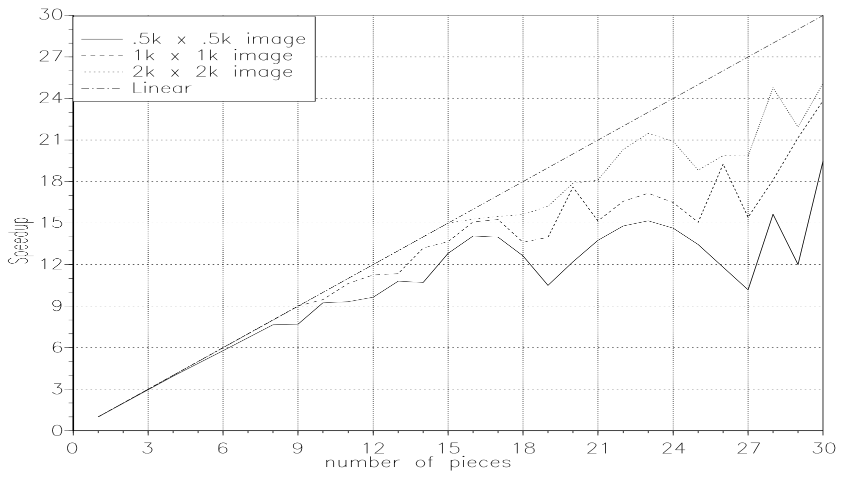
3.5 Gaussian Elimination with Partial Pivoting

Our third application, the solution of a system of linear equations using Gaussian elimination with partial pivoting, was chosen because it exhibits frequent synchronization, a large amount of communication, it is well-known to readers, and performance results are available for other systems. Recall that the problem is to solve a system of linear equations, i.e., to solve for x in $Ax = b$, where A is an $N \times N$ matrix and b is a vector of size N. The algorithm consists of two phases to compute x, forward elimination in which the extended matrix is reduced to upper triangular form, and back substitution. The first phase is executed in parallel, the second sequentially.

4. Not all $5k+1$ messages are sent by the master. The master receives one message and sends $3k$ short messages. The remaining communication is solely between the workers.



(a) 16-processor Sparc IPC network



(b) 32-processor Intel iPSC/2

Figure 12—Speedup for Convolver using two 9×9 filters.

The Mentat implementation is as part of a library of linear algebra routines provided by the `regular` Mentat class `matrix_ops` (see Figure 13). The `matrix_ops::solve()` function takes two parameters, the `DD_floatarray* A`, and the vector `b`. We begin by partitioning `A` and `b` into horizontal strips and distributing the strips to workers. Each worker is responsible for a set of rows. The workers are instances of the persistent mentat class `matrix_sub_block`. The important member function of the class `matrix_sub_block` is `reduce()`. The function `reduce()` reduces the current column of the rows that it owns that are not already in their final form, selects a candidate next pivot row,

```
// matrix_sub_block’s reduce defintion

vector* matrix_sub_block::reduce(vector* pivot) {
   // reduce current column using pivot
   // find candidate row, (largest absolute value in current column)
   // reduce candidate row
   // return the candidate row (but don’t quit the function)
   // reduce remaining rows (computationally expensive)
}

// matrix_ops’ solve definition

void matrix_ops::solve(DD_floatarray *A, vector *b) {
   // create workers and distribute A and b by strips
   for (col = 0; col < N; col++) {
      for (i = 0; i < num_workers; i++) {
         // invoke reduce using best pivot
         //    to get candidate next pivot row
         // determine best pivot among candiates:
         //    row with largest absolute value
      }
   }
   // back subsitute determining x
   // return x to caller
   // destroy workers
   // exit
}
```

Figure 13—Two instrumental member functions for Gaussian elimination with partial pivoting.

reduces that row, returns the candidate row to the master, and then completes the computationally expensive reduction of the remaining rows.

Once the master has distributed the strips to the workers, it then iterates over the columns, selecting a candidate pivot row and reducing the workers by that row. At each iteration the master retains the pivot row as part of the final upper triangular matrix. After N iterations the matrix is in upper triangular form and the master has a copy. The master then sequentially performs the back substitution, returns the resulting vector `x`, destroys the workers, and exits.

