

BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-94-M2

“Explicit Versus Implicit Remote Procedure Call Based
Parallel Programming Languages: An Analysis”

by

Rajesh Radhakrishnan

**Explicit Versus Implicit
Remote Procedure Call Based
Parallel Programming Languages:
An Analysis**

Rajesh Radhakrishnan

**Department Of Computer Science
Brown University**

**Submitted in partial fulfillment of the requirements for
the degree of Master of Science in the Department of
Computer Science at Brown University**

December 1993.

A handwritten signature in black ink, appearing to read 'F. P. Preparata', is written over a horizontal line.

Professor Franco P. Preparata

Advisor

Explicit versus Implicit Remote Procedure Call Based Parallel Languages: An Analysis

Rajesh Radhakrishnan

rjr@cs.brown.edu

Dept. of Computer Science

Brown University

Box 1910,

Providence, RI 02912.

Abstract

We begin by presenting a survey of C/C++ type parallel languages. The survey indicates that there is no emerging standard C/C++ type parallel language. One of the fundamental reasons for this is that languages take different approaches to the use of Remote Procedure Calls (implicit or explicit or a combination) to express remote invocation of tasks. We analyze the pure Remote Procedure Call (RPC) paradigm for a parallel language by using a language we developed for this purpose. The analysis reveals some of the advantages and disadvantages of the RPC paradigm. Our work is novel in that we suggest an extension to the RPC paradigm which improves its efficiency while yet retaining most of the transparency (from the message passing system) it provides the user. We present a few algorithms and their performance on a network of sparcstations. based on which we conclude that given the state of parallel compiler technology today, the explicit RPC paradigm is one of the best modes to give the user enough control to write efficient programs. We also examine the disadvantages of the RPC paradigm, notably the fact that porting programs to other architectures will result in a loss of performance.

1. Introduction

Over the last few decades, sequential machines have steadily improved in terms of speed, size of chips etc. However, it is becoming increasingly clear that this steady increase in performance is likely to taper off as time passes [1]. The natural place to look for continued improvement in performance is parallel machines.

While parallel machines have been talked about for a very long time, it is only recently that high performance parallel machines have become technologically feasible. The field of parallel algorithms has on the other hand been active for a very long time. It would therefore seem only natural that the moment parallel machines became available that they be used widely in solving problems. This has unfortunately not been the case.

The reason for this has been the lack of standard parallel languages and operating systems. One of the most important reasons for the lack of progress in these fields is the absence of a model for a parallel machine. For the last few decades the model for sequential computation has been the Von Neuman machine. This has ensured that compiler designers, architecture designers, systems people and algorithm designers can work almost independently (of course certain machine specific features necessitate closer working, but the large body of work has been produced in this fashion).

Parallel Algorithm designers have either adopted the PRAM (the Parallel Random Access Memory) model or a network model. The PRAM model assumes uniform memory access across the memory of the parallel machine. This model is not realizable in practice. Using network models, algorithm designers have proposed various algorithms for specific interconnections (like the Mesh, the hypercube, the tree etc.). The single biggest problem with the assumption of specific interconnections is the issue of portability. An optimal algorithm meant for a specific interconnection may perform badly on other interconnections.

Parallel machine designers have developed shared and distributed memory machines over the last few years. The languages provided on them so far have been sequential languages with access to a parallel library in order to perform communication between various processors. Users program in an SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) fashion on these machines.

Current experience with parallel programming reveals the following problems:

- Parallel algorithms are usually more difficult to understand and implement than their sequential counterparts.
- It is in general difficult to program in a manner which assures deterministic behavior and efficiency. Tools which help to eliminate or at least determine the existence of data races[2] (the causes of non determinacy) are only now becoming available.
- In order to produce efficient code the user has to fine tune his algorithm to the specific machine in terms of packet size, communication bottlenecks etc.

A Parallel language must make the user's task of programming easier. Designing a good parallel language is a difficult task as:

- Efficient algorithms are usually machine dependent and yet the code must be portable.
- The algorithm must be easy to write implying that the user should not be encumbered with low level details and yet the code produced must be efficient, i.e while the actual machine model is difficult to program in, the language must provide an abstraction which should make programming easy and yet the abstraction should be at a level that the user has enough control to write efficient code

Many attempts have been made at designing parallel languages. In the Fortran community HPPF[3] (High Performance Fortran) has already been accepted as a standard. However, in the C and C++ community little common ground has been established among the compiler designers and we appear to be far from reaching a standard. The essential reason why the Fortran community has managed to reach a standard while the C and C++ communities have not is two fold.

