Portability and Performance:
Mentat Applications on Diverse Architectures

Padmini Narayan,
Sherry Smoot,
Ambar Sarkar,
Emily West,
Andrew Grimshaw,
Timothy Strayer
Technical Report No. CS-92-22

This work was supported in part by grants from the National Science Foundation, CDA-8922545-01, the National Aeronautics and Space Administration, NAG-1-1181, and the National Laboratory of Medicine LM04969.

Portability and Performance: Mentat Applications on Diverse Architectures

Padmini Narayan
Sherry Smoot
Ambar Sarkar
Emily West
Andrew Grimshaw
Timothy Strayer

University of Virginia
Charlottesville, Virginia
pn8f@virginia.edu
grimshaw@virginia.edu
July 22, 1992

Abstract

Parallel programs are complex, especially when the communication, synchronization, and scheduling issues must be managed by the programmer. Considerable amounts of time and effort are put into developing a parallel program for a particular application, yet once written for one architecture, it may be prohibitively difficult to port the code to another architecture. Mentat is an object-oriented parallel processing system designed for the development of parallel programs where the underlying architecture is an abstraction. Mentat masks the complex aspects of communication, synchronization, and scheduling from the programmer. This facilitates both ease of use and portability.

Here we present several applications that have been implemented within the Mentat environment running on three different platforms. For each application we describe the major aspects of its implementation within Mentat, and present performance comparisons of the Mentat implementation against a serial implementation of the application.

1. Introduction

Two problems plague programming on MIMD architectures. First, writing parallel programs by hand is very difficult since the programmer must manage communication, synchronization, and scheduling of potentially thousands of independent processes. The burden of *correctly* managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting code is usually not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. Consequently, porting the application to a new architecture requires considerable effort. Given the plethora of new architectures and the rapid obsolescence of existing ones, communities are reluctant to commit to a single platform for their applications.

Mentat [1] has been developed to directly address the difficulty of developing architecture-independent programs. The three primary design objectives of Mentat are to provide easy-to-use parallelism, to achieve high performance via parallel execution, and to facilitate the portability of applications across a wide range of platforms. Mentat is based on the premise that writing programs for parallel machines does not have to be hard; rather, it is the lack of appropriate abstractions that has kept parallel architectures difficult to program and, hence, inaccessible to mainstream, production system programmers.

The Mentat approach exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling, from the programmer [2, 3]. Instead of managing these details, the programmer concentrates on the

application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. The parallelization of these complex tasks is handled by Mentat.

In this report we present a set of diverse applications. We show that these applications have straight-forward implementations within Mentat, and that these implementations are easily ported to various MIMD platforms. In particular, we have developed and ported these applications to two loosely-coupled networks of Sun workstations (one of only Sun 3/60s, and one of only Sparc IPCs), and to a tightly-coupled Intel iPSC/2 hypercube. Among the applications presented here are:

- An in-core matrix multiplication implementation
- A Gaussian elimination method of solving linear systems
- An image processing application
- A DNA sequence decomposition application
- A process pipeline example

Ease of implementation and portability are important aspects of the Mentat approach [2], but these benefits are moot if there is no performance advantage as well. Since Mentat provides a framework within which the programmer and compiler cooperate to parallelize applications, performance gains through parallel execution are realized. We use the applications listed above to demonstrate the relative speed-up from a well-optimized serial implementation to an implementation within Mentat.

Applications other than the ones presented here are also being developed within Mentat, and experience with these applications will appear in subsequent reports. Also, we are constantly increasing the number of platforms on which Mentat applications can be run. In addition to the architectures described here, Mentat is also supported on the Silicon Graphics' Iris and the Intel iPSC/860 and

ports are underway for the Intel Paragon, the IBM RS/6000 and the TMC-CM5 machines.

2. Test Environment

We conducted experiments on three different platforms, a network of Sun 3/60s, a network of Sparc IPCs, and an Intel iPSC/2 hypercube. These three platforms are distinguished by their processor computational capabilities and their interconnection network characteristics. The network of Sun 3/60s and the network of Sparc IPCs differ by processor but not by interconnect. The Sparcs and the Intel iPSC/2 have processors of similar capacity but the interconnect is different.

One platform is a local area network of eight Sun 3/60 workstations, each with 8 Mbytes of memory and a math coprocessor, and served by a Sun 3/280 file server. Another is a local area network of eight Sparc IPCs, each with 16 Mbytes of memory, served by a Sun 4/260 file server. Both of these networks use Ethernet as their interconnection medium.

The third platform is an Intel iPSC/2 hypercube configured with thirty-two 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 for its 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. Performance of Primitive Operations

The performance of Mentat, and of the Mentat approach, hinges on the speed with which primitive operations can be performed. In particular, the

Function	Time		
	Sun 3/60	Sparc IPC	Intel iPSC/2
Message Latency (one way)	5.6 ms	2.8 ms	0.8 ms
Single Null RPC	14.3 ms	6.9 ms	3.9 ms
Mentat Overhead	3.1 ms	1.9 ms	2.3 ms
Double Null RPC	20.1 ms	9.2 ms	5.3 ms

Table 1. Communication and Mentat Overhead for Various platforms

communication latencies, the time to detect data dependencies, and the time required to construct program graphs all contribute to the overhead. This overhead must be reduced so that parallelism can provide gains in performance. Table 1 shows the communication and Mentat processing overheads incurred for a Null RPC.[1] The one-way message latencies for the Sun 3/60s and the Sparc IPCs are due to the characteristics of UDP and Ethernet, and also include the host operating system scheduling and task switch overhead. This latency is of course signifi cantly reduced for the Intel iPSC/2 since the interconnection network is so tightly coupled. The Mentat overhead is computed by subtracting twice the message transport cost (once for request and once for reply) from the Single Null RPC time.