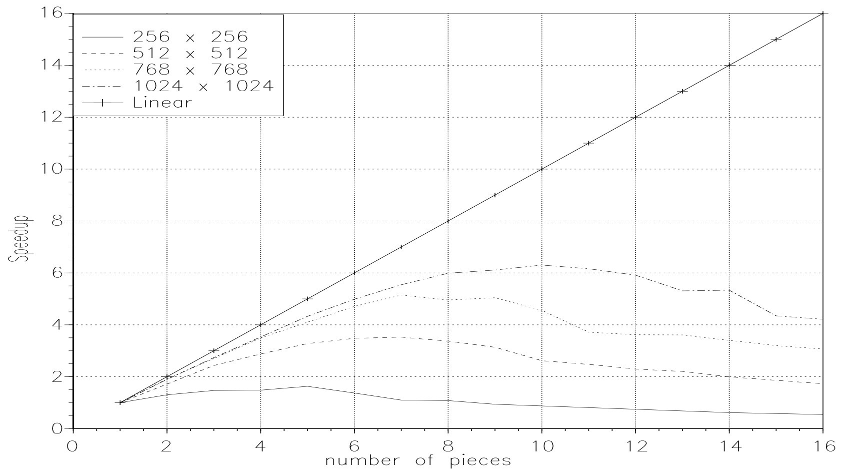
Overhead analysis

The program graph for this application is a classic fan-out, fan-in, fan-out, where finding the next pivot row, and then distributing it to the workers, dominates the overhead. Each iteration of the outer loop of the algorithm (executed *N* times) generates 2*k* messages, each containing a vector (albeit of decreasing size). The effect is that near the end of the algorithm, when most of the matrix has been reduced, the computation granularity becomes quite small and the overhead begins to dominate. Further, the total number of messages sent is 2*Nk*, i.e., it is linear in *N*. Similarly, 2*Nk* short program graphs are generated. The generation of each graph requires a small, constant amount of time. Given that the computation complexity of Gaussian elimination is $O(N^3)$, it is not surprising that for small *N* performance is dominated by communication, and thus is poor.

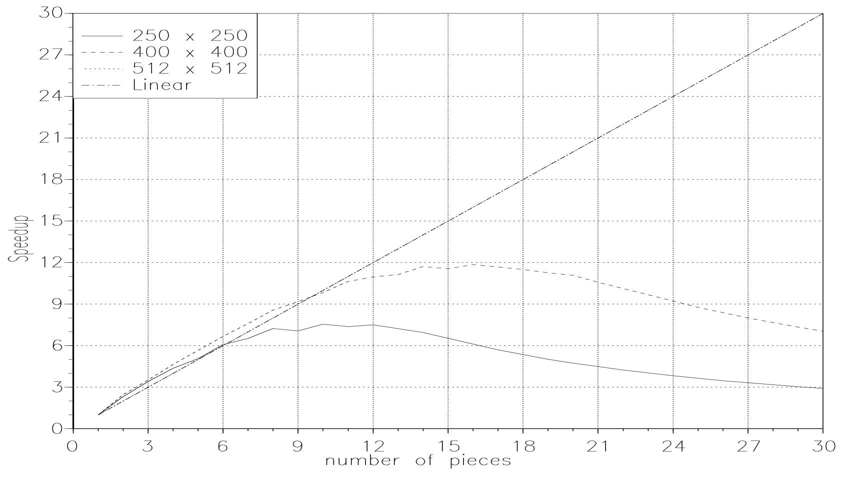
Performance

Performance for this application is shown in Figure 14 (a) and (b). Note that the problem sizes (matrix dimensions) for the two platforms are different. These two machines have very different floating point speeds and memory capacity. The four megabytes of RAM on the iPSC/2 limits the size of problem we can solve to very small problems. The performance curves for any particular size problem, e.g., 400×400 on the iPSC/2, follow the classic parallel processing pattern. At a low number of processors, and hence large granularity, speedup and processor utilization is good. As more workers are added speedup continues to improve at a reduced rate until a maximum is reached. At this point the problem has been partitioned into the optimum grain size for the architecture. Adding additional workers will decrease the grain size below optimum, and therefore, decrease speedup. This is clearly visible in the graphs. Currently, finding the optimum operating point is the programmers responsibility. In [10] we present techniques still in their formative stages that automate this process.

How significant are the overhead costs? If overhead costs, including communication, were zero, then the performance curves would not hit a maximum and then decline (as long as *N* is much greater than the number of workers). Since performance does drop-off, overhead is clearly significant. However, most of the overhead is the result of communication inherent to the algorithm and cannot be avoided, regardless of the underlying environment. In Section 4 we compare our performance on this problem with four other implementations, including a hand-coded version.



(a) 16-processor Sparc IPC network



(b) 32-processor Intel iPCS/2. The data for 512×512 on 32 processors is not available at this time. It will be available soon.

