Object-Oriented Parallel Processing with Mentat[1]

A. S. Grimshaw
Department of Computer Science, University of Virginia,
Charlottesville, VA {grimshaw@virginia.edu}

1. Introduction

Mentat is an object-oriented parallel processing system designed to address three problems that face the parallel computing community, the difficulty of writing parallel programs, the difficulty achieving portability of those programs, and the difficulty exploiting contemporary heterogeneous environments. Writing parallel programs by hand is more difficult than writing sequential programs. 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. If parallel computing is to become mainstream it must be made easier for the average programmer. Otherwise, parallel computing will remain relegated to specialized applications of high value where the human investment required to parallelize the application can be justified.

A second problem is that once a code has been implemented on a particular MIMD architecture, it is often not readily portable to other platforms; the tools, techniques, and library facilities used to parallelize the application may be 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. One can view the different platforms as one dimension of a two dimensional space, where the other dimension is time. One would like the implementation to be able to cover a large area in this space in order to amortize the development costs.

Finally there is heterogeneity. Today's high performance computation environments have a great deal of heterogeneity. Many users have a wide variety of resources available, traditional vector supercomputers, parallel supercomputers, and different high performance workstations. The machines may be connected together with a high speed local connection such as FDDI, ATM, or HIPPI, or they may be geographically distributed. Taken together these machines represent a tremendous aggregate computation resource.

Mentat was originally designed to address the first two of these issues, implementation difficulty and portability. The primary Mentat design objectives were to provide 1) easy-to-use parallelism, 2) high performance via parallel execution, 3) system scalability from tens to hundreds of processors, and 4) applications portability across a wide range of platforms. The premise underlying Mentat is that writing parallel programs need not be more difficult than writing sequential programs. Instead, it is the lack of appropriate abstractions that has kept parallel architectures difficult to program, and hence, inaccessible to mainstream, production system

[1]. This work is partially funded by NSF grants ASC-9201822 and CDA-8922545-01, National Laboratory of Medicine grant (LM04969), NRaD contract N00014-94-1-0882, and ARPA grant J-FBI-93-116.

programmers. The third issue, heterogeneity, is the focus of the Legion project which is an outgrowth of Mentat.

To date Mentat has been ported to a variety of platforms that span the latency and bandwidth spectrum; from heterogeneous workstation networks, to distributed memory MPP's, to shared memory machines[2]. Workstations include generations of Sun workstations, IBM RS 6000's, Hewlett Packard's, and Silicon Graphics workstations. MPP platforms have included the BBN Butterfly, the Intel iPSC/2, iPSC/860, and Paragon, the IBM SP-1 and SP-2, the Convex Exemplar, the Mieko CS-2, and the Silicon Graphics Power Challenge. Recently we have ported to Linux [5], allowing Mentat to run on PC-based systems as well. For the most part we have had applications source code portability between platforms. The only exception being the iPSC/860. Applications performance has been good as well, though not always good. Mentat is not appropriate for fine-grain applications that require either very frequent communication, or large volumes of communication. This is not surprising, as even hand-coded parallel programs often have difficulty with such applications.

The objective of this paper is to provide the reader with a solid introduction to Mentat and to provide intuition as to the performance that can be expected from Mentat applications. This will be accomplished by first examining the Mentat philosophy to parallel computing and reviewing the Mentat programming language basics. We will then move onto applications performance. For each of several applications we will address two questions. 1) What is the shape of the Mentat solution? 2) How did the implementation perform? We conclude with a retrospective and a look forward to our next system, Legion.

2. Mentat

The Mentat philosophy on parallel computing is guided by two observations. The first is that the programmer understands the application domain and can often make better data and computation decomposition decisions than can compilers. The truth of this is evidenced by the fact that most successful applications have been hand-coded using low-level primitives. The second observation is that management of tens to thousands of asynchronous tasks, where timing-dependent errors are easy to make, is beyond the capacity of most programmers. Compilers, on the other hand, are very good at ensuring that events happen in the right order and can more readily and correctly manage communication and synchronization than programmers. The Mentat solution is driven by those two observations, that there are some things people do better than compilers, and that there are some things that compilers do better than people. Rather than have either do the complete job, we exploit the comparative advantages of each.

Mentat is more than a programming language and run-time system. Mentat is a complete parallel-application development environment for workstations and MPP's. It consists of the Mentat Programming Language (MPL) [18] and all of the required supporting infrastructure. The infrastructure consists of the run-time system [20] that supports object invocation, program graph construction, and scheduling; support tools such as a complete post-mortem debugger, resource

[2] For more information on Mentat, an up-to-date list of platforms, or a copy of Mentat, see http://www.cs.virginia.edu/~mentat.

management tools, accounting tools, and system status tools; and extensive documentation and user tutorials. The workstation version of the run-time system is completely fault-tolerant. The system automatically adapts to host failure and recovery without human intervention.

The MPL is an object-oriented programming language based on C++. The programmer uses application domain knowledge to specify those object classes, called Mentat classes, that are of sufficient computational complexity to warrant parallel execution. 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 extends object implementation and data encapsulation to include parallelism encapsulation. Parallelism encapsulation takes two forms that we call *intra-object* 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 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.

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically there are differences. First, Mentat member function invocations are non-blocking, providing for the parallel execution of member functions when data dependencies permit. Second, each invocation of a regular Mentat object member function causes the instantiation of a new object to service the request. This, combined with non-blocking invocation, means that many instances of a regular class member function can be executing concurrently. Finally, Mentat member functions are always call-by-value because the model does not provide shared memory.

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 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.

2.1. Mentat Classes

In C++, objects are defined by their class. Each class has an interface section in which member variables and member functions are defined. Not all class objects should be Mentat objects. In particular, objects that do not have a sufficiently high communication ratio, i.e., whose object operations are not sufficiently computationally complex, should not be Mentat objects. Exactly what is complex enough is architecture-dependent. In general, several hundred executed instructions long is a minimum. At smaller grain sizes the communication and run-time overhead takes longer than the member function; resulting in a slow-down rather than a speed-up.

