No Pain and Gain! - Experiences with Mentat on a Biological Application

Andrew S. Grimshaw and Emily A. West, Department of Computer Science
William R. Pearson, Department of Biochemistry
University of Virginia
grimshaw@virginia.edu

Abstract
Throughout much of the parallel processing community there is the sense that writing software for distributed memory parallel processors is subject to a "no pain - no gain" rule: that in order to reap the benefits of parallel computation one must first suffer the pain of converting the application to run on a parallel machine. We believe this is the result of inadequate programming tools and not a problem inherent to parallel processing. We will show that one can parallelize real scientific applications and obtain good performance with little effort if the right tools are used. Our vehicle for this demonstration is a six-thousand line DNA and protein sequence comparison application that we have implemented in Mentat, an object-oriented parallel processing system for both parallel and distributed architectures. We will briefly describe the application and present performance information for both the Mentat version and a hand-coded parallel version of the application.

1: Introduction
There is a persistent notion in the parallel processing community that writing software for distributed memory parallel processors is inherently hard, and that it is subject to a "no pain - no gain" rule: that in order to reap the benefits of parallel computation one must first suffer the pain of converting the application to run on a parallel machine. This notion is driven by the low level at which many of these machines must be coded, typically in C or Fortran extended with send and receive. Such tools are the "assembly language" of parallelism. They can be difficult to use, and debugging such programs can be very painful. Fully automatic compilation tools such as Paraphrase [11,16] and PFC Fortran [1] are best at finding small-grain parallelism. Distributed memory machines such as net-works of workstations or the Intel iPSC/2 require medium-grain parallelism. Thus, fully automatic techniques are inappropriate for distributed memory architectures. We believe that the problem is not the architectures, but the tools that have been used to program them. We will show in this paper that one can parallelize a real scientific application and obtain good performance with little effort if the right tools are used.

The tool that we have used is Mentat [7-9], an object-oriented parallel processing system developed at the University of Virginia. Mentat abstracts parallelism in C++ [19] objects. The user is responsible for identifying object boundaries and specifying those object classes that have sufficient computational complexity to warrant parallel execution. The Mentat compiler and run-time system are responsible for managing all aspects of communication, synchronization, and scheduling for the user. Mentat performs those tasks that humans perform poorly, while the programmer performs those tasks (data and program decomposition) that compilers perform poorly. Thus, Mentat exploits the capabilities of both compilers and humans. Mentat is currently available on a range of platforms from networks of heterogeneous workstations to tightly coupled machines such as the Intel iPSC/2. As an added bonus, applications developed on one platform are source code-portable from one platform to another. This eliminates another problem common to writing software for parallel architectures, that applications are not portable across platforms.

The vehicle of our demonstration is a six-thousand line DNA and protein sequence comparison application [4]. The application was chosen for four reasons: 1) it is a real, non-trivial, scientific code; 2) the sequential code was readily available; 3) the time required to execute the application on sequential machines is hampering further developments in the field; and 4) the application had previously been hand parallelized for the Intel hypercube, providing us with the opportunity to compare the performance of hand generated parallelism against our compiled Mentat version.

1. This work was partially supported by a grant from the National Laboratory of Medicine (LM04969) and by NSF grant CDA-892354-01.
We begin our presentation by first describing the application domain. We next introduce Mentat, briefly discussing the objectives, attributes, supported systems, and syntax of the programming language. The sequential algorithms, the hand-coded parallel version, and the Mentat versions of the application are then described, followed by performance results. Performance results for both the Intel iPSC/2 and a network of Sun SPARC station IPC's are given. We also discuss the programmer time required to implement the two parallel versions. The results clearly support our no pain and gain claim: the performance penalty associated with Mentat is negligible for this application, while the implementation in Mentat was straightforward.