Figure 14—Speedup of Gaussian elimination with partial pivoting

3.6 Sparse Matrix-Vector Multiply

Our final application, sparse matrix-vector multiply, was chosen because it clearly illustrates the short-coming of the object-oriented paradigm when applied in distributed memory environments. The issue is one of granularity versus encapsulation: when good object-oriented encapsulation properties are employed the granularity may become too small for good performance. We begin by first introducing sparse matrix-vector multiply and one of its primary uses. We then sketch our implementation and analyze the overhead costs.

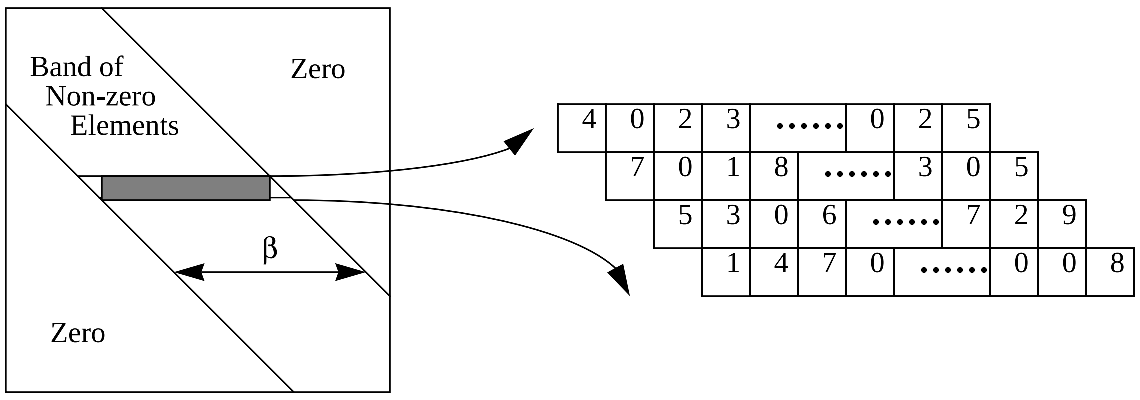


Figure 15—The band structure of a sparse matrix. On the left is a typical banded sparse matrix. On the right is an expanded view of a region of the matrix. Note that each row may contain zero's within the band β.

The sparse matrix vector multiply problem involves multiplying a sparse matrix *A* by a vector *b* generating a result vector *c*. What distinguishes this problem from a typical matrix vector multiply is the sparsity of the matrix. A typical sparse matrix may have dimensions of $100{,}000 \times 100{,}000$ yet, of the ten billion array elements, the sparse matrix will contain under a million non-zero entries. The band structure for a typical sparse matrix is shown in Figure 15. Because of the large dimension and sparsity, the full matrix is not stored. The computation complexity of the matrix-vector multiply is $O(\beta^2 N)$ if the width of the band is β. If *r* is the number of non-zeros in the row, $r < \beta$, then a good approximation is $O(r^2 N)$. Typical values for β range from 50 to 300, and for *r* from 9 to 30.

Sparse matrix-vector multiply is often used in iterative techniques for the solution of large sparse linear systems such as the Bi-Conjugate Gradient method (BCG). The BCG, for example, performs two sparse matrix-vector multiplies and several vector operations per iteration on both the operands and the results of the multiplies.

Here then is the problem. If the sparse matrix is an object that encapsulates its representation, and in particular, its distribution of data to processors, then the vector argument must be passed into the matrix multiply, distributed to the workers, the partial results returned to the master, formed into an answer, and communicated to the caller. This process is shown in Figure 16. Conceptually the equivalent of four copies of the vector must be transported. If the number of non-zeros is small then the overhead of transporting the data will dominate, and performance will suffer.

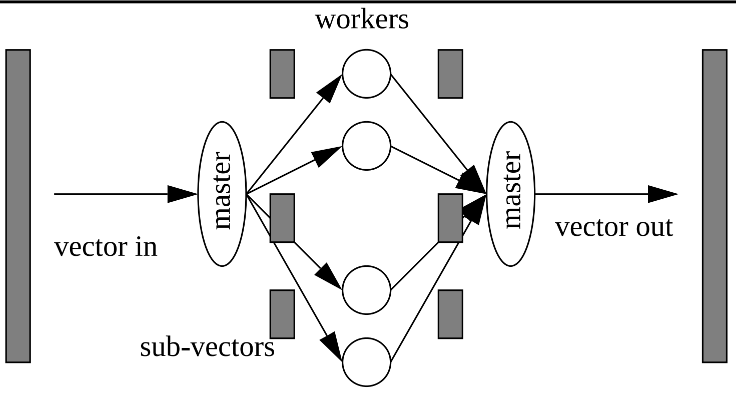


Figure 16—Encapsulation of the sparse matrix illustrating the communication required for a sparse matrix vector multiply.