[1.] A Null RPC is a remote procedure call with one integer argument and one integer result, but no computation is performed in the body. Hence, the service time is zero, and only the communication and Mentat overheads are observed. This serves as a baseline measurement.

```
timer.start();
for (i = 0; i < iterations;
i++)
 {
   int j, k,l;
   j = node1.one_arg(0);
   k = j+1;
 }
timer.stop();
```

Single Null RPC Code

```
timer.start();
for (i = 0; i < iterations;
i++)
 {
   int j, k,l;
   j = node1.one_arg(0);
   l = node2.one_arg(0);
   k = j+1;
   k = l+1;
 }
timer.stop();
```

Double Null RPC Code

Figure 1. RPC Timing Code

Table 1 also shows the effect of overlapping communication with processing overhead when two Null RPCs are performed back-to-back. The code segments for a single Null RPC and a double Null RPC are given in Figure 1. The only requirement of an RPC is that the result of the RPC be available when used in some future code segment. In the double Null RPC case, the communication involved when *node2.one_arg()* is called, is overlapped with the computation of *k*.

4. Applications

In this section we briefly describe each of the five applications listed earlier. For each of these applications, we give the class definitions and pseudo-code for the implementation. The class definitions show how the applications were decomposed. The pseudo-code shows the implementation of the class member functions in more detail. Once implemented, these applications were run on the various platforms, and execution time measurements were made. In each case the performance of the Mentat version of the application is compared with the performance of an equivalent sequential C++ implementation. Rather than constraining the Mentat implementation to run on a single processor, we used a

C++ implementation for our comparisons, because the latter does not incur the communication overhead inherent with any parallel implementation. 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.

Unless otherwise specified, all times are carriage return to complete times, i.e., all overhead including loading of object executables and data distribution have been included. The best times and not averages are used to calculate the speed-ups for each application and architecture. This is done to demonstrate the capabilities of Mentat. Some of the irregularities seen in the graphs are due to the Random algorithm used by the scheduler [4]. We observe that the scheduler 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.

4.1. In-core Matrix Multiplication

This matrix multiplication utility operates on any two matrices already in memory. For the multiplication of matrices A and B, the user of this utility specifies how to partition the problem so that pieces of the matrices can be distributed among the various processors. If the user requests that the problem is to be divided into k pieces, then the B matrix is split into floor(sqrt(k)) vertical slices, and the A matrix is split into floor(k/floor(sqrt(k))) horizontal slices. The actual number of worker processes created, w, is floor(sqrt(k)) $\times$ floor(k/floor(sqrt(k))), and is less than or equal to k. Each of the workers gets the appropriate piece of A and B to multiply. The results of the workers are merged together and the final result is made available to the user of this utility.

4.1.1. Class Definitions

The two Mentat class definitions that are used to implement this utility are:

```
regular mentat class matrix_class {
public:
   DD_floatarray* mult_mat(DD_floatarray* mat1,
                           DD_floatarray* mat2,
                           int pieces);
}

persistent mentat class work_class {
public:
   DD_floatarray* mult_work(DD_floatarray* mat1,
                            DD_floatarray* mat2);
}
```

The *matrix_class* is the primary class; it creates the workers which are instances of *work_class*. Multiplication member functions are defined for both classes. The member function *matrix_class.mult_mat()* is invoked by the user which in turn invokes *work_class.mult_work()* to perform the actual multiplication. A diagrammatic representation of this utility is shown in Figure 2.

The pseudo-code for the member functions of the two classes are:

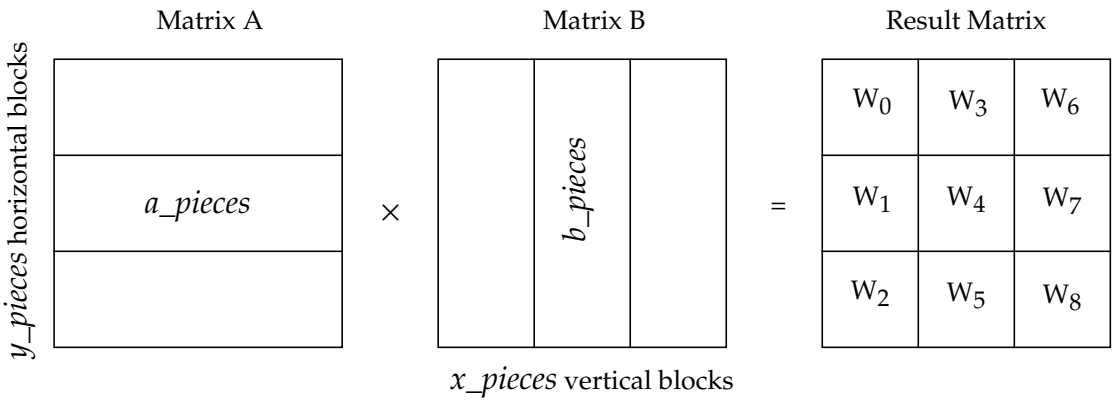


Figure 2. Two-Matrix Multiplication

`matrix_class::mult_mat():`

1. Create *w* workers
2. Let *x_pieces* = fl oor(sqrt(*k*))
   Let *y_pieces* = fl oor(*k*/fl oor(sqrt(*k*)))
3. For each *x_pieces* do
   Give each worker a vertical piece of B, *b_pieces*
   For each *y_pieces* do
   Give each worker the correct horizontal piece from A, *a_pieces*
   Perform multiplication in each worker.
4. Reassemble all parts of the workers' results and return resulting matrix

`work_class::mult_work():`

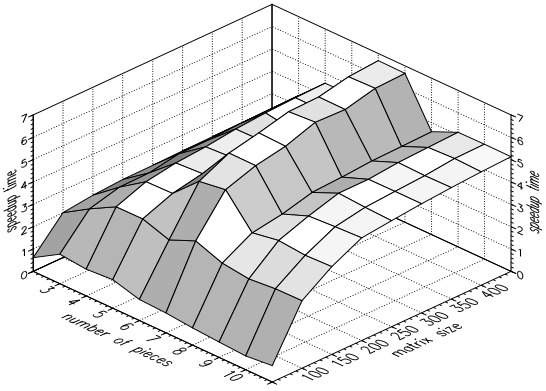
1. Perform the multiplication:
   Let *result* = *a_pieces* × *b_pieces*
2. Return result to manager process

4.1.2. Communication and Computation Complexity:

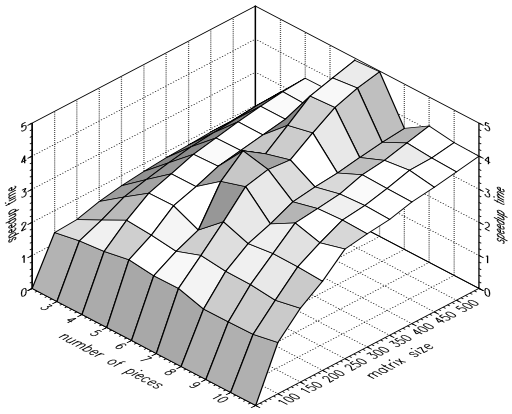
Since each worker receives one block of each of the two input matrices, and returns the resultant submatrix to the controlling object, there are a total of three messages to or from each worker. Hence the total number of messages generated during the execution of this problem is 3*w*. The amount of data that is actually moved around is (*x_pieces*+*y_pieces*+1) × *n*. The computation complexity is O(n^3).

4.1.3. Performance

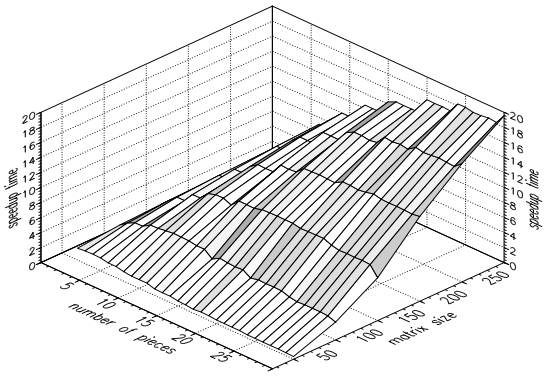
Figure 3. shows the performance improvements from a serial implementation to the parallel Mentat implementation, of a matrix multiplication for each of the platforms in our study. These speed-ups are given with respect to matrix size and the number of pieces into which the matrix is partitioned. It is interesting to note the effect that the number of pieces has on the speed-up. In the graphs for Sun3/60s and Sparcs, partitioning of the problem into six pieces provided the best performance, even though there were eight processors available on each of the networks. This is because both the main program and the manager



(a) 8-processor Sun 3/60 network



(b) 8-processor Sparc IPC network



(c) 32-processor Intel iPSC/2

Figure 3. speed-up for matrix multiply.

object *matrix_class* required a processor each, leaving six processors for the workers. If the problem is partitioned into a number of pieces greater than six, some objects are made to share a processor. On the Intel iPSC/2, where there are 32 nodes available, no two objects are scheduled on the same processor, and so this effect is not observed.

Since the problem is actually partitioned into w workers instead of the requested k, there will be some values for the number of workers requested that will have the same performance as for fewer number of workers requested. For example, if five workers are requested, then w is four, and the performance for both $k = 4$ and $k = 5$ will be the same. This happens for $k = 6$ and $k = 7$ as well.

The speed-up on the Sparcs is not as good as that on the Sun3's for small size problems as the granularity of these problems is very fine. The Sparcs are approximately ten times faster than the Sun3's, making the amount of computation smaller for the same amount of communication.

4.2. Gaussian Elimination with Partial Pivoting[2]

Many algorithms have been developed to solve systems of linear equations by operating on matrix systems. The algorithm we have implemented here is the Gaussian Elimination method with partial pivoting. This algorithm primarily consists of two phases, a forward elimination phase and a back substitution phase.