Mentat uses an object model that distinguishes between two “types” of objects, contained and independent objects.[3] Contained objects are objects contained in another object’s address

space. Instances of C++ classes, integers, structs, and so on, are contained objects. Independent objects possess a distinct address space, a system-wide unique name, and a thread of control. Communication between independent objects is accomplished via member function invocation. Independent objects are analogous to UNIX processes. Mentat objects are independent objects.

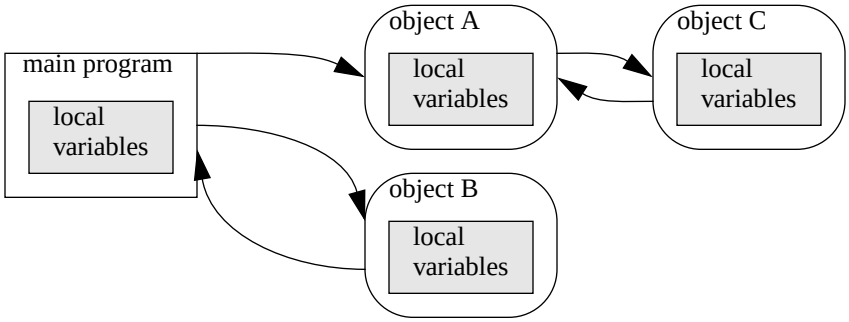


Figure 1 The Mentat object model. Mentat objects are address space disjoint and communicate via member functions and return values, shown here as directed arcs. Both the main program and Mentat objects may contain local variables.

The programmer defines a Mentat class by using the keyword **mentat** in the class definition. The programmer may further specify whether the class is **persistent**, **sequential**, or **regular**. Instances of Mentat classes are called Mentat objects.

Persistent and sequential objects maintain state information between member function invocations, while regular objects do not. Thus, regular object member functions are pure functions. Regular classes may have local variables much as procedures do and may maintain state information for the duration of a function invocation.

Mentat class interfaces may not have public member variables. The only way to modify or observe the state of a persistent or sequential Mentat object is via member functions. It is illegal to directly access member variables. This restriction follows from the address-space-disjoint nature of Mentat objects.

Example 1.

```
regular mentat class integer_ops {
public:
   int add(const int arg1,const int arg2);
   int mpy(const int arg1,const int arg2);
   int sqrt(const int val);
};
```

The interface for the regular Mentat class `integer_ops` defines three functions that operate on integers. Note that they are all pure functions, the output depends only on the input. Because they are pure function and do not depend on any persistent state the system may schedule invocations of those functions on any processor in the system.

---

[3] The distinction between independent and contained objects is not unusual and is driven by efficiency considerations.

Example 2.

```
sequential mentat class integer_accumulator {
   int running_total;
public:
   create(const int initial_value);
   void add(const int value);
   int current_value();
};
```

The persistent Mentat class `integer_accumulator` differs from the `integer_ops` class in that it does have state that persists from invocation to invocation. The state of the object is the member variable `running_total`. Once an instance of the class is created and the initial value assigned, subsequent calls to the add member function are applied to the same object, incrementing the value of the running_total. The function current_value returns running_total. We will describe use of the create() member in more detail later.

Mentat objects are used much as C++ objects. The code fragment below uses the integer_ops and integer_accumulator classes to sum the squares of n integers.

Example 3.

```
{
   integer_accumulator A;
   integer_ops B;
   A.create(0); // Create an integer_accumulator with
                // an initial value of 0
   for (int i=0; i<N; i++)
      A.add(B.mpy(i,i));
   cout << A.current_value();
}
```

In this example the loop is unrolled at run-time and up to N instances of the integer_ops class may execute in parallel. Note that parallel execution of the B.mpy() operation is achieved simply by using the member function. All of the A.add() operations are executed on the same object instance which is created and initialized with the A.create(0); statement.

The above examples illustrate the definition and use of Mentat classes. However, if we compiled and executed the above code fragments in Mentat, the parallel version of the program would execute far slower than an equivalent sequential program. The reason is that integer operations are of too fine a granularity for Mentat to exploit usefully. Larger grain operations are usually required for good performance.

2.2. Return-to-Future `rtf()`

The return-to-future function (`rtf()`) is the Mentat analog to the `return` of **C**. Its purpose is to allow Mentat member functions to return a value to the successor nodes in the macro data-flow graph in which the member function appears. Mentat member functions use the `rtf()` as the mechanism for returning values. The returned value is forwarded to all member functions that are data dependent on the result and to the caller *if necessary.* In general, copies may be sent to several recipients.

Example 4.

The implementation of the integer_ops::add() function illustrates the basic case well:

```
integer_ops::add(const int arg1, const int arg2) {
   rtf(arg1+arg2);
   return arg1+arg2; // C++ compilers require return
}
```

While there are many similarities between `rtf()` and `return`, `rtf()` differs from `return` in three significant ways. First, in **C**, before a function can `return` a value, the value must be available. This is *not* the case with an `rtf()`. Recall that when a Mentat object member function is invoked, the caller does not block. Instead the results are forwarded wherever they are needed. Thus, a member function may `rtf()` a "value" that is the result of another Mentat object member function that has not yet been completed or perhaps even begun execution. Indeed, the result may be computed by a parallel subgraph obtained by detecting inter-object parallelism.

Second, a **C** `return` signifies the end of the computation in a function while an `rtf()` does not. An `rtf()` only indicates that the result is available. Since each Mentat object has its own thread of control, additional computation may be performed after the `rtf()`, e.g., to update state information or to communicate with other objects. In the message passing community this is often called send-ahead. By making the result available as soon as possible, we permit data dependent computations to proceed concurrently with the local computation that follows the `rtf()`.