This problem is solely the result of encapsulating the representation of the matrix *and* of distributing the matrix to different address spaces (processors). If the representation of the matrix into sub-matrices was not encapsulated then, rather than invoking the master to perform the multiply, the caller could invoke the workers separately, eliminating one pair of vector communications. Further, if the representation was exposed then algorithms, such as BCG, could place the vectors being multiplied on the same processor as the sub-matrices they are multiplied against, further reducing communication. This is what is done in hand-coded versions of BCG. A good question is, why is encapsulation not a problem in sequential object-oriented programs? The important difference is that in a sequential program communication is via memory: a reference to the vector is passed into the matrix-vector multiply, and references to the sub-vectors can be passed to the sub-matrices. Communication is very fast because no copies are required, and even if they were, communication is fast because it all occurs in a single address space.

Efficient object-oriented implementations of algorithms such as BCG are certainly possible. But to achieve good performance the objects that are manipulated must change. Rather than using a natural, high-level class structure of sparse matrices and vectors, it is necessary to recognize processor boundaries and expose the implementation by sub-matrices and sub-vectors. To further reduce inter-address space communication the vector class and sparse matrix class may be combined into a “sparse matrix and associated vectors class” that is essentially equivalent to the worker code in a fully hand-coded parallel implementation of the algorithm. At that point, one no longer has an object-oriented program, rather one has a program written in an object-oriented language.

The Mentat implementation of sparse matrix-vector multiply follows the pattern established in our earlier examples. The sparse matrix consists of a master and workers. The matrix is partitioned by row and distributed to the workers. Each worker is itself a non-square sparse matrix of lower dimension. The multiply is accomplished by breaking the argument vector into sub-vectors and performing a matrix-vector multiply on each of the workers as described above. The results are combined and returned to the caller.

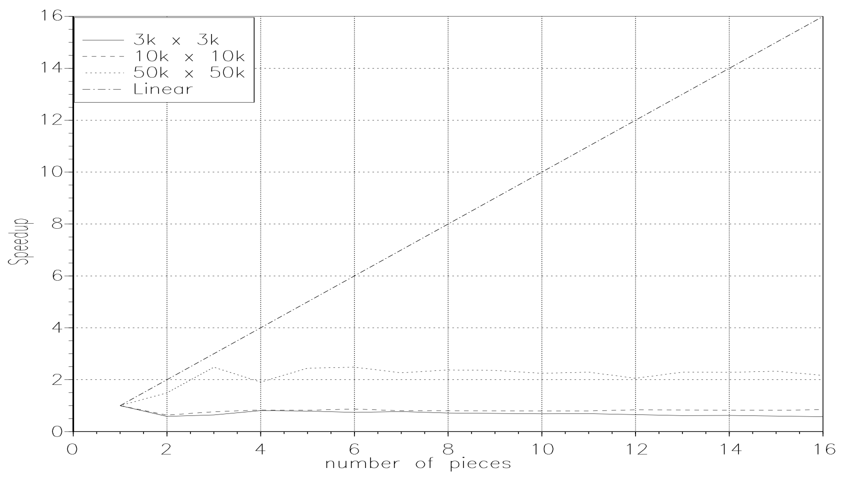
Overhead Analysis

Overhead in this application is dominated by the communication required to transport sub-vectors up and down the “call stack.” For a dimension 100,000 problem the total number of bytes transported is on the order 1.6 megabytes for single precision, 3.2 megabytes for double precision, and 6.4 megabytes for double precision complex numbers. The few hundred micro-seconds required to construct the program graphs is dominated by communication time. On the other hand, the use of our virtual machine model makes it very difficult to optimize away any communication. Even if the sub-vectors were stored on the processors as the corresponding sub-matrices we would require inter-process communication.