2: Background
2.1: Biological Sequence Comparison

With advances in DNA cloning and sequencing technologies, biochemists and molecular biologists today can determine the sequence of a protein far more easily than they can determine its function. Historically, biochemists first characterized the chemical reactions of biological processes, later purified the enzymes that catalyzed these reactions, and then, in rare instances, invested the months or years required to determine the primary structure of the protein - its sequence of amino acids. With modern cloning and sequencing technique this process has been reversed. Today, it is common to sequence first and ask the biological questions later. Indeed, as a result of the human genome initiative, it is likely that tens of thousands of protein sequences will be determined without any biological insight into their functions.

The most efficient way to determine the function of a protein, given its sequence, is to search for other proteins that have evolved from a common ancestor. If a newly determined protein sequence is very similar to an enzyme whose function is known, one would first ask whether the uncharacterized protein has a similar enzyme activity. Related proteins often perform related functions. In the hierarchy from three-dimensional protein structure to protein function to one-dimensional structure (or sequence), it is the sequence of a protein that changes most rapidly over evolutionary time and the three dimensional structure that is most highly conserved. Today, there are about 30,000 proteins whose sequence have been determined, but three-dimensional crystal structures are known for only about 300. Since three-dimensional structures are more highly conserved than sequences, proteins that appear related based on sequence comparison are very likely to share the same structure. Again, this insight can suggest a great variety of additional experiments that can be used to refine our understanding of the protein's function.

DNA and protein molecules are strings of building blocks: the building blocks are the four nucleotides (A,C,G,T) in the case of DNA and the 20 amino acids for proteins. Thus, comparing protein sequences or DNA sequences is a string distance problem [2,13,17,18]. DNA and protein sequence libraries are available that include most published sequences. In May, 1992, protein sequence libraries included approximately 10,000,000 amino acids in 36,000 sequences, while the DNA sequence libraries contained about 65,000 entries and 85,000,000 nucleotides. Comparing a newly determined protein sequence (300 amino acids) to a protein sequence library of this size with a rigorous dynamic programming algorithm \(O(n^m)\) requires about 5 hours on a high-performance Unix workstation; to search the DNA sequence library would take about 30 times as long. To address this problem, heuristic methods have been developed that are 100 - 1000 times faster than the rigorous approach [2]. Today, these programs are widely used throughout the world; as many as 200 FASTA searches are performed every day in one DNA sequence repository in the US alone.

Sequence comparison programs thus provide a spectrum of computational demands. The rigorous Smith-Waterman algorithm \(O(nm)\) is the most time consuming and sensitive sequence comparison algorithm. The heuristic FASTA program uses a lookup table to rapidly identify regions with high densities of identity and can search libraries 20 - 100 times faster, depending on the granularity of the lookup table (1 residue, 2 residues, etc.). BLAST [2] uses a statistical criterion to build a DFA that can scan sequence libraries 1000 times faster than the rigorous approach.

The use in recent years of these and other algorithms has greatly increased the pool of identified proteins, and as libraries grow in size, more time is required to complete a comparison of an unknown sequence. While incremental improvements in sequential algorithms are expected, it is unlikely that algorithmic performance will improve as fast as the libraries increase in size. It has been suggested that there are approximately 100,000 different proteins encoded by the human genome; to date, approximately 5 - 15% of these sequences have been determined (the 30,000 sequences in the protein databases include many redundant sequences, i.e. the same sequence determined in many different organisms). The fifteen year goal of the human genome project is to determine the entire 3,000,000,000 nucleotides of the human genome. As sequences accumulate and more sophisticated questions are asked, higher performance machinery must be brought to bear on the problem.

Biological sequence comparison is ideally suited to parallel computation. Each comparison between the query sequence and an entry in the sequence library is com-
pletely independent. Consequently, the Smith-Waterman and FASTA algorithms have been ported to a limited number of parallel architectures. However, since parallel codes are typically restricted to the particular machine on which they were developed, these applications are only available to biologists with access to such machines. Exchange of information and programs among researchers is often vital, and the existence of a myriad of computing resources available to domain scientists raises the question of portability with respect to the comparison algorithms. Portability and performance are two of the issues addressed by Mentat. Furthermore, Mentat is available for the networks of workstations that are readily available at most institutions.