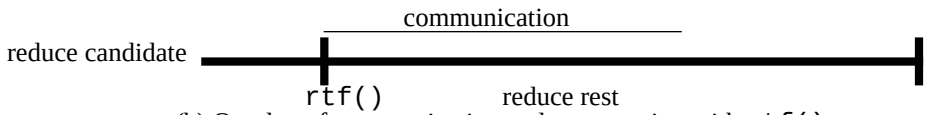
Third, a `return` returns data to the caller. Depending on the data dependencies of the program, an `rtf()` may or may not return data to the caller. If the caller does not use the result locally, then the caller does not receive a copy. This saves on communication overhead. The next two examples illustrate these features.

Example 5.

Consider a `persistent Mentat class sblock` used in Gaussian elimination with partial pivoting. In this problem we are trying to solve for x in Ax=b.. The `sblocks` contain

```
vector*sblock::reduce(vector* pivot) {
   reduce current column using pivot
   find candidate row
   reduce candidate row
   rtf(candidate row);
   reduce the rest of the sub-matrix
   return
}
```

(a) `sblock::reduce()` pseudo-code



(b) Overlap of communication and computation with `rtf()`.

Figure 2 Gaussian elimination with partial pivoting illustrating the use of `rtf()`

portions of the total system to be solved. The sblock member function

```
vector* sblock::reduce(vector*);
```

performs row reduction operations on a submatrix and returns a candidate row. Pseudo-code for the reduce operation is given in Figure 2*a*. The return value can be quickly computed and returned via `rtf()`. The remaining updates to the sblock then can occur in parallel with the communication of the result (Figure 2*b*). In general, the best performance is realized when the `rtf()` is used as soon as possible

3. Applications Experience

The bottom line for parallel processing systems is performance on real applications. Our goals include both performance and ease-of-use. Over the past three years we have tried to answer three questions about Mentat and our approach to object-oriented parallel processing.

- Is MPL easy to use?
- Is the performance acceptable to users?
- What is the performance penalty (if any) with respect to hand-coded implementations?

To answer these questions we set out to implement a set of both real and synthetic applications. Our application selection criteria was that the application must be representative of a class of applications, and that the application must be of genuine interest to identifiable users. Further, we wanted variety in our applications, not just linear algebra applications. In all cases the implementation processes has been carried out in collaboration with domain scientists (e.g., biochemists) who had an interest in the codes. For each application we are interested in the level of difficulty in implementing the code, and in the resulting performance. For some of the applications there already existed a hand-coded parallel C or Fortran implementation of the application. In those cases we compare to the hand-coded version.

The set of Mentat applications is diverse and includes DNA and protein sequence comparison (biochemistry), automatic test pattern generation (electrical engineering), genetic algorithms (searching a combinatorial space), image processing (both libraries and for target recognition), command and control, and parallel databases (computer science) to name a few. Only a few of Mentat application have been implemented by our research group at the University of Virginia. Below four of those applications are presented. The four applications are DNA and protein sequence comparison, a dense linear algebra library, a stencil library, and a 2D electromagnetic finite element method. For each of the four applications (or libraries that are used in several applications) we will address two questions. 1) What is the shape of the Mentat solution? 2) How did the implementation perform?

3.1. DNA and Protein Sequence Comparison[4]

Our first application is DNA and protein sequence comparison. 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 rela-

[4] Portions of this section first appeared in [19].

tionship 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 [33], FASTA [28], and Blast [1]. 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.

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 the *scanlib* problem. In *scanlib*, a 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.

A second common operation is to compare two sequence libraries, i.e., to compare very sequence in a source library against every sequence in the target library. For each sequence in the source library statistics are generated on how the sequence compares to the target library as a whole. This is known as the *complib* problem.

The Mentat implementation of *scanlib* uses `regular` Mentat class workers. The class definition for the Smith-Waterman worker is given in Figure 3.The private member variables have been omitted for clarity. The single member function, `compare()`, takes three parameters:

```
1  regular mentat class sw_worker {
2     // private member data and functions
3  public:
4     result_list *compare(sequence, libstruct, paramstruct);
5     // Compares sequence against a subset of the library. Returns
6     // a list of results (sequnce id, score).
7  }
```

Figure 3 . Class definition for scanlib worker. Note it is a regular class indicating that the compare function is a pure function, and that the system may instantiate new instances as needed.

`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 4. 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.

```
1  // master program routine -- initialization details omitted
2  // -- get the number of workers
3  // -- divide the library into a partition for each worker
4
5  sw_worker worker;
6  // invoke the comparison for each partition of the library
7  for (i = 0; i < num_workers; i++) {
8     // compute library parcel boundaries
9     results[i] = worker.compare(the_seq, libparcelinfo, param_rec);
10 }
11 // for each partition's result, and for each comparison
12 // within the partition, compute statistics
13 for (i = 0; i < num_workers; i++) {
14    for (j = 0; j < results[i]->get_count(); j++) {
15       // get_count() returns the number of scores returned.
16       // for each result, update mean, stdev, and other statistics.
17    }
18 }
```

Figure 4 . Code skeleton for the master program routine for scanlib.

Performance on the iPSC/2 is given in Table 1. Performance information is given for both the Mentat implementation and for a hand-coded message passing C implementation. A 3.8-

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

Workers	1		3		7		15	
Sequence/Algorithm	M	HC	M	HC	M	HC	M	HC
LCBO - FASTA	2:47	2:42	0:59	0:57	0:28	0:26	0:17	0:16
LCBO - SW	-	-	95:49	95:19	41:12	41:00	19:22	19:14
RNBY3L[a]- FASTA	7:19	6:56	2:30	2:33	1:07	1:04	0:35	0:32

a. The Smith-Waterman times are not shown. They provide no additional insight.