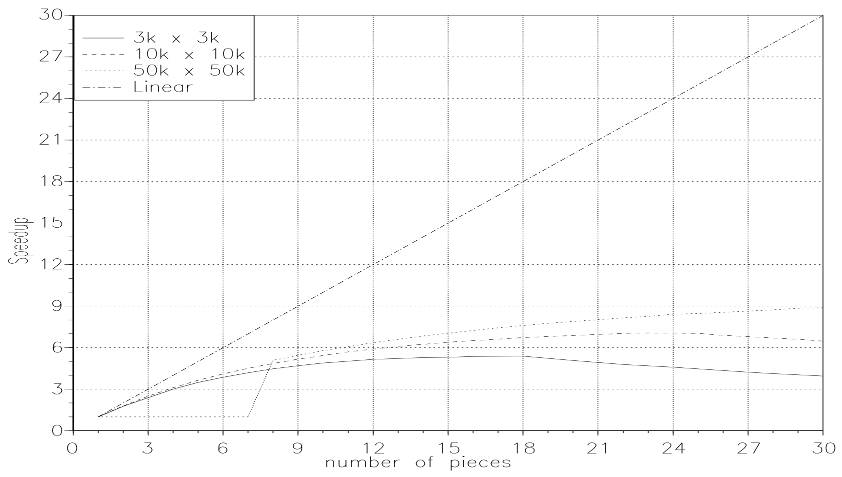
Performance

Performance is shown in Figure 17. One note on the collection of these numbers is in order before we begin. Unlike our earlier examples where I/O and data distribution costs were included in the execution time, we have not included them in this problem. Execution time is measured from after the matrix has been loaded into the workers. The reason for this is that the matrix is stored on disk in Harvel-Boeing format, a column major, ASCII data representation. Reading the data takes longer than performing the computation. All performance numbers were collected for double precision floating point matrices.

Caveats aside, the figures present results for the two platforms on varying problem sizes. To illustrate the effect of granularity, in Figure 18 we have varied the number of non-zeros per row for fixed problem dimension of 10,000 rows. Recall that for realistic problems the number of non-zeroes per row is between 9 and 30. At the lower end of that range the performance is quite limited for both architectures. We attribute this to the small computation granularity. At 20 non-zeroes per row the performance is better



(a) 16-processor Sparc IPC network. Nine non-zeroes per row. The 50K problem performs better because the message start-up cost is amortized over a larger computation.



(b) 32-processor Intel iPSC/2. Nine non-zeroes per row. We were unable to run the 50k problem on less than 8 processors. This is evident from the constant speedup of 1 and the sudden rise at 8processors.

Figure 17—Speedup for sparse matrix multiply

on the iPSC/2, but it is clear from the shape of the curve that granularity is still a problem and that more processors will not help. The results are unambiguous: even on a well-balanced machine such as the iPSC/2, encapsulation of the matrix implementation causes a significant increase in communication leading to poor performance.

Hand-coded implementations of this application expose the decomposition of the matrix of the operand vectors to the programmer. Hand-coded implementations enjoy nearly linear performance improvements due to the reduced communication. The difficulty with hand-coded implementations is that

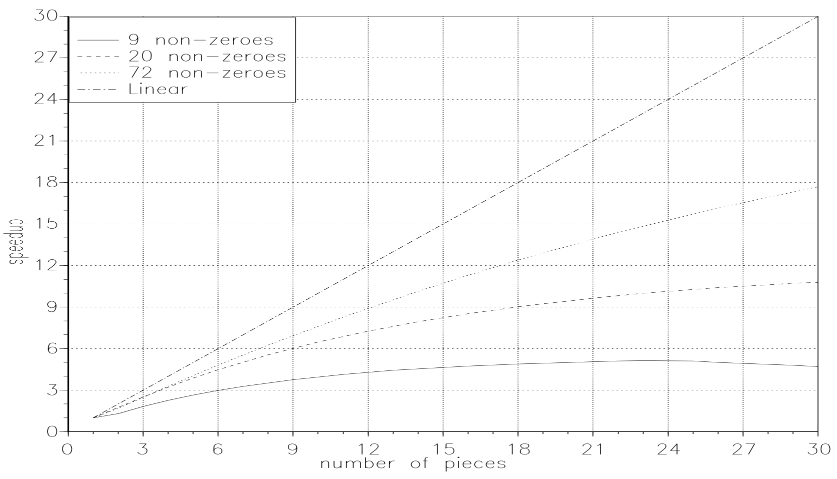


Figure 18—Sparse matrix multiply. The matrix dimension is fixed at 10Kx10K.

they are more complicated, and more importantly, they are less useful as software modules that can be plugged together with other modules. The hand-coded implementations, because they expose every detail are more difficult to integrate into a larger problem. Thus, the trade off between performance and the benefits of the object-oriented paradigm is greatest for this application.