2.2: Mentat

Mentat was designed to address two problems that plague programming parallel MIMD architectures. First, writing parallel programs by hand is very 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 considerable investment of time and energy. Second, once implemented on a particular MIMD architecture, the resulting codes are usually not usable on other MIMD architectures; the tools, techniques, and library facilities used to parallelize the application are specific to a particular platform. Thus, 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 architectures, this represents a continuing time investment.

The three primary design objectives of Mentat are to provide: 1) easy-to-use parallelism; 2) high performance via parallel execution; and 3) applications portability across a wide range of platforms. The premise underlying Mentat is that writing programs for parallel machines does not have to be hard. Instead, 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. Instead of worrying about and managing these details, the programmer is free to concentrate on the details of the application. The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. The complex tasks are handled by Mentat.

There are two primary components of Mentat: the Mentat Programming Language (MPL) [7,8] and the Mentat run-time system [9]. The MPL is an object-oriented programming language based on C++ [19] that masks the complexity of the parallel environment from the programmer. The granule of computation is the Mentat class member function. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control.

Mentat classes are denoted by the inclusion of the keyword "mentat" in the class definition, as in the mentat class sw_worker shown below. Mentat classes may be defined as either persistent or regular. Instances of regular Mentat classes are logically stateless, thus the implementation may create a new instance to handle every member function invocation. Persistent Mentat classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data, or that require persistent semantics. Of course, the maintenance of state has implications for determinism.

Instances of Mentat classes are used exactly like C++ classes, as in the fragment below. Note that Mentat provides far more than just RPC. In the code fragment above, the thread of control executing the statement will not block on the member function invocation. Instead the thread will block only if it requires the result in an strict expression in which the value must be available for the thread to proceed. If the data value is used in another Mentat expression, the data is transferred directly to the consumer. If the result is never used in a strict expression in the calling thread, then a copy of the result will not be transferred to the caller.

Mentat extends object encapsulation from implementation and data hiding to include parallelism encapsulation. Parallelism encapsulation takes two forms that we call intra-object encapsulation and inter-object encapsulation. Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of a member function is sequential or parallel. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between

59
the different Mentat object member functions they invoke. Thus, the data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

The computation model underlying Mentat is the macro data flow model [12], a large-grain, graph-based, data-driven computation model. The Mentat run-time system supports the macro data flow model via the provision of a virtual macro data flow machine. The virtual machine provides support routines that perform run-time data dependence detection, program graph construction, program graph execution, scheduling, communication, and synchronization. The compiler generates code that communicates with the run-time system to correctly manage program execution.

The virtual machine model permits the rapid transfer of Mentat to new architectural platforms. Only the machine-specific components need to be modified. Because the compiler uses a virtual machine model, porting applications to a new architecture does not require any user source level changes. Once the virtual machine has been ported, user applications are re-compiled and can execute immediately.

An alpha release of Mentat for Sun 3's, Sun 4's, the Intel iPSC/2, and the Silicon Graphics Iris is available. We are currently porting Mentat to the TMC CM-5 and the Intel Delta (Paragon). Performance results on a range of applications are available, and are quite encouraging.

In summary, we believe that by splitting the responsibility between the compiler and the programmer, we can exploit the strengths and avoid the weaknesses of each. The underlying assumption is that the programmer can make better decisions regarding granularity and partitioning, while the compiler can better manage synchronization. This simplifies the task of writing parallel programs and makes parallel architectures more accessible to non-computer scientists.

3: Implementations

There are two basic problems that need to be solved in biological sequence comparison, sequence-to-library comparison, and library-to-library comparison. These are referred to as scanlib and complib respectively. There are three different implementations of each problem, a sequential implementation, a hand-coded parallel implementation for the iPSC/2 (hereafter referred to as the hand-coded implementation), and a Mentat implementation. For all implementations, versions exist that use the two algorithms, Smith-Waterman and FASTA.