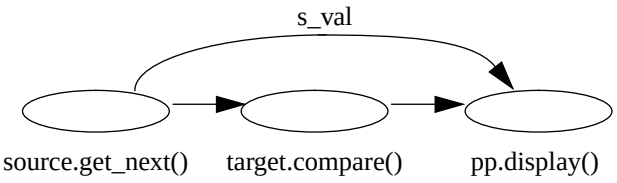
Mbyte target library containing 9633 sequences was used. LCBO and RNBY3L are different source sequences. They are 229 and 1490 bytes long respectively. Execution times for both the Smith-Waterman and much faster FASTA are given. Performance on this application clearly demonstrates that for naturally 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, the performance of the Mentat version is always within 10% of a hand-coded version, and usually within 5%

The complib implementation is more complex, though the main program remains straight forward. The main program manipulates three objects, the source genome library, the target genome library, and a recorder object that performs the statistical analysis and saves the results. See [19] for more detail. The main program for loop is shown in Figure 5 below. The effect is that

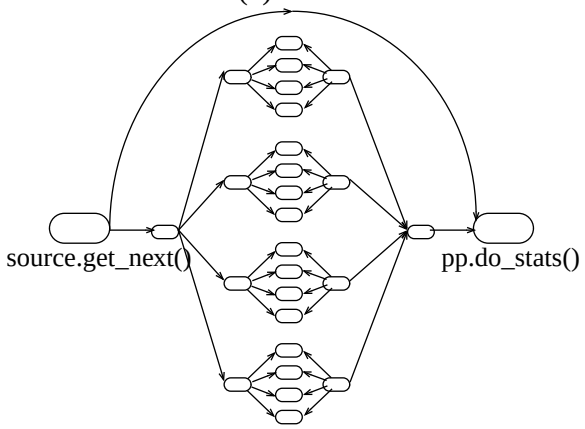
a pipe is formed, with sequence extraction from the source, sequence comparison in the target, and statistics generation are executed in a pipelined fashion. Each high-level sequence comparison is transparently expanded into a fan-out, fan-in program graph where the “leaves” are the workers, the source sequence is transmitted from the root of the tree to the leaves, and the results are collected and sorted by collators.

```
for(i=0;i<num_source_seq;i++) {
   //for each sequence
   s_val = source.get_next();
   //Compare against target library
   result = target.compare(s_val);
   //Do statistics
   post_process.do_stats(result,s_val);
}
```

(a)



(b)



(c)

Figure 5 Mentat implementation of *complib*. The main loop of the program is shown in (a). Three objects are manipulated, the source, the target, and the post_processor. The pipelined program graph is shown in (b). *Target.compare()* has been expanded showing sixteen workers in (c). The fan-out tree distributes the source sequence to the workers. The internal nodes of the reduction tree are collator objects. The reduction tree sorts and merges the results generated by the workers.

Table 2 presents results for five complib implementations on a network of 16 Sun IPC workstations. The execution times for four hand-coded implementations are compared to the Mentat time. (The non-Mentat implementations were done by William Pearson of the Department of Biochemistry at the University of Virginia.) To reduce the effect of the perturbations caused by other users five runs for each were taken, and the *best* times reported. Keep in mind that *complib* is not a low communication application. At fifteen workers over fourteen megabytes of data are moved through the pipeline shown in Figure 5. We used the faster FASTA algorithm, a twenty sequence source library and a 10,716 sequence target library. The same kernel C code to actually perform the comparisons was used by all five implementations. The sequential time (without any parallel constructs) is 583 seconds. The most surprising aspect of the results is that the performance of the tools is so similar. With the exception of Express most of the execution times are within 5% of each other.

Table 2- Complib performance on Sun IPC workstations. Execution time in seconds.

Workers	Express	Linda	Mentat (2.6)	P4 (1.3b)	PVM (3.2.6)
3	220	211	**202**	218	206
7	98	95	**91**	98	92
11	80	62	63	65	**60**
15	NA	50	**48**	49	49

3.2. Matrix Algebra

Linear algebra is the basis for a wide variety of applications. Because it is so often the computational kernel of applications and so well understood by users we chose to construct a set of non-optimized linear algebra libraries to package with Mentat as examples. We begin with a discussion of an important support class. For more information see [26].

The foundation for many Mentat applications, including the linear algebra library, is a C++ library that provides classes for two dimensional arrays and vectors. Much of the functionality of these classes can be found in numerous matrix library classes [12,31]. The Mentat implementation environment (disjoint address spaces) placed the requirement on our class that parameters passed as arrays must be memory contiguous. The class *DD_array* provides an interface to provide the user with a class instance that is contiguous in memory without binding the size of the array involved or its orientation in memory. (*DD_array* stands for “double dimension array, we did not want to start a class name with a “2”.) The *DD_array* methods provide for the decomposition of an array into subarrays, the creation of new arrays by overlaying or extracting from existing arrays, the access of array elements, and so on. Finally, *DD_array* serves as a base class from which arrays of any type can be created. Currently, arrays of floating point, double precision floating point, integer, and character have been developed. In addition, sparse and dense vectors of integers, floating point, and double precision floating point are derived from the class *DD_array.* The class *DD_array* also provides functions to decompose the array into one of five forms: by row, by column, cyclically by row, cyclically by column, or by block. For example, if the array consists of 100 rows, a row-cyclic decomposition into four pieces would allot rows 0, 3, 7,.. to the first subarray, 1,4,8,..to the second subarray, etc. A column-cyclic decomposition is analogous to a row-cyclic decomposition.

Using the *DD_array* class as a basis we then developed two Mentat linear algebra classes, one persistent, and one regular. The interface for the persistent class is shown below in Figure 6. (Space limitations prevent the presentation of both classes.) The implementations were coded in a classic master-worker/data parallel manner. Each matrix is internally sub-divided into several sub-matrices (*matrix_sub_block*s) which are themselves Mentat objects. Operations performed on the entire matrix are then implemented in the member functions by manipulating the sub-matrices.