3.7 Performance Summary

We set out at the beginning of this section to show that the use of the object-oriented paradigm, a virtual machine, and the dynamic management of parallelism does not necessarily lead to poor performance. We have demonstrated that it is possible to achieve good performance. However, good performance is not guaranteed. The four applications exhibit successively decreasing performance as the nature of the applications change from computation intensive to communication and synchronization intensive. This is not unexpected, In the next section we will see how the results presented compare with other, simillar systems.

4. Related Work

Work related to our effort falls into four categories, other object-oriented parallel processing systems, other compiler-based distributed memory systems, other portable parallel processing systems, and hand-coded implementations of the same applications. Both qualitative and quantitative comparisons can be made against members of all four sets when implementations exist. Two problems arise when making quantitative comparisons. First, it is rare for two different projects to implement and publish performance results that are comparable. The applications differ, the problem sizes differ, and the platforms upon which the results were gathered differ. In theory benchmark suites such as the Perfect club [11] and SLALOM [12] provide an even playing field on which to compare systems. Unfortunately there are no C++ benchmark standards of which we are aware. Another problem with benchmark suites are that they are usually toy problems. Toy problems do not always bring out the full richness of complexity that real problems possess. The issue of comparability has guided our choices of applications to include both image convolution and the solution of a system of linear equations using Gaussian elimination. Admittedly these two applications are toy, kernelized, problems that normally are a single component of a larger application. But they are widely used.

The second problem that arises when making quantitative comparisons are the many and varied ways that performance is measured. Some measure wall-clock time, others the CPU time of the workers (thus ignoring communication). When they start measuring also varies, some measure from when a program starts executing to when it completes, others begin counting after initialization and distribution of the data, ignoring I/O and initial data communication, while still others measure only selected kernels of the application. When speedups are used another problem arises, how are the sequential times determined?

With these caveats in mind we will attempt to provide meaningful comparisons to a few related systems. We will begin with other object-oriented parallel processing systems, followed by other compiler-based distributed memory system, then other portable systems, and finally a few hand-coded results.

4.1 Parallel Object-Oriented Systems

In the object-oriented parallel processing domain Mentat differs from systems such as [13][14] (shared memory C++ systems) in its ability to easily support both shared memory MIMD and distributed memory MIMD architectures, as well as hybrids. PC++ [15] and Paragon [16] on the other hand are data-parallel derivatives of C++. Mentat accommodates both functional and data-parallelism, often within the same program. ESP [17] is perhaps the most similar of the parallel object-oriented systems. It too is a high-performance extension to C++ that supports both functional and data-parallelism. What distinguishes

Mentat is our compiler support. In ESP remote invocations either return values or futures. If a value is returned, then a blocking RPC is performed. If a future is returned, it must be treated differently. Futures may not be passed to other remote invocations, limiting the amount of parallelism. Finally, ESP supports only fixed size arguments (except strings). This makes the construction of general purpose library classes, e.g., matrix operators, difficult.

Performance is difficult to compare. The performance of PRESTO [14] is very good, much better than Mentat for Gaussian elimination. However, the results are for the Sequent, a shared memory processor for which communication and synchronization are very cheap. They do not extend to distributed memory machines.

4.2 Compiled Distributed Memory Systems

Until recently, there were few results for compiled, as opposed to hand-coded applications on distributed memory machines. There are now several active projects in this area, Fortran-D [20], Dataparallel C [21], Paragon [16], and the inspector/executor [22] model to name a few. The underlying characteristics of each of the above is that they are primarily data-parallel languages. What differentiates our work from theirs is that Mentat exploits opportunities for both functional and data-parallelism. Further, in Mentat, parallelism, and the communication and synchronization constructs that are required, is dynamically detected and managed. Most other systems statically determine the communication and synchronization requirements of the program at compile-time.

Performance results for the solution of a system of linear equations on the Intel iPSC/2 are available for both Fortran D [20] and Dataparallel C [23][5]. Table 1 compares the three implementations. It

Table 1: Comparison of iPSC/2 implementations Gaussian elimination

System	Number of PE’s	Problem Size	Sequential Time (sec)	Speedup
Mentat	8	250×250	74.2	7.24
	16	250×250	74.2	6.13
	8	512×512	652.6	8.7[a]
	16	512×512	652.6	13.1
Fortran D	8	256×256	73.4	5.32
	16	256×256	73.4	9.51
Dataparallel C	32	256×256	67.6	12.4
	32	512×512	583.2	20.2
Linda[b]	16	400×400	NA	13
	32	400×400	NA	18

a. The apparent superlinear speedup is due to the sequential code being executed on the host running Unix. This introduces overhead not present on the nodes.

b. The Linda times were interpreted from a graph on page 10 of [24] using a straight-edge.

is unclear from [20] exactly what is timed, i.e., whether data distribution time is included. The Dataparallel C implementation uses Gauss-Jordan to solve the system of equations. As can be seen, the three systems offer comparable performance for relatively small problems.