3.1: The Sequential Implementation

The sequential version of scanlib (see Figure 1) reads in the query sequence from disk, opens the library, and performs initialization of tables and parameters used in the sequence comparison. Each sequence in the library is then sequentially read from the library file and scored against the query sequence using either the FASTA or Smith-Waterman algorithm. Only the top 6000 scoring sequences are kept as the algorithm progresses through the library. The calculation of the mean, standard deviation and a histogram of all scores is output to the user. In addition, the user may request to see any number of the highest scoring sequences of the comparison. Often the first scoring uses the FASTA algorithm because of its greater speed. If the user requests to see the highest scoring sequences they are re-scored using the slower, but more accurate, Smith-Waterman algorithm. This allows exploitation of the speed of the FASTA algorithm, and reserves the use of the more accurate, but time-consuming, Smith-Waterman algorithm for the sequences of interest.

3.2: The Hand-Coded Implementation

The hand-coded implementation of scanlib is described completely in [4]. It is written in C with Intel iPSC/2 NX/2 [10] message passing calls. The implementation took six months to complete (some of the time must be attributed to learning the machine).

The implementation consists of three separate entities, a host program, a manager program which resides on

```plaintext
initialization;
get query sequence from disk;
get name of library;
initialize algorithm specific data structures;
open the library;
while (get next library sequence != NULL) {
    compare with query sequence
    enter score in histogram
    if one of better scores - save it
    if max # of allowed best scores drop bottom 3/4
} 
compute mean and std. dev. of all scores
display histogram
if FASTA algorithm then recompose n best scores from all scores
   display histogram
   display best score sequence
   display best score
   show user n best scores
```

Figure 1. Sequential algorithm pseudo-code.
node 0, and a worker program for each node in the subcube acquired by the host program. The host program is responsible for loading node 0 with the manager process, loading the remaining nodes with the worker program, and handling all user interaction. The manager process handles all communication with the worker processes. It partitions the library into a number of pieces (not necessarily equal to the number of workers) and assigns each node a piece. A copy of the query sequence is also sent to each of the worker processes. When a worker has completed the comparisons between the query sequence and its library portion, it returns the results to the manager process and is assigned a new piece of the library. This cycle continues until all library sequences have been compared with the query sequence. The manager then sorts the similarity scores and reports them to the host process. The worker processes simply receive the query sequence, perform initialization, and compute similarity scores. The libraries are stored on the Concurrent File System (CFS) in order to allow concurrent access by the workers.

3.3: The MPL Implementation

Our current implementation of scanlib in Mentat is based on the sequential version and took two days to implement. We began by defining three new classes, result_list, sequence, and sw_worker (Figure 2). A result_list is a variable size list of result records. Each result record has the library offset of a sequence and its score. A sequence is a variable size structure that contains the name of the sequence, whether it is a DNA or protein sequence, and the actual sequence itself.

![Figure 2. Salient features of Mentat scanlib.](image)

The class sw_worker is a regular Mentat class. The member function compare() does all of the work, taking as arguments a library structure that defines a subrange of the library, a parameter structure, and the sequence to compare. The function returns a result_list. Upon receiving its parameters, each instance opens the library and seeks to the beginning of its library portion. A score is generated for each library sequence by comparing it against the query sequence using the specific scoring technique, e.g., Smith-Waterman or FASTA. When all scores for a library parcel have been computed by a particular worker, the library file is closed, the worker returns the results to the manager, and the worker terminates. Again, the library file is kept on the CFS to allow concurrent access by the workers.

The main program accepts as inputs a single sequence, the name of a library of sequences, and the number of workers to be assigned a portion of the library. The size of the library is determined, and the portion boundaries are computed for each worker. Note that one instance of sw_worker is defined. The first loop iterates num_worker times, calls worker.compare(), and puts the result into an array of result pointers. Because sw_worker is a regular Mentat class, a new instance is created by the run-time system for every invocation. The programmer is not responsible for scheduling the computations or for managing communication and synchronization between the workers and the main program. The second loop is a nested loop in which each of the results returned from the workers is used to compute statistical and histogram information.
One problem with this implementation (shared by the hand-coded version) is that I/O is a significant fraction of the execution time for the faster algorithms, and limits performance. This is clear in the performance data later.