Performance in the libraries is presented below in Table 3. The same network of 16 Sun IPC’s was used. In [26] we present more complete results, including Intel Paragon performance. Due to performance difficulties with NFS the run-time was measured from a point *after* the matrices had been loaded into memory[5]. The results are for single precision numbers. The matrix mul-

tiply uses a naive algorithm requiring $O(n^3)$ operations. The solver uses Gaussian elimination with partial pivoting which is also $O(n^3)$.

```
persistent mentat class p_matrix {
    int total_rows, total_cols;          // Total rows and columns in the array
    int num_pieces;                      // Number of pieces into which the array is divided
    int decomp_type;                     // Decomposition method
    p_matrix_sub_block *sub_arrays;      // Pointers to the objects holding the decomposed matrix
public:
     initialize(DD_floatarray *data, int decomp_type, int pieces);
     initialize(string *file_name, int decomp_type, int pieces);
     // Creates instances of p_matrix_sub_block and distributes the array data by  decomp_type.
     void scalar_add_mat(float scalar);              // Add a scalar to each of the subarrays.
     int transpose();                                // Perform the transpose of the array.
     sp_dvector *vec_mult(sp_dvector *arg_vect);     // Performs matrix-vector multiplication.
     DD_floatarray *mat_mult(DD_floatarray *data);// Performs matrix matrix multiplication.
     p_matrix  mat_mult(p_matrix);                   // Performs matrix matrix multiplication.
     sp_dvector *solve(sp_dvector *rhs_vector);      // Gaussian Elimination using partial pivoting.
  };
```

Figure 6. Function prototypes for the `persistent mentat class p_matrix`. Only a subset of the interface is shown.

Table 3 . Sun 4 Network Execution Times for 1024 x 1024 Matrices

Problem	Sequential Time	4	6	8	10	12	14
Regular Class							
Matrix Multiply	1380	363	258	202	163	206	193
Gauss	626	179	131	109	108	118	126
Persistent Class							
Matrix Multiply	1380	356	258	228	151	137	137
Gauss	626	172	123	102	99	107	126

3.3. Stencil Libraries[6]

Stencil algorithms are used in a wide range of scientific applications, such as image convolution and solving partial differential equations (PDE's). Stencil algorithms are a class of algorithms that have several features in common: (1) the input data set is an array of arbitrary dimension and size, (2) there is a *stencil* that defines a local neighborhood around a data point, (3) some function is applied to the neighborhood of points that are "covered" when the stencil is centered on a particular point, and (4) this function is applied to all points in the data set to obtain a

5. We found that a large number of simultaneous bulk I/O requests will bring NFS to it's knees. The packet loss rate, combined with other factors that we do not understand, often causes the I/O to fail.

6. Portions of this section first appeared in [23].

new data set. Figure 7-*a* shows a two dimensional 3 × 3 stencil that indicates that each output value will depend *only* on the "north," "east," "west," and "south" (called NEWS) neighboring points of the corresponding point in the input array. The associated function is an example of a stencil function that uses NEWS neighbors.

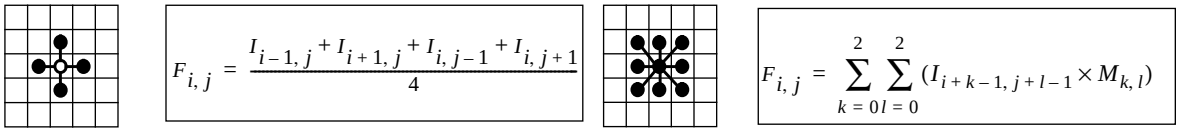


Figure 7. Typical 2 dimensional stencils. *F - final matrix, I - input matrix, M - convolution mask.* (*a*) 2D 3×3 NEWS stencil. (*b*) 2D 3×3 eight-connected stencil.

We have defined a base stencil class, `Stenciler`, that is designed to manage those issues that are common to all stencil algorithms while providing a framework for the user to create derived classes that can be tailored to specific applications [23]. The base class contains built-in member functions to perform common tasks, such as managing data communication between pieces. The base class also contains well-commented stubs for member functions that the user must define, such as the stencil function. This approach minimizes the effort needed to create new stencil applications through reuse of common code while supporting flexibility in creating parallel stencil applications. The user creates a new class derived from `Stenciler`. The derived class inherits all of the member functions of the base class, so instances of this new class have all of the built-in common functions provided with the `Stenciler` class. The user then supplies the application-specific code by overloading certain member functions.

An instance of a `Stenciler` or derived class is designed to handle one piece of the total array. Each `Stenciler` instance can create additional workers to split the work-load into smaller pieces. These pieces, in turn, may be further divided, creating a general tree structure of pieces as shown in Figure 8. Each new level of the tree has a "contained-in" relationship to the previous higher level. The pieces at leaves of this tree structure are the workers who perform the stencil function. The interior instances are managers for the workers below them; the managers distribute and synchronize the work of their sub-piece and collect the results. This hierarchical tree structure of processes is a powerful and flexible tool for decomposing a stencil problem, especially when running on different hardware platforms.

To illustrate the use of the stencil framework we describe our experience with two sample implementations: an image convolver and a PDE solver using Jacobi iteration.

Image convolution is a common application in digital image processing. In two dimen-

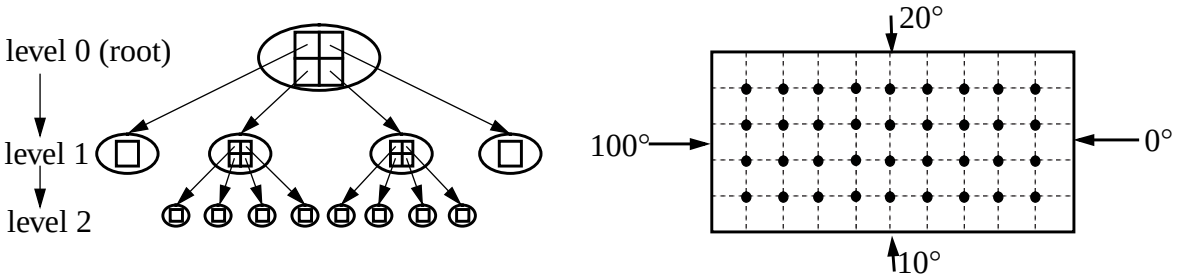


Figure 8 . On the left is a tree of `Stenciler` instances. In this case a 4-ary tree. On the right is the cannonical grid approximation of a heated plate used in our PDE example.

sional image convolution, a small 2D stencil, also called a *filter* or *mask*, defines a region surrounding each picture element (pixel) whose values will be used in calculating the corresponding point in the convolved image. Each element of the filter is multiplied by the corresponding neighbor of the current pixel, and the results are summed and normalized. Figure 7-*b* shows a stencil function for a 3×3 mask.