4.2.1. Class Definitions

The following are the two Mentat class definitions that are used to implement this algorithm.

```
regular mentat class matrix_ops{
public:
   DD_floatarray* solve(DD_floatarray* mat,
          DD_floatarray* vector,
          int pieces);
}
```

[2] Other iterative methods like the Choleski method are available to solve systems of linear equations. The Choleski method has inherently fewer synchronization points than Gaussian Elimination and may result in better speed-ups.

```
persistent mentat class sblock{
public:
   int initialize(DD_floatarray* the_block);
   DD_floatarray* get_max(int for_col);
   DD_floatarray* reduce(DD_floatarray* by_row);
}
```

An instance of the *matrix_ops* class, which is the controlling object, partitions the matrix, into *k* strips and distributes each strip into an instance of an *sblock*. Then, for each row, the reduce operator is called for each *sblock* using the partial pivot calculated at the end of the previous iteration. The reduce operation of the *sblock* reduces the *sblock* by the vector, selects a new candidate partial pivot, and forwards the candidate row to the controlling object for use in the next iteration. Once this is done for each row, back substitution is performed by the *matrix_ops* object. This algorithm results in frequent communication and synchronization. The pseudo-code shown outlines the implementation.

matrix_ops::solve():

1. In the controlling function, create *k* workers, where k is the number of blocks into which the matrix is divided.
2. Initialize workers with the appropriate block of the matrix.
3. For each worker, get candidate pivot row.
4. For each column:
   Get row with largest absolute value in the first column from among the candidate pivot rows. This is the new pivot row.
   Distribute pivot row to all workers.
   Make workers "reduce" submatrix and return next candidate pivot.
5. Go to last row and do back substitution to get result.

sblock::reduce():

1. Find candidate pivot by reducing appropriate column in submatrix.
2. Find row with largest entry in the same column.
3. Reduce that row.
4. Return that row to *matrix_ops*.
5. Reduce rest of the rows in the submatrix.[3]

4.2.2. Communication and Computation Complexity:

Let k be the number of workers and m be the matrix size. Then the communication complexity is of the order of $O(3km^2)$. This is because for each column of the matrix, we initialize k workers, do the reduction on each of them and get back the result from each of them.

4.2.3. Performance

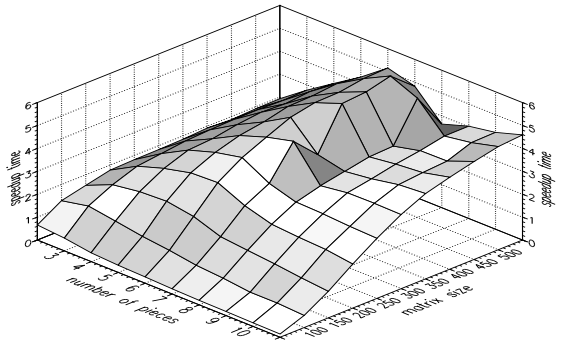
The effect of frequent synchronization in the algorithm can be clearly seen when the speed-up results for Gaussian Elimination in Figure 4. are compared to the results for matrix multiplication. The speed-up obtained is lower for Gaussian Elimination. As expected, speed-up on the Sparcs is not as good as that on the Sun3s for small problems.

4.3. Image Convolution

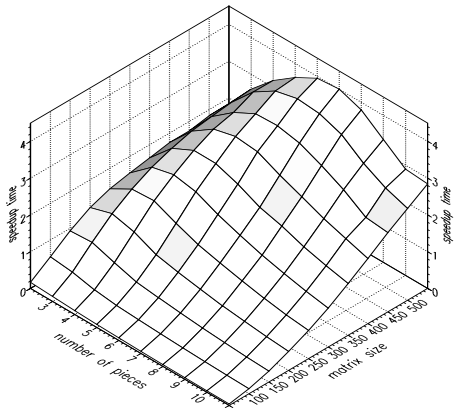
Image convolution is a common technique used in image processing. It is an instance of a class of algorithms called *stencil* algorithms, other examples of which include the *Jacobi iterative method*. Convolution is used to filter out distortion from images and obtain progressively better images. It is a computationally intense application and has relatively few synchronization points.

---

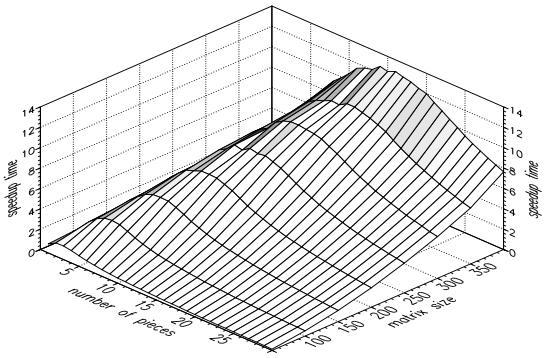
[3] Note: Steps 4 and 5 permit overlapping much of the communication with computation.



(a) 8-processor Sun 3/60 network



(b) 13-processor Sparc IPC network



(c) 32-processor Intel iPSC/2

Figure 4. speed-up for Gaussian elimination.