3.3.1 Complib: After our initial success with scanlib we began to design the Mentat version of compilib, the library-to-library comparison application. We decided to change our approach in order to reduce the I/O overhead and to provide a solution that is both more flexible, and that can be scaled to larger numbers of processors. Toward this end we decided that the workers will be persistent, and that they will load their portion of the library exactly once and keep it in memory. This design decision has an important consequence. The I/O overhead of loading the library is a one time cost that is amortized over all comparisons. This significantly increases the computation granularity for algorithms such as FASTA.

We define three new Mentat classes, genome_lib, collator, and recorder. Their interfaces are shown in Figure 3. The persistent genome_lib class open() function opens the library and breaks ownership of the library into "num_workers" pieces. Each piece is an instance of genome_lib. The implementation may transparently construct a multilevel hierarchy of genome_lib objects.

Hierarchical construction is important to control overhead. Suppose no hierarchy is constructed and 256 children exist. To perform an operation, e.g., compare_sw0, on the children requires either a multicast or 256 separate messages. The parent, therefore, is a communication hot-spot and will likely become overloaded. In the case of 16 children, each with 16 children, the parent requires only 16 messages, and each child requires 16 messages. The total number of messages sent is increased from 256 to 272, but the message effort, and traffic, is distributed to the children. We believe this will improve performance and provide for a scalable solution.

The collators are instantiated and scheduled on the fly as needed, ensuring a balanced load. The collator class is used to sort and merge the results produced by genome_lib objects.

The recorder class is a base class that prints the results of sequence-to-library comparisons to a text file. Historically, because of the slowness of processing, the text files could be examined by hand. As the data volume increases due to faster processing this will not be acceptable. We envision deriving new classes off of recorder that will process and present the data in more useful ways.

Given these classes, the main program is straightforward and is shown in Figure 4. The execution of this program creates a stream of requests "flowing" through the graph in Figure 5 in a pipeline fashion. This is an example of inter-object parallelism. The construction of this program graph is completely transparent and is shown for illustrative purposes. All aspects of communication and synchronization are managed for the programmer.

Additional intra-object parallelism is realized when the compare_sw elaborates into a parallel subgraph. The pseudocode is shown in Figure 6 (a), the resulting graph in 6 (b), and the total execution graph for a single sequence comparison in Figure 7. The pseudocode is for the special case of a fan-in of four workers. In the actual

```cpp
main () {
    // initialization
    recorder pp; // post processor (display)
    genome_lib source, target;
    pp.create(); source.create(); target.create();
    source.initialize("source_name");
    target.initialize("target_name");
    source.open(1);
    target.open(1);
    num_s_seq = source.num_seq();
    for(i=0; i<num_s_seq; i++) {
        // for each sequence
        s_val = source.get_next();
        // Compare against target library
        target.compare_sw(s_val);
        // Display results
        pp.display(target.compare_sw(s_val), s_val);
    }
}
```

Figure 4. Main program for compilib.

```cpp
Figure 5. Program graph for single iteration of compilib.
```

```cpp
Figure 3. Mentat classes for compilib.
```
if (num_children > 1) {
    collator coll;
    rtf(coll.merge(
        child[0].compare_sw(s_val),
        child[1].compare_sw(s_val),
        child[2].compare_sw(s_val),
        child[3].compare_sw(s_val));
    //return to graph *descendants*)
else {
    //actually do the comparisons
    //building and returning a result_list
    rtf(result);
}

code the number of workers may be quite large. There are then log(num_workers) collator layers. Note that in general, the children may have children, and will therefore be expanded into subgraphs.

Figure 6a. compare_sw() Implementation.

Code fragment for compare_sw().

At first glance the process of dynamically generating graphs and transferring arguments may seem expensive, and likely to incur a high latency. If only one sequence were being compared this would be true. However, we are comparing thousands of sequences. The graph forms a long pipeline, and while the time to fill the pipe may be high, once the pipe is full, outputs are generated at a high rate. Thus, the impact of the high latency is negated by overlapping computation and communication.