Another common class of stencil algorithms are iterative methods. Jacobi iteration is a method for solving certain systems of equations of the form $A\vec{x} = \vec{b}$, where A is a matrix of coefficients, $\vec{x}$ is a vector of variables, and $\vec{b}$ is a vector of constants. The general procedure for using Jacobi iteration is to first guess a solution for all variables, and then to iteratively refine the solution until the difference between successive answers is below some pre-determined threshold.

The application of Jacobi iteration used here is the classic heated plate problem. The heated plate problem consists of a plate or sheet of material that has constant temperatures applied around the boundaries, and the goal is to determine the steady-state temperatures in the interior of the plate (Figure 8). The temperature in the interior region is approximated by dividing the plate into a regular 2D grid pattern and solving for each of the grid points. The values at each point are approximated by the average of the values in the NEWS neighboring points. This transforms the problem into a system of linear equations which can be solved using Jacobi iteration. The form of the stencil for Jacobi iteration is shown in Figure 7-*a*.

To test performance we created two versions of each class, a sequential version written strictly in C++, and a parallel version written in C++/MPL. We executed the sequential versions on a Sun IPC and recorded the best of the wall-clock execution times. Similarly, we ran the parallel versions on a network of 16 Sun IPCs connected via ethernet. The parallel versions were executed decomposing the problem into from two to fourteen row pieces. Each decomposition was run several times and the best time for each decomposition was recorded.

For the `Convolver` tests, both the sequential and parallel versions were executed using identical problems: a 2000×2000 8-bit grey scale image convolved with three successive 9×9 filters. The `PDE_Solver` problem used a 1024×1024 grid of floating point numbers to estimate the interior temperatures of the heated plate problem. Table 4 shows the raw best execution times.

Table 4 Stenciler Performance Results

Number of Pieces	Convolver		PDE_Solver	
	Best Execution Time (min:secs)	Speedup	Best Execution Time (min:secs)	Speedup
1	49:33	N/A	43:39	N/A
2	24:33	2.0	23:01	1.9
4	12:37	4.0	11:37	3.8
6	8:47	5.6	7:53	5.5
8	7:07	7.0	7:08	6.1
10	5:53	8.4	6:47	6.4
12	5:08	9.7	6:23	6.8
14	4:57	10.0	6:10	7.1

3.4. 2D Electromagnetic Finite Elements[7]

The finite element method (FEM) has been in use for many years in structural mechanics and has become popular in recent years as a technique for use on electromagnetic problems. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding which follows an object's shape. This is of particular importance in electromagnetic (EM) scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. Finite elements are used in 2 and 3 dimensional EM scattering problems to model objects of complex composition. A "hand-coded" version of the code has been implemented on several MIMD computers by my collaborator using explicit message passing. A complete description of this code, along with parallel implementation description and performance, is found in [13]. A description of the Mentat implementation can be found in [36].

To solve the problem a 2D integral equation is transformed into a set of linear equations by decomposing the problem domain into a set of finite elements. The problem domain is meshed with nodal points at which the solution is to be found, matching the geometry of the objects. These nodes are then tiled with a set of finite elements. In 2D, the elements might be triangles or quadrilaterals. A set of basis functions are defined at each node in the mesh, which have nonzero value only within the elements of which it is a part. These basis functions are generally some polynomial function which is 1 at the node defining it, 0 at all other nodes in the element, and 0 along the edges of the element opposite the defining node.

The EM finite element application consists of two primary computation phases, matrix assembly and matrix solve. In matrix assembly, the finite elements compute contributions (i.e. matrix values) that are assembled (i.e. added) into the stiffness matrix **K**. The stiffness matrix is banded, symmetric, and very sparse. During assembly, the force vector **F** is also computed by the elements. This vector is the right-hand-side vector during the matrix solve computation.

In matrix solve the system of equations represented by the stiffness matrix with the force vector as the right-hand-side is solved by a conjugate-gradient algorithm known as Bi-conjugate gradient [21]. The algorithm uses three basic operations: matrix-vector multiplication, vector dot product, and vector saxpy. The solve phase poses challenges to achieving good performance on parallel machines due to the sparse nature of the matrix-vector operations.

The Mentat EM code was developed at JPL and run on a 64-node Intel iPSC/860 at Caltech. The data collected are from a data set that consisted of 2304 9pt quadrilateral elements (9313 nodes). This is considered a small problem. We computed speedups with respect to the sequential C++ EM code run on a single i860 node. The results are divided into the two dominate phases: 1) assembly is the time taken to complete the matrix assembly operations, and 2) solve is the time taken for the matrix solve operation. The sequential C++ is a factor of two slower for the solve phase, and a factor of three slower for the assembly phase. This is expected as the C++ compiler technology is much less mature than the Fortran technology; in particular the C++ compiler does not exploit the vector unit.

We compared the results with a hand-coded optimized parallel Fortran EM implementation that has been in development for some time. We expected the performance to be worse than the hand-coded version, but how much? The results indicate that this is indeed the case, though

---

[7] Portions of this section first appeared in [36].