Performance results for the solution of a system of linear equations on a network of workstations are available for Dataparallel C and Linda. Once again, the Dataparallel C implementation uses Gauss-

Table 2: Performance of network implementations of Gaussian elimination

System	Number of PE’s	Dimension	Sequential Time	Speedup
Mentat	4	1024×1024	649.7	3.5
	8	1024×1024	649.7	6.0
Dataparallel C	4	1000×1000	397.5	2.8
	8	1000×1000	397.5	2.2
Linda[a]	4	800×800	NA	3.5
	8	800×800	NA	5.5

a. Once again, the numbers are approximations from a graph.

5. The results are for the version of C* that later became Dataparallel C.

Jordan, although they compare to a sequential Gaussian Elimination implementation. The sequential time of the Dataparallel C implementation is on a SPARC2 workstation. The Linda implementation is LU factorization only. We are uncertain how the Linda speedups were computed, or what type of workstation was used.

4.3 Portable Systems

Applications portability across parallel architectures is an objective of many projects. Examples include PVM [25], Linda [26][24], the Argonne P4 macros [27], and Fortran D [19]. Our effort shares with these and other projects the basic idea of providing a portable virtual machine to the programmer. The primary difference is the level of the abstraction. Low-level abstractions such as in [25][26][27] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [16][20][21] share our philosophy of providing a higher level language interface in order to simplify applications development.

What differentiates our work from other high-level portable systems is that we support both functional and data-parallelism as well as support the object-oriented paradigm. We have already presented a performance comparison with Fortran D, Linda, and Dataparallel C. We have seen no comparable performance numbers for PVM, although we are sure that the applications exist. Our expectation is that performance is comparable.

4.4 Hand-coded Results

Most production parallel applications to date have been hand-coded using C or FORTRAN extended with low-level primitives, such as send and receive. Since all high-level systems such as Mentat, FORTRAN D, and Dataparallel C, are ultimately implemented using these primitives, it will always be possible for a hand-coded version of an application to do at least as well as the compiled version. The human need only to “write” the same code the compiler writes; the converse, that a compiler can do as well as a human, is not always true. A similar contest exists between high-level sequential languages and human assembly language programmers. The issues are almost identical. The really important questions are: What is the performance penalty of compiler generated code versus hand code? What is the productivity penalty of hand code versus compiler generated code? We will not take up this debate here, but it is clear that comparing compiler generated code against the best hand written code that is humanly possible sets too high a standard.

Table 3 shows the Mentat performance of Gaussian Elimination and scanlib compared to results from hand-coded versions given in [20] and [7]. The Mentat version compares well with the hand-coded versions.

Table 3: Mentat versus hand-coded on the iPSC/2

Problem	Number of PE’s	Hand-coded Speedup or execution time	Mentat Speedup or execution time
Gauss 256×256	8	5.3 speedup	7.2 speedup
Gauss 256×256	16	9.7 speedup	6.1 speedup
Scanlib - SW	7	41.0 minutes	41.2 minutes
Scanlib - SW	15	19.2 minutes	19.4 minutes

5. Summary and Future Work

Overhead is the friction of parallel processing. The dynamic detection and management of parallelism, the use of the object-oriented paradigm, and the use of a virtual machine all contribute to overhead in Mentat. It has been argued that including features such as these in a parallel processing environment would negate performance gains. In this paper we have presented evidence that this is not always true. In particular, we have shown that for a range of applications very good performance is possible even in the presence of overhead. However, the news is not all good. There are applications where the use of the object-oriented paradigm significantly impacts the performance because of its encapsulation properties. In particular we have found that applications perform poorly when encapsulation increases the amount of data communication (by hiding the underlying data distribution) and drives computation granularity down, reducing the gains from parallelism.

The vehicle for our discussion is Mentat, a portable object-oriented parallel processing system developed at the University of Virginia. We presented Mentat, its key features, and the Mentat philosophy: to recognize and exploit the comparative advantages of both compilers and humans. We then examined the performance of four applications under Mentat on two different platforms. The four applications were chosen because they span the spectrum of communication and synchronization requirements, because they each are representative of a class of applications, and in one case because it highlights the problems with object-oriented programming on a distributed memory machine.