4. Results
4.1 Implementation - No Pain

As noted earlier, the Mentat version of scanlib was based on the sequential C version. The conversion process from straight C to MPL went very smoothly and took only two days. More than half of the time required to perform the conversion was spent changing the C function prototypes and definitions to ANSI C style, which the C++ compiler requires. The two days for the Mentat version can be contrasted with the four to six months required to hand-code the application for the iPSC/2.

The Mentat version of compilib has taken longer. There was no sequential version with which to start. Instead, we wrote a sequential C++ version from scratch that called C routines to actually do the comparison. Writing the sequential version took four weeks. Once the sequential code was debugged, two days were required to convert the code to MPL and find bugs that unfortunately did not manifest themselves in the sequential code. Several additional days were spent waiting for the Mentat staff to fix system bugs uncovered during testing. In total, six weeks have been spent on the Mentat compilib, including delays incurred from extraneous events. We attribute much of the rapid implementation to the fact that we needed to concentrate only on the algorithms, not on scheduling, communication, or synchronization.

The ease of implementation is one of the most important results of this effort. We have shown that a parallel code can be constructed in very little time with a small cognitive burden and with good performance results. As a further bonus, unlike the hand-coded iPSC/2 code, the MPL code can be readily moved to all architectures supported by Mentat. The resulting code also has very good performance characteristics.

4.2 Performance - The Gain

The objectives of our performance testing were two-fold. First, we measured the performance penalty associ-
ated with Mentat by comparing Mentat performance with the hand-coded version. Second, we measured the absolute speedup attained when using Mentat on this application. To accomplish the first objective we compared both the speedups and absolute wall clock times. Wall clock times were measured from just after the last parameter has been input and parsed to when the results are available and ready for display. We chose these two points because the applications are interactive and we felt that user time should not be included, and because all of the real computation time and I/O occurs between these two points.

The scanlib experiments were conducted in two different environments, an Intel iPSC/2 with attached CFS and four megabytes of memory per node, and a network of sixteen Sun IPC SPARC stations. The SPARC stations were each equipped with sixteen megabytes of physical memory and a local hard-disk for swap and system executables. All sixteen of the workstations were attached to a single ethernet segment served by a common file server. Both the file server, the network segment, and the workstations are public resources and were in use during the experiments. Every effort was made to reduce the amount of interference from other users. In particular, all experiments on the Suns were run between the hours of 2:00 am and 6:00 am. Oddly enough, there are usually several (graduate student) users at that hour.

The experiments consisted of running the two algorithms, Smith-Waterman and FASTA, on the two hardware platforms, and on a range of sequences from small (CCHU 104 residues), to medium (LCBO or RSMD, 229 residues), to large (RNBY3L, 1490 residues). The hand-coded results were obtained from [4]. Not all sequence sizes were run on all algorithms on all architectures. The small sequence (CCHU) was not run on the iPSC/2 because there are no results in [4] against which to compare. The large sequence (RNBY3L) was not run with Smith-Waterman on the iPSC/2 because collecting a single data point would take over ten hours and would not provide any useful additional information. The results are shown in terms of wall-clock time in Table 1. As the results in Table 1 clearly demonstrate, the performance penalty of using Mentat as opposed to hand-coding is negligible even for the faster algorithm, FASTA.

Speedups for the Mentat versions are also very good. Speedup is determined by taking the sequential C code execution time and dividing by the Mentat parallel execution time. Figure 8 shows speedup for both the Mentat and hand-coded versions on the Intel iPSC/2, while Figure 9 shows speedup for the Mentat version on the Sun network.

A few observations are in order. First, speedup is almost linear for Smith-Waterman on both architectures. Second, FASTA on both the iPSC/2 and the Suns suffers from too small a computation granularity in scanlib because I/O and communication dominate. This effect can clearly be seen as speedup ceases to improve with additional processors. The maximum speedup depends on both

<table>
<thead>
<tr>
<th>Workers</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>MHC</td>
<td>M</td>
<td>MHC</td>
</tr>
<tr>
<td>LCBO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FASTA</td>
<td>2.8</td>
<td>2.7</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>SW</td>
<td></td>
<td></td>
<td>95.8</td>
<td>95.3</td>
</tr>
<tr>
<td>RNBY3L</td>
<td></td>
<td></td>
<td>7.3</td>
<td>6.9</td>
</tr>
<tr>
<td>FASTA</td>
<td></td>
<td></td>
<td>7.3</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Table 1: Scanlib execution times on the iPSC/2 in minutes.tenths.