TABLE 5 Mentat and hand-coded Fortran speedup on the iPSC/860

Processors/Phase	**2**	**4**	**8**	**16**
Assembly phase: Mentat	1.9	3.8	7.0	11
Assembly phase: hand-coded	2.0	2.9	7.2	11
Solve phase: Mentat	1.8	2.8	3.9	4.9
Solve phase: hand-coded	1.9	3.0	4.8	6.4

neither implementation achieves good speedup on the solve phase. They also indicate that the optimized version is scaling in a manner similar to the hand-coded Fortran. The assembly phase scales identically to the hand-coded while the solve phase scales almost as well. The slight discrepancy is due to Mentat overheads often seen for small problems. We expect the performance of the Mentat version to more closely match the hand-coded for larger problems.

4. Related Work

There are many other projects with objectives similar to Mentat's. One important dimension along which solutions to the parallel software problem can be placed is the level of programmer awareness and control of the parallel environment. Solutions can be placed along a spectrum that ranges from fully automatic solutions to completely explicit solutions. In fully automatic the programmer is completely freed from the responsibility of managing any aspect of the parallel environment. A parallelizing compiler, often acting in concert with a sophisticated run-time system, finds and *safely* exploits opportunities for parallelism in the application. The user is not responsible for threads, synchronization, or other details. At the other extreme are the explicit, manual systems where the programer writes programs in the assembly language of parallelism. The programmer must decompose the problem, distribute data structures, manage scheduling and communication, and *safely* manage synchronization between tasks.

An advantage of automatic techniques are that programs are deterministic and semantically equivalent to the sequential program. Compilers are very good at finding large amounts of parallelism in programs, often down to the statement level. Further, automatic techniques can be applied to "dusty deck" programs, usually Fortran, to leverage the huge existing software base. A disadvantage of automatic techniques is that often the granularity of the detected parallelism is too small to efficiently exploit on contemporary distributed memory machines. To the compiler, the program is a large directed graph. It must attempt to partition and schedule the graph onto the parallel computer. Finally, automatic techniques are often applied to dusty decks and are often defeated by spurious dependencies in the program that require programmer intervention. Despite these difficulties there have been significant advances in automatic solutions [27, 29].

Explicit approaches require the programmer to manage all of the details of parallel programming. This is an advantage in that the programmer has total control of the program and can tune the program to the particular machine. Another advantage is that explicit techniques are an easy add-on to existing programming languages. They are often implemented as libraries, e.g., send and receive functions in a library, or shared memory and semaphores in a library. The downside of explicit approaches is that the programmer has total control of the program; including the opportunity to make synchronization errors leading to non-deterministic deadlock or non-deterministic program behavior, and the opportunity to do a very poor job of decomposing and sched-

uling the problem. Examples of explicit systems include [2,6,35].

There are systems that operate in the middle of the spectrum. By doing so they attempt to capture the benefits of both explicit and implicit approaches. Mentat, Linda [8], Fortran D [14], HPF Fortran [25], and Dataparallel C [30] are just a few of these systems.

Another dimension of the solution space is the programming language style, e.g., object-oriented versus functional. There are a variety of object-oriented parallel processing systems. Examples include Charm++ [22],CC++ [9], pC++ [4], ESP [32], and Presto [3]. Mentat differs from systems such as [2,3] (shared memory object-based systems) in its ability to easily execute on both shared memory MIMD and distributed memory MIMD architectures, as well as hybrids. pC++ [4] and Paragon [11] on the other hand are data-parallel derivatives of C++. Mentat accommodates both functional and data-parallelism, often within the same program. Mentat differs from other distributed object based systems and languages [10] in our objectives, we strive for performance via parallelism rather than distributed execution.

Applications portability across parallel architectures is an objective of many projects. Examples include PVM [35], Linda [8], the Argonne P4 macros [6], and Fortran D [14]. 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 [6,8,35] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [14,24,25,30] share our philosophy of providing a higher level 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.

5. Retrospective

In the four years that the MPL compiler has been operational we have developed several applications and learned much about the Mentat approach. The results are not all good. Like many other parallel processing systems Mentat performs well on some applications and not so well on others. The primary factors that influence performance are application granularity and application load balance. Mentat performs poorly when application granularity is small or when the application has load imbalances that Mentat cannot correct. This is not unexpected. There is not much that can be done about the granularity restrictions, overheads can be reduced to lower the minimum effective grain size. However there is a limit on how low we can drive the overhead. Similarly, the underlying communication system of MPP's favors large grain computations. That is unlikely to change. With respect to load imbalance, there is room for significant improvement in the areas of dynamic scheduling of objects, and dynamic re-distribution of data parallel objects. We are working on both.

On the plus side we have learned that the use of the object-oriented paradigm combined with compiler-based parallelism detection and management can be performance competitive with hand-coded implementations using send and receive. This is significant because it means that the future of parallel processing is not limited to send and receive, and that the benefits of the object-oriented paradigm can be realized in high-performance parallel environments.

6. The Future - Legion

Technology continues its inexorable march forward. When the Mentat project was first conceived in the mid 1980's high-end wide-area network bandwidth was 56Kb/sec. To realize the bandwidth necessary for parallel computation required either local area networks, or more realistically, tightly coupled MIMD architectures. Today 155Mb/sec networks are available, and networks up to 2.4Gb/sec are expected to be widely available in just a few years. The availability of so much wide-area bandwidth presents the opportunity to apply parallel processing technology to the construction of large-scale, wide-area, metasystems.

The Legion project at the University of Virginia is an attempt to provide system services that provide the illusion of a single virtual machine to users, a virtual machine that provides *both* improved response time via parallel execution and greater throughput 15. Legion is targeted toward large wide-area assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation based parallel processing tools such as fault-tolerance, wide area network support, heterogeneity, the lack of a single file name space, protection and security, as well as providing efficient scheduling and resource management. At the same time Legion provides the parallel processing, object-interoperability, task scheduling, security, and file system facilities not usually found in job-based load balancing systems.