The input image can be regarded as an $p \times q$ matrix, f, of pixels. Let us consider a fi lter, h, of size $m \times n$. Then, according to the convolution algorithm, each pixel in the resultant matrix (image), g, is calculated as follows:

$$g[i,j] = \sum_{m=-\frac{p}{2}}^{\frac{p}{2}} \sum_{n=-\frac{q}{2}}^{\frac{q}{2}} \frac{f[i+m, j+n] \times h[m,n]}{m \times n} \quad \text{(Eq 1)}$$

4.3.1. Class Definitions

Given below are the two Mentat class defi nitions that have been used to implement this algorithm.

```
persistent mentat class Convolver{
public:
   void initialize();
   void setSource(string *file_name);
   int convolve();
}

persistent mentat class convolver{
public:
   int convolve(DD_chararray* filter);
   void set_top(DD_chararray *top);
   void set_bottom(DD_chararray* bottom);
   DD_chararray get_top();
   DD_chararray get_bottom();
}
```

An instance of the *Convolver* class is the controlling object in this implementation. It creates *k* instances of the *convolver* class and distributes appropriate parts of the input fi le to each*convolver* object. Boundary information is exchanged between neighboring convolver objects, as shown in Figure 5., using the *convolver.get_bottom()* and *convolver.get_top()* member functions. The actual *convolver.convolve()* function is performed in each *convolver* object and results are returned to the *Convolver* object. The pseudo-code for this implementation is outlined below:

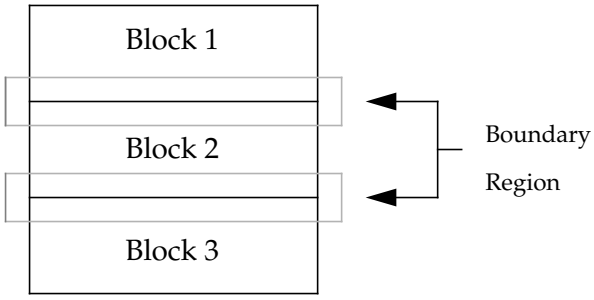


Figure 5. Boundary exchange between convolver objects.

`Convolver::convolve()`:

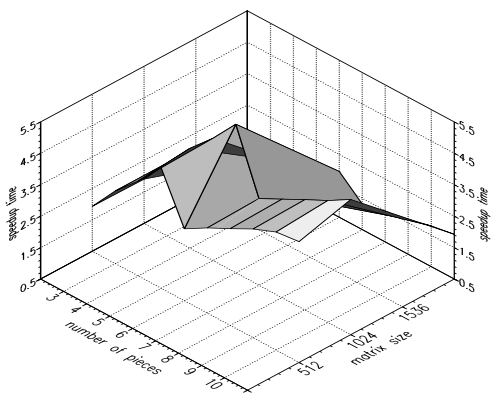
1. In the controlling object, create *k* workers, where k is the number of horizontal blocks the source fi le is divided into.
2. Distribute a block to each worker.
3. For each fi lter do:
   Tell workers to exchange boundaries with neighbors:
   i.e. for each worker, i do:
   convolver[i].set_top(worker[i-1].get_bottom())
   convolver[i].set_bottom(worker[i+1].get_top())
   Tell workers to convolve.
4. Tell workers to store results in target fi le.

4.3.2. Communication and Computation Complexities

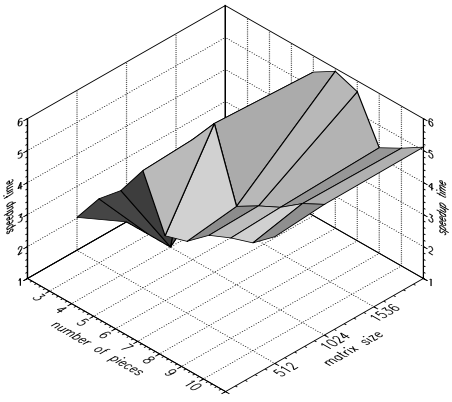
Let us consider a source image fi le, which is basically an n× n matrix that needs to be convolved, and a sequence of fi lters of dimension fs_i, 1 < i< m. If *k* is the number of pieces to divide the input image fi le into (number of *convolver* objects), communication complexity is given by:

$$\sum_{1 < i < m} \left\lfloor \frac{fs_i}{2} \right\rfloor n\,(k-1)\,2 \qquad \text{(Eq 2)}$$

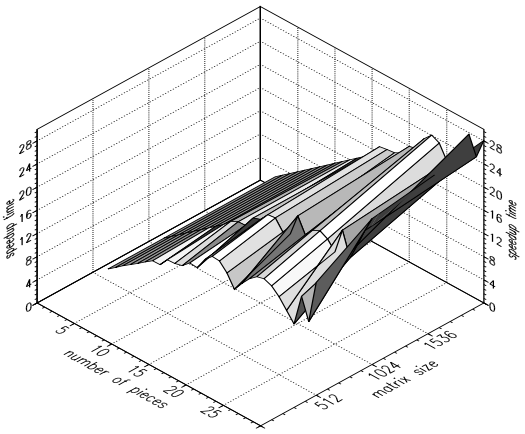
If fs_i is a constant, the communication complexity is solely dependent on the number of workers for a fi xed size matrix. The computation complexity is given by:



(a) 8-processor Sun 3/60 network



(b) 8-processor Sparc IPC network



(c) 32-processor Intel iPSC/2

Figure 6. speed-up for Convolver.