- Pointers are hard to implement in a transparent, efficient fashion on a parallel machine, and use of pointers is one of

the fundamental differences between C and Fortran.

- While Object oriented parallel languages seem to be an elegant way of expressing parallelism there is little consensus on the meaning of parallel object. Issues like inheritance etc. also need to be resolved.

The organization of this thesis is as follows. In section 2 we present a survey of some C and C++ type languages. The survey examines a large number of paradigms of parallel computation. The survey seems to indicate that one of the fundamental differences in paradigms is that some languages allow the user to express remote access of data and remote invocation of tasks through the use of explicit remote procedure calls (RPCs) while in other cases the fetching of data and performing of tasks is implicitly done. While the former method gives the user greater control, the latter is an easier paradigm to program in. The main emphasis of this thesis is an examination of the paradigm of remote procedure call parallel computation.

In order to explore this paradigm, we developed an object oriented parallel language based on remote procedure calls (RPC) [4]. The language is implemented using PVM[5] (parallel virtual machine) as a base. A description of this language and some implementation details are provided in section 3.

In section 4 we present performance results of programs written in this language. We motivate the need to extend the paradigm and also use the example programs to point out the advantages and disadvantages of the RPC paradigm. We conclude giving directions for future work.

2. A Survey of C/C++ type Parallel Languages.

2.1 Introduction

Over the last few years a large number of parallel languages have been developed. In this chapter we furnish a description of the more prominent C/C++ type parallel languages. Given the sheer variety of parallel languages, almost all languages have interesting features. Rather than dwell on these, we have attempted to categorize these languages under various headings. This categorization reveals current trends in this field. We also try to analyze the various paradigms of parallel programming that these languages provide.

2.2 Categories

We provide below a brief description of the various categories under which we evaluated the languages.

2.2.1 Target Architecture

- **Coarse/medium/fine grain machines:** We make this classification based on how long it takes to communicate between adjacent processors of the parallel machine. Coarse grain machines have a high communication time, medium grain machines have a moderate delay in communication and in fine grain machines, communication delay is small. A network of sparcstations would be considered a coarse grain parallel machine. iPSC/860 would be a medium grain machine and Motorola's *T would be a fine grain machine. Languages could be designed for a specific architecture or they could be designed for the entire spectrum of architectures. Clearly the latter is an extremely difficult task. An algorithm for a fine grain machine may use too much communication and therefore be infeasible

for a coarse grain machine. Similarly an algorithm for a coarse grain machine may be based on a computationally inefficient scheme so as to avoid excessive communication and this algorithm may not be the most efficient on a fine grain machine. Hence a language which is not specific to a type of architecture must allow the user to define the algorithm at a high enough level of abstraction, in order to assure efficient behavior on all architectures.

- **Distributed/Shared memory machines:** This classification is made based on the access time for remote data. In a shared memory machine, the access time for non-local data is not very high compared to that of local data. Hence it is feasible to have global data in a shared memory machine. Distributed memory machines have a large access time for non-local data and therefore it is not feasible to have large amounts of global data. In some sense a shared memory machine is inherently fine grained. Here again there is distinct difference in the kind of algorithms that need to be developed. While a shared memory machine algorithm may rely heavily on shared data structures, a distributed memory machine algorithm will perform better when there is less global data (even if that means that the resulting algorithm is computationally more complex). Here again one sees the difficulty of developing a language which is equally applicable to both kinds of machines, and assures portability of code across them.

2.2.2 Model presented to the User:

A distinction must be made between the actual machine and the model the language presents to the user. Even if the machine is a distributed memory machine, for ease of programming, the language may present a shared memory model to the user. It appears unreasonable at first glance to present the user with a distributed memory model especially if s/he is working on a shared memory machine. However, the distributed memory model enforces a discipline on the user. It ensures that the user is made aware of communication costs and is persuaded to take greater advantage of locality. Further a distributed memory algorithm will perform fairly well on a shared memory machine whereas the reverse is not quite true due to the fact that a large number of remote data accesses may be performed in an efficient shared memory algorithm.

- **Implicit/Explicit data access and task invocation:** A language could allow the user to implicitly specify remote data access and task invocation or make these explicit constructs. In the implicit case, the model presented is of a single address space and if the user makes a procedure call on an object which is not local to the current machine, a

remote procedure call is made. The compiler and the run time system take care of fetching remote data. While this paradigm is easy to program in, it does not explicitly show the user how costly the program s/he writes is and this may result in the writing of extremely inefficient code. An implicitly parallel language may or may not allow the user to explicitly partition (allocate) data. If the user has control on data partitioning ¹, it will be easier for him/her to write more efficient code. If on the other hand, the compiler controls data partitioning, it may be possible (at a later time) with more sophisticated compilers to examine the entire program and decide on the best possible partitioning, and this would result in more efficient code. The parallel language community is at odds over the issue of whether the user should have control over data partitioning or whether the compiler should.