The results indicate that even with the overhead, the Mentat approach is viable for a wide range of applications. Our current and future work falls into two areas: collecting quantitative performance information and qualitative ease-of-use information on the usefulness of Mentat on real applications, and extending Mentat into a heterogeneous meta-system environment.

We are actively involved with collaborators in a spectrum of application domains, including Biochemistry, Physics, Chemical Engineering, Electrical Engineering, Radio Astronomy, and Computer Science. Concurrently we are both porting Mentat to new platforms and experimenting with a heterogeneous Mentat testbed. The meta-systems environment presents new challenges in scheduling, problem partitioning, and data coercion that we will address using the Mentat philosophy of carefully dividing the responsibility for managing the environment between the human programmer, and the compiler and run-time system.

6. References

[1] A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," to appear in *IEEE Computer*, May, 1993.

[2] C. Polychronopoulos, *Parallel Programming and Compilers,Kluwer* Academic Publishers, 1988.

[3] A. S. Grimshaw and V. E. Vivas, "FALCON: A Distributed Scheduler for MIMD Architectures", *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 149-163, Atlanta, GA, March, 1991.

[4] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *J. Mol. Biol.*, 147, pp. 195-197, 1981.

[5] W. R. Pearson and D. Lipman, "Improved tools for biological sequence analysis", *Proc. Natl. Acad. Sci. USA*, 85, pp. 2444-2448, 1988.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, "Basic local alignment search tool", *J. Mol. Biol.*, 215, pp. 403-410, 1990.

[7] A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 57-66, Syracuse, NY, Sept., 1992.

[8] M. J. Quinn, *Designing Efficient Algorithms For Parallel Computers*, McGraw-Hill Book Company, New York, 1987.

[9] P. Lemkin, *Personal Communication*, April, 1992.

[10] A.S. Grimshaw, J.B.Weissman, and E.A. West, “Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” *Submitted to Journal of Parallel and Distributed Computing.*

[11] L. Pointer, (ed). Perfect Report 1, Center for Supercomputing Research & Development, Univ. Illinois, Champaign-Urbana, IL. July 1989.

[12] J. Gustafson, D. Rover, S. ELbert, and M. Carter, “SLALOM: The First Scalable Supercomputing Benchmark,” Ames Laboratory, Ames, IA 50011-3020.

[13] B. Beck, “Shared Memory Parallel Programming in C++,” *IEEE Software*, 7(4) pp. 38-48, July, 1990.

[14] B. N. Bershad, E. D. Lazowska, and H. M. Levy, “Presto: A System for Object-Oriented Parallel Programming,” *Software - Practice and Experience*, 18(8), pp. 713-732, August, 1988.

[15] J. K. Lee and D. Gannon, “Object Oriented Parallel Programming Experiments and Results,” *Proceedings of Supercomputing ‘91*, pp. 273-282, Albuquerque, NM, 1991.

[16] A.L.Cheung, and A.P. Reeves, “High Performance Computing on a Cluster of Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 152-160, Syracuse, NY, Sept., 1992.

[17] S. K. Smith, et al., “Experimental Systems Project at MCC,” MCC Technical Report Number: ACA-ESP-089-89, March 2, 1989.

[18] G. C. Fox, et al., “Fortran D Language Specifications,” Technical Report SCCS 42c, NPAC, Syracuse University, Syracuse, NY.

[19] D. Callahan and K. Kennedy,‘‘Compiling Programs for Distributed-Memory Multiprocessors’’ *The Journal of Supercomputing*, no. 2, pp. 151-169, 1988, Kluwer Academic Publishers.

[20] Min-You Wu, and G.C. Fox, “A Test Suite Approach for Fortran90D Compilers on MIMD Distributed Memory Parallel Computers,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 393-400, Syracuse, NY, Sept., 1992.

[21] N. Nedeljkovic, and M.J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 28-36, Syracuse, NY, Sept., 1992.

[22] J. Saltz, H. Berryman, and J. Wu, “Multiprocessors and Runtime Compilation,” ICASE Report No. 90-59, September 1990.

[23] P.J. Hatcher, “A Production-Quality C* Compiler for Hypercube Multicomputers,” *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Williamsburg, VA, April 21-24, 1991.

[24] R. Bjornson, et al., “Experience with Linda,” Yale University, YALEU/DCS/TR-866, August 1991.

[25] V.S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

[26] N. Carriero and D. Gelernter, ‘‘Linda in Context,’’ *Comm. of the ACM*, pp. 444-458, April, 1989.

[27] J. Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.