Legion builds extensively on our experience with Mentat. We are leveraging the compiler and run-time system technology developed, the applications base, and our experience with solving real problems. Early results using Legion both in a campus-wide environment and on the I-Way testbed at Supercomputing '95 in San Diego have encouraged us to forge ahead with system development.

Acknowledgments.

I would like to thank Bill Pearson of the biochemistry department for introducing me to sequence comparison, and Robert Ferraro of the Jet Propulsion Lab for introducing me to finite elements. I would also like to thank all of the members of the Mentat team. Staff members are Mark Hyett and Lindsey Faunt. The students who have worked on various components are Tony Chang, Gorrell Cheek, Adam Ferrari, Roger Harper, John Karpovich, Mike Lewis, Ed Loyot, Laurie MacCallum, Dave Mack, Mark Morgan, Padmini Narayan, Anh Nguyen-Tuong, Sherry Smoot, Virgillo Vivas, Jon Weissman, and Emily West. As noted portions of this paper, in particular the performance data, have appeared elsewhere [19,23,26,34,36]. Portions of this research were performed using the Intel Gamma operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access was provided by the Jet Propulsion Laboratory.

7. References

1 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.

2 B. Beck, "Shared Memory Parallel Programming in C++," *IEEE Software*, 7(4) pp. 38-48, July, 1990.

3 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, 1988.

4 F. Bodin, et. al., "Distributed pC++: Basic Ideas for an Object Parallel Language," *Proceedings Object-Ori-*

ented Numerics Conference, pp. 1-24,Sunriver, Oregon, April 25-27, 1993.

5 S. H. Bokhari, "The Linux Operating System,", *IEEE Computer*, vol. 28, no. 8, pp. 74-79, August, 1995.

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

7 J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, pp. 111-120, vol. 16, no. 2, Feb., 1990.

8 N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April, 1989.

9 K. M. Chandy, and C. Kesselman, "CC++: A Declarative Concurrent Object Oriented Programming Notation,", Technical Report CS-TR-92-01, Department of Computer Science, California Institute of Technology, 1993.

10 R. Chin and S. Chanson, "Distributed Object-Based Programming Systems," *ACM Computing Surveys*, pp. 91-127, vol. 23, no. 1, March., 1991.

11 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.

12 J. Dongarra, R. Pozzo, and D. W. Walker, "An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures," *Proceedings Object-Oriented Numerics Conference*, pp. 257-264, Sunriver, Oregon, April 25-27, 1993.

13 R. D. Ferraro, "Solving Partial Differential Equations for Electromagnetic Problems on Coarse-Grained Concurrent Computers," *Pier 7*, T. Cuik and J. Patterson, Eds., pp. 111-155, EMW Publishing, Cambridge, MA, 1993.

14 G. C. Fox, et al., "Fortran D Language Specifications," Technical Report SCCS 42c, NPAC, Syracuse University, Syracuse, NY.

15 A. S. Grimshaw, W. A. Wulf, J. C. French, A.C. Weaver, and Paul Reynolds Jr. "Legion: The Next Logical Step Toward a Nationwide Virtual Computer," Computer Science Technical Report, University of Virginia, CS 94-21, June, 1994. See our web page at *http://www.cs.virginia.edu/~legion* for all technical reports.

16 A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May, 1993.

17 A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 34-47, May, 1993.

18 A. S. Grimshaw, et. al., "The Mentat Users Manual," available on the world wide web, *http://www.cs.virginia.edu/~mentat/*, 1994.

19 A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, July, 1993.

20 A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," *to appear ACM Transactions on Computer Systems*.

21 D. A. H. Jacobs, *Sparse Matrices and their Uses*, Academic Press, London, 1982.

22 L. V. Kale, and S. Krishnan, "CHARM++: A Portable Concurrent Object-Oriented System Based on C++", *Proceedings of OOPSLA '93*, pp. 91-108, Washington DC, 1993.

23 J.F. Karpovich, M. Judd, W.T. Strayer, and A.S. Grimshaw, "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of the Second Symposium on High-Performance Distributed Computing*, pp. 34-41, Spokane, WA, July, 1993.

24 J. R. Larus, B. Richards, and G. Viswanathan, "C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language," UW Technical Report #1126, Computer Sciences Department, Univerrsity of Wisconsin, November 1992.

25 D. B. Loveman, "High Performance Fortran,"*IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25-42, February, 1993.

26 L. MacCallum and A.S. Grimshaw, "A Parallel Object-Oriented Linear Algebra Library," *Proceedings*

Object-Oriented Numerics Conference, pp. 233-249, Sunriver, Oregon, April 25-27, 1994.

27 J. McGraw, et. al., “Sisal: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2,” Lawrence Livermore National Laboratory Manual M-146, Lawrence Livermore National Laboratory, Livermore, CA, 1984.

28 W. R. Pearson and D. Lipman, “Improved tools for biological sequence analysis”, *Proc. Natl. Acad. Sci. USA*, 85, pp. 2444-2448, 1988.

29 C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

30 M. J. Quinn and P. J. Hatcher, “Data-Parallel Programming on Multicomputers,” *IEEE Software*, pp. 69-76, September 1990.

31 Rogue Wave Software, “Math.h++, Matrix.h++, and Linpack.h++,” C++ based math libraries from Rogue Wave Software, Corvallis, OR.

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

33 T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences”, *J. Mol. Biol.*, 147, pp. 195-197, 1981.

34 S. Srinivasan and J.H. Aylor, “Digital Circuit Testing on a Network of Workstations,” *Proceedings of the International Conference on Parallel Processing*, August, 1994.

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

36 J. B. Weissman, A. S. Grimshaw, and R. Ferraro, ‘‘Parallel Object-Oriented Computation Applied to a Finite Element Problem,’’ *Scientific Computing*, vol. 2, no. 4, pp. 133-144, Feb., 1994.