Explicitly parallel languages are usually of the RPC (remote procedure call) type. This paradigm enforces the user to specify a procedure call on a remote processor in order to perform a remote task. The model may allow the user to specify data accesses only through RPCs or may allow explicit remote accesses. The following example highlights the difference between implicit and explicit use of RPCs:

Implicit:

```
x = y;          /* y is a remote value and the compiler fetches it */  
A->foo();      /* A is a remote object, foo is a method of that object and the procedure call is per-  
                formed implicitly as an RPC. */
```

Explicit:

```
x = y;[2]      /* get the value of y from logical processor 2 */  
A->foo();[3]   /* Perform foo on object A on processor 3; explicit RPC*/
```

Some RPC based languages do not allow the user to perform remote data accesses directly. The user would have to define accessor functions to access these values. For example, in the instance shown above, the statement “x = y;[2]” would have to be written as “x = bar();[2]” where “bar” is function which returns y.

There are pros and cons for both schemes (implicit and explicit parallel languages) and we will talk about them in greater length at the end of this section. The focus of this thesis is to examine these pros and cons.

1. Data partitioning refers to a data allocation scheme by which variables are mapped to memory (or memories of different processors).

2.2.3 Source of Parallelism

• **Loop Parallelism:** A language exploits loop parallelism when different iterations of a loop are executed in parallel on different processors. Loop parallelism requires temporal independence of the events which are occurring in parallel. The parallelism may be implicit in that the user writes a normal sequential loop and the compiler parallelizes it, or the language may provide constructs which allow the user to express this kind of parallelism. For example.,

1.

```
for(i=0;i<10;i++)  
    a +=bar(i);    /* If the compiler attempts to parallelize this loop and call mul-  
                  tiple invocations of bar in parallel, loop parallelism is being  
                  exploited; the user can explicitly use loop parallelism as well.  
                  A sophisticated compiler may even use a parallel prefix kind  
                  of operation to add the values of different iterations. */
```
2.

```
dopar i = 0 to 10  
    bar(i);        /* Explicit use of loop parallelism. Since the iterations do not  
                  deal with the similar data, this is referred to as loop rather than  
                  data parallelism. */
```

• **Data Parallelism:** When operations on arrays and similar data structures are performed in parallel in a SPMD style, data parallelism is exploited (i.e spatially independent similar objects). Data parallelism is a generalization of loop parallelism and it usually tends to be implicit.,

```
A->foo();    /* If A is a collection of objects and foo is a method of these objects and this state-  
            ment causes all objects which A represents to execute foo in parallel, then this  
            exploits data parallelism. */
```

If the language can express data parallelism, either the user or the compiler must perform some kind of data partitioning. The trend today appears to be that the compiler allows the user to specify simple patterns of data partitioning (block, cyclic etc.). Though as mentioned earlier, there is debate of whether the compiler, or the user (or both) should have control over data partitioning.

• **Functional Parallelism:** All sources of parallelism, including running several function or procedure activations in parallel fall under the category of functional parallelism. Functional parallelism subsumes data parallelism; however,

compilers which exploit functional parallelism do not focus as strongly on optimizing the exploitation of data parallelism within an application. Functional parallelism is probably the best way to describe what most parallel object oriented languages provide. If objects are distributed across processors, and they are all similar then we are exploiting data parallelism, whereas if the objects are different we are exploiting functional parallelism. Performing the same operation on a bunch of similar objects in parallel is akin to implicit loop parallelism. Hence the object oriented model very elegantly expresses the different sources of parallelism. A classical problem which fits into functional parallelism is producer consumer problem. Typically it involves one thread (the producer) adding items to a finite buffer and another (the consumer) taking items out of the buffer and doing some processing on them. Since the buffer is of finite length, synchronization between the two threads is required. The following is an example of functional parallelism in the producer - consumer problem.

```

pardo                                /* Do in parallel */
    producer();                       /* Call the producer and do not block */
    consumer();                       /* Call the consumer and do not block */
end pardo                             /* wait for both producer and consumer to complete */