$$\frac{1}{k} n^2 \sum_{1 < i < m} fs_i^2 \qquad \text{(Eq 3)}$$

4.3.3. Performance

All the speed-ups for this application shown in Figure 6. are for two fi lters of size 9. This means that for the image convolution, the fi lter used was a 9× 9

matrix and two such filters were used (i.e., step 3 of the Convolver pseudo-code was executed twice before getting the performance numbers). For the Sun3s, the speed-up falls rapidly for a 2Kbytes image size because of the fact that these machines have only 8Mbytes of memory and the entire problem does not fit into memory. This results in frequent page faults, which in turn leads to poor performance. On the Sparcs and on the Intel iPSC/2, we see better speed-ups for large size problems. On the Intel iPSC/2, the speed-up is close to linear for the 2K $\times$ 2K image. But once again, for the Sparcs the speed-up decreases rapidly if the input file is partitioned into greater than six pieces. This is again due to the poor decisions made by the scheduler at high load.

4.4. DNA Sequence Comparison

With the advances in DNA cloning and sequencing technologies, biologists today can determine the sequence of a protein more easily than they can determine its structure. The current technique used for determining the structure of new proteins is to compare their DNA sequences with those of known proteins. DNA and protein sequence comparison involves comparing a single query sequence (for the unknown protein) against a library of sequences to determine the evolutionary history of the query sequence. In addition, this comparison can identify related sequences in the library. Since the DNA and protein sequences are represented as strings of characters, the problem reduces to one of pattern matching. Several algorithms, including Smith-Waterman, FASTA, and Profile, are used to compare the query sequence against the library. Among these algorithms, the Smith-Waterman algorithm is the most rigorous and time-consuming. FASTA, which is based on heuristics, is about 20 to 100 times faster. Apart from speed, these methods vary in their accuracy as well. As each comparison between entries in the

sequence library and the query sequence is completely independent, these algorithms are ideally suited for parallelization.

4.4.1. Class Definitions

Given below is the Mentat class defi ned for this application.

```
regular mentat class worker{
public:
   result_list* dowork(sequence, libinfo, pstruct);
}
```

This application requires the defi nition of a single Mentat class for its implementation. The input fi le is partitioned into*k* blocks and distributed to *k workers*. The query sequence is also passed to the *workers*, who then perform the comparison by invoking either the Smith-Waterman, FASTA or Profi le algorithms.The pseudo-code for this implementation is outlined.

`main():`

1. In the controlling object get query sequence from input fi le
2. Determine size of library fi le and parcel it into *k* blocks.
3. Fire each of *k workers* by distributing block information along with the query sequence.
4. For *k* results
   compute statistics.
5. Generate output scores and statistics.

`worker::dowork():`

1. Open library fi le.
2. Initialize comparison sequence.
3. For each comparison sequence in library parcel,
   Get sequence from library fi le
   Compute similarity score to query sequence.(scoring technique used is either Smith-Waterman, FASTA or Profi le)
   Append score to result list
4. Return result list.
5. Close library fi le.

4.4.2. Communication and Computation Complexities

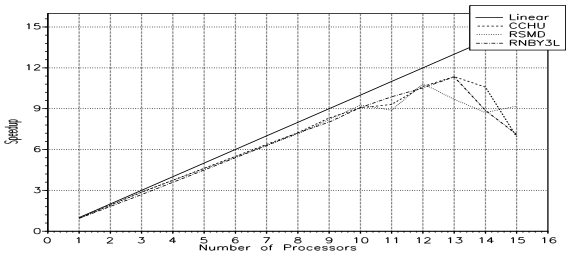
The computation complexity of the comparison algorithms is O(nm) where n is the length of the query sequence and m is the length of the longest library sequence. The communication complexity is small for these algorithms. It includes transporting a copy of the query sequence for each worker which in turn send a result structure back to the manager program. The length of a typical query sequence is 25 - 3500 bytes and the total number of results generated is equal to the length of the library in sequences, where each result is at most 5 -10 bytes. A typical library size is in the range of ten to twenty thousand sequences.

4.4.3. Performance

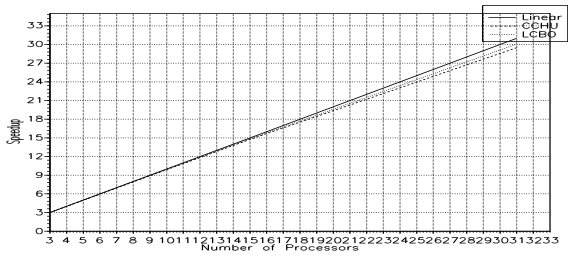
The performance speed-ups for this application and observations on them are given in [5]. As can be seen in Figure 7., the speed-up is almost linear for Smith-Waterman on both architectures. FASTA on the iPSC/2 and Sparcs suffers from very small computation granularity. This is quite apparent as speed-up ceases to improve with additional processors. The irregularities of the curves for both algorithms run on the Sparcs are due to poor scheduling decisions made by the scheduler at high loads. Maximum speed-up depends on the size of the search sequence and architecture.