Key: M - Mentat    HC - Hand-coded
the size of the search sequence and architecture. The hand-coded version suffers from this effect as well. Third, the "bobbles" in the Sun performance graphs when the number of pieces approaches the number of processors is a known effect and is the result of the scheduling algorithm in use [6]. The scheduling algorithm is a randomized algorithm that performs poorly under heavy load.

The early results for complib are also very promising. We have concentrated on FASTA as Smith-Waterman performance is very good even for scanlib. The complib experiments to date have been conducted only on the network of Sun IPC's, the more demanding of the two environments. (Our iPSC/2 failed before complib was ready. We have learned that finding working obsolete computer parts can be a trial.)

For the Sun complib-FASTA experiments we compared a small library of 70 sequences against a large library of 9633 sequences using a ktp of 2. We used Mentat config-urations of 2, 4, 8, and 16 processors. The speed-ups are shown in Figure 10. Speed-ups are very good up to four processors. At both eight and sixteen processors the efficiency is approximately 75%. We believe that the cause for the reduced efficiency is communication overhead. Unlike scanlib, complib is requires a great deal of communication. Communication on the Sun network is relatively slow. Unfortunately without the iPSC/2 data we lack a means of comparing with the hand-coded version. However, given our experience with scanlib we expect speed-ups to be better on the iPSC/2 because of both the better interconnection network and the slower processors.

The early results for complib are also very promising. We have concentrated on FASTA as Smith-Waterman performance is very good even for scanlib. The complib experiments to date have been conducted only on the network of Sun IPC's, the more demanding of the two environments. (Our iPSC/2 failed before complib was ready. We have learned that finding working obsolete computer parts can be a trial.)

For the Sun complib-FASTA experiments we compared a small library of 70 sequences against a large library of 9633 sequences using a ktp of 2. We used Mentat config-urations of 2, 4, 8, and 16 processors. The speed-ups are shown in Figure 10. Speed-ups are very good up to four processors. At both eight and sixteen processors the efficiency is approximately 75%. We believe that the cause for the reduced efficiency is communication overhead. Unlike scanlib, complib is requires a great deal of communication. Communication on the Sun network is relatively slow. Unfortunately without the iPSC/2 data we lack a means of comparing with the hand-coded version. However, given our experience with scanlib we expect speed-ups to be better on the iPSC/2 because of both the better interconnection network and the slower processors.

5. Summary

There is a persistent notion in the parallel processing community that writing software for distributed memory parallel processors is subject to a "no pain - no gain" rule. In this paper we have presented evidence that this is not always true. Given appropriate tools at the right level of abstraction, real applications can be parallelized for execution on networks of workstations or on distributed memory multicomputers with very little effort while realizing good performance. The tool that was used is Mentat, an object-oriented, parallel processing system developed at the University of Virginia. Mentat simplified the parallelization effort by managing all aspects of scheduling, communication, and synchronization. The programmer is responsible only for encapsulating large-grain computation in objects. The class encapsulation simplified development, and the inheritance features will
make future enhancements simpler.

Both the scanlib and complib applications are naturally very parallel. While this is certainly not true of all applications, it is the case for a large number of applications in the biological sciences. Consequently, our success with this data-parallel application does not indicate that Mentat will make all applications easy to parallelize. Even so, we believe that Mentat is a powerful tool that will work for a range of applications and are currently implementing other varieties of applications in Mentat to test this hypothesis. This project is one of the first.

Future work on this project falls into two areas. First, we will incorporate additional sequence comparison algorithms into the system; this will further the mission of making all applications easy to parallelize. While this is certainly not true of all applications, it is the case for a large number of applications in the biological sciences. Consequently, our success with this library-to-library comparison will be executed on large networks of workstations (more than 40) and on much larger multicomputers. This will test the scalability of the hierarchical complib algorithm.

6: References