4.5. Pipeline example

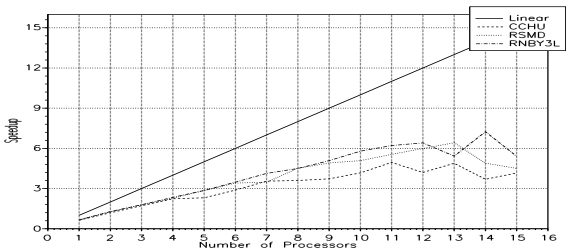
This example illustrates the construction of a simple pipeline process. It is presented to illustrate the fact that even if the time spent within the pipe stages increases, the time to execute the entire process does not increase appreciably.



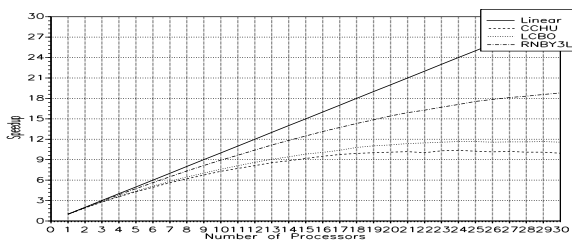
(a) Smith-Waterman on the Sparc IPCs



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



(c) FASTA on the Sparc IPCs



(d) FASTA on the Intel iPSC/2

Figure 7. speed-up for DNA Sequence Comparison.

4.5.1. Class Definitions

The following are the Mentat classes defi ned for this example.

```
regular mentat class gfilter{
public:
   int one_arg(int arg1);
}

persistant mentat class generic{
public:
   int one_arg(int arg1);
}
```

The member functions *gfilter.one_arg()* and *generic.one_arg()* each execute for some period of time. Consider the following code fragment:

```
generic node1, node2;
gfilter filter;
node1.create();
node2.create();
int i;
 for (i=0; i < MAX_ITERATIONS; i++) {
   int j;
   j = node1.one_arg(delay);
   j = filter.one_arg(j);
   j = filter.one_arg(j);
   j = node2.one_arg(j);
   j = 0;
}
```

This code fragment executes each of the four function calls in a loop. Note that the variable *j* is a temporary variable used as a conduit through which information passes between the fi lters.

In a traditional RPC system, this fragment would execute sequentially. Suppose that each member function execution takes 10 time units, and that each communication takes 5 time units. Then the time required to execute an iteration of the loop in a sequential RPC system is the sum of four times the member function execution time (due to the four member function calls), plus seven times the communication time (because all parameters and results must be communicated from/to the caller). Thus the total time required is 75 time units.

The average time per iteration for the Mentat version is considerably less, just over 10 time units. We arrive at this result by fi rst observing that the time for a single iteration is four times the communication time, or 20 time units, plus four times the execution time, or 40 time units, for a total of 60 time units. Next, consider that the reads, the two fi lter operations, and the writes can be executed in a pipelined fashion with each operation executing on a separate processor as shown in Figure 8.

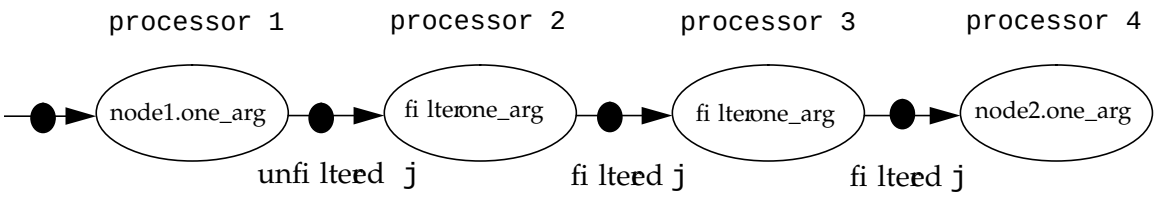


Figure 8. Program Execution Graph for the Pipeline Example.

Under these circumstances each of the four member function invocations, and all of the communication, can be performed concurrently. The communication for the *i*th iteration can be overlapped with the computation of the (*i*+1)th iteration. (We assume that communication is asynchronous and that sufficient communication resources exist.)

Using a standard pipe equation

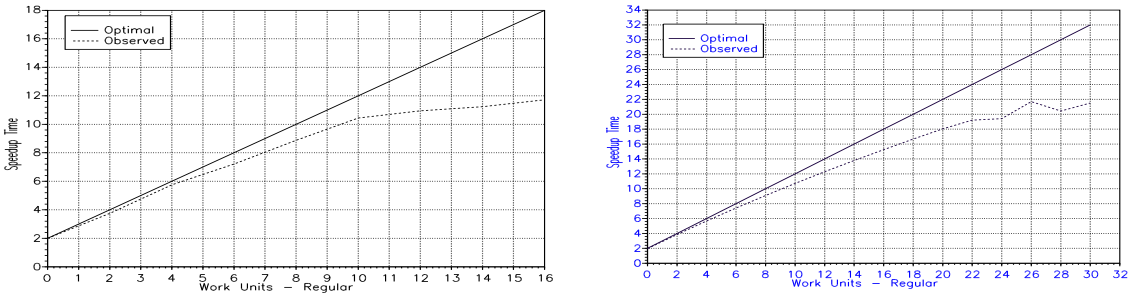
T_{All} = time for all iterations
T_{Stage} = time for longest stage = 10 time units
T_1 = time for first iteration = 60 time units
T_{Avg} = average time per iteration
$T_{All} = T_1 + T_{Stage} \times (MAX\_ITERATIONS\text{-}1)$
$T_{All} = 60 + 10 \times (MAX\_ITERATIONS\text{-}1)$
$T_{Avg} = (60 + 10 \times (MAX\_ITERATIONS\text{-}1))/MAX\_ITERATIONS$

When *MAX_ITERATIONS* is one, the time to complete is 60 time units, with an average of 60 time units. This is faster than a pure RPC (75 time units) because we don't send intermediate results to the caller. However, as *MAX_ITERATIONS* increases, the average time per iteration drops, and approaches 10 time units.

There are five points to note from this example. First, the variable *filter* is an instance of a regular Mentat class. So the system is free to instantiate new instances at will. At any given time there will be two instances executing. Second, the main loop may have executed to completion (all *MAX_ITERATIONS* iterations) before

the first write has completed. Third, suppose our "caller" (the main loop) was itself a server servicing requests for clients. Once the main loop has completed the caller may begin servicing other requests while the first request is still being completed. Fourth, the order of execution of the different stages of the different iterations can vary from a straight sequential ordering, e.g., the last iteration may "complete" before earlier iterations. This can happen, for example, if the different iterations require different amounts of filter processing. This additional asynchrony is possible because the run-time system guarantees that all parameters for all invocations are correctly matched, and that member functions receive the correct arguments. The additional asynchrony permits additional concurrency in those cases where execution in strict order would prevent later iterations from executing even when all of their synchronization and data criteria have been met. Finally, in addition to the automatic detection of inter-object concurrency, we may also have intra-object parallelism encapsulation, where each of the invoked member functions may be internally parallel. Thus we obtain even more parallelism.

Now consider the effect of quadrupling the time to execute the filter from 10 to 40 time units. The time to execute the traditional RPC version goes from 75 to 135 time units. But the time per iteration for the Mentat version remains unchanged at 10 time units if there are sufficient computation resources. To see why, consider the regular Mentat class *gfilter*. This means that the system may instantiate new instances of this class at will to meet demand. As a result, there would be eight instances of the *gfilter* class active at a time, four performing the first filter, and four performing the second filter



(a) Pipelining on the Sparc IPCs (b) Pipelining on the Intel iPSC/2

Figure 9. speed-up for the pipeline example.

4.5.2. Communication and Computation Complexities

The communication complexity in terms of the number of messages transmitted is four times the number of iterations.

4.5.3. Performance

The speed-ups obtained on the Sparc IPC's and the Intel iPSC/2 are shown in Figure 9. As the number of worker units increase, the start-up costs become more apparent as fall off from the optimal. To completely amortize the start-up cost (as in the optimal case), the value of *MAX_ITERATIONS* has to be large(i.e., tend to infi nity for best results). In the above experiment *MAX_ITERATIONS* was set to 10 for experimental purposes.

5. Conclusion

Mentat provides a common environment within which diverse applications can be implemented on a wide-range of MIMD platforms. There are three advantages to using Mentat for developing programs on parallel machines: (1) the

user is relieved of managing the complex details of communication, synchronization, and scheduling, (2) the resultant code is portable across various MIMD platforms, and (3) the implementation can exploit the underlying parallel architecture to improve performance. In this report we have used several applications to prove these points by example.

Mentat supports an object-oriented parallel programming approach. As a consequence, the decomposition of the problem results in objects that may be executed concurrently with other objects. Mentat detects such opportunities, and also detects when operations within an object may be made concurrent. Mentat then makes all of the communication and synchronization transparent to the user, and handles the scheduling of the various tasks onto the processors.

In addition to aiding the programmer in organizing an implementation of a problem and supporting concurrency, the object-oriented approach by its nature abstracts out the details of underlying structures. Each of the application examples were implemented on the three platforms currently in our testbed. As a result of this abstraction the application programs were ported between the various platforms with only minor changes to the compile directives.

Mentat provides the best performance improvements to applications that can be decomposed into medium-grain parallel tasks. The DNA sequence comparison and the Image Convolution applications are well-suited for this type of implementation, as evidenced by their near-linear speed-up as the application used more processors. Applications such as the Process Pipeline, where communications between workers can be overlapped, show dramatic performance increases. However, applications that require frequent

synchronization, and hence have fine grain, do not demonstrate the same performance increase when increasing the number of processors.

6. References

[1] A. S. Grimshaw, "The Mentat Run-Time System: Support for Medium Grain Parallel Computation," *Proceedings of the Fifth Distributed Memory Computing Conference*, pp. 1064-1073, Charleston, SC., April 9-12, 1990.

[2] A. S. Grimshaw, E. Loyot Jr., and J. Weissman, "Mentat Programming Language (MPL) Reference Manual," University of Virginia, Computer Science TR 91-32, 1991.

[3] A. S. Grimshaw, "An Introduction to Parallel Object-Oriented Programming with Mentat," TR-91-07, Department of Computer Science, University of Virginia, April, 1991.

[4] A. S. Grimshaw, 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.

[5] A. S. Grimshaw, E. West and W. R. Pearson, "No Pain and Gain! - Experiences with Mentat on a Biological Application," to appear in *Proceedings of Symposium on High Performance Distributed Computing*, Syracuse, NY, September 1992.