```

Consider the following example of the use of object oriented parallelism in exploiting data parallelism and functional parallelism:

```

pardo
    A[0]->foo();
    A[1]->foo(); /* if the objects pointed to by A[0] and A[1] are similar, this block
                  of code is exploiting data parallelism, however if the objects are
                  substantially different, functional parallelism would be a better
                  way to describe it. */
end pardo

```

2.2.4 Determinacy:

Parallel computations can be non-deterministic i.e two executions of the same program may proceed in different ways and even produce different results. In a message passing parallel program, this is essentially caused by racing messages

(i.e those which are not ordered by the Lamport Happens Before relation[6]). Due to varying message latencies, different executions of the same program may cause messages to arrive in different orders thus changing the execution itself. Non determinacy is a problem, especially while debugging programs, as it is not possible to guarantee that a bug that manifests itself in one execution of a program will do so in another.

A distinction must be drawn between non deterministic executions and non deterministic algorithms. Certain algorithms are non deterministic in that they use a random number generator (or some other source of randomness) to make some decisions (e.g. randomized algorithms). What we are concerned with however is non-deterministic executions resulting from operations which are not synchronized (e.g. racing messages).

Deterministic programs can be written in any language; however, we categorize a language as being deterministic if it requires the introduction of specific non-deterministic operations to create a non-deterministic program (e.g a call to a random number generator). Determinacy is desirable in that it is easier to write and debug a deterministic program. Hence the way in which a language tackles this issue is an important categorization of the language. Some languages provide the user with synchronization primitives (thus putting the responsibility on the user's shoulders), whereas others ensure determinacy through compile time analysis and run time checks. Still others compromise by assuring determinacy on some data structures and expecting the user to synchronize operations on others.

The flip side is that by assuring determinacy, the language could be preventing the user from expressing some naturally non-deterministic phenomena. For example a producer - consumer program with the producer and consumer competing for a lock is non-deterministic (in that either thread could get the lock first) but the overall behavior is deterministic. Further there are associative operations which while non-deterministic in their execution are deterministic in their results (e.g if you add n numbers which are on different processors, the order of the addition is unimportant and you could typically just add the values you receive in messages, without worrying about the order in which the messages were sent). Non-determinacy inherently stems from races and depending on the kind of race, the behavior is either desirable or not[7]. Hence if a language does not allow the user to express desirable non-determinacy, it is sacrificing some performance. However, deterministic programs are easier to analyze and the compiler may be able to carry out more sophisticated optimizations on these which may not be possible on a non-deterministic program.

Hence a language could be

- Focused at a particular kind of architecture or meant for the entire spectrum.
- Implicitly or explicitly (e.g RPC) parallel in the model it presents the user.
- Loop/Data/Functionally parallel (or all three).
- Deterministic or non-deterministic.

Further a language could be object oriented or data parallel, it could be SPMD, SIMD etc. There are a large number of categorizations and we have described what appear to be the significant dividing features in the field of parallel languages today.

The C/C++ parallel language community seems pretty far from reaching a consensus on a standard currently. Some of the factors which are proving to be obstacles are

- Determinacy versus efficiency.
- Whether the user or the compiler exploits parallelism.
- Whether the user or the compiler defines the data partition.
- How are pointers tackled?
- What is parallel object?
- Implicit versus explicit parallel programming.

In this report we attempt to answer the last question. More precisely we look at the RPC paradigm of parallel programming and examine how efficient it is and how easy it is to express a parallel algorithm in this paradigm.

First however we look at some of the more popular parallel languages and what categories they fall into.

2.3. The Languages

2.3.1 C** (James R. Larus, University of Wisconsin at Madison)

1. The Model (presented to the user)

C**[8, 9] is a large grain, object oriented, data parallel language. Parallelism is explicit in that the user declares certain function to be parallel. The language does not assure determinacy, however most user programs are likely to be deterministic (look below at Semantic properties). The user does not have to deal with communication issues.

2. Target architecture

C** is targeted towards coarse grain machines. The language is similar to C++ in many respects. It provides most of the elegant features of C++ like inheritance, virtual functions etc. and also provides a few extra features which are particularly useful.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

In C**, Aggregate objects are the basis for parallelism. Aggregate objects in C** are similar to C++ classes in that they have data members, member functions, private and public members etc. They differ in four fundamental respects from C++ arrays of objects

1. An Aggregate declaration specifies the type of the collection, not the individual elements. (In C++ one could declare a matrix as a class having a single floating point value and then making a two dimensional array of that class, whereas in C**, the aggregate would be the two dimensional array of floating point numbers.)
2. Aggregate member functions operate on the entire collections of elements, not on individual elements.
3. Elements in an Aggregate can be operated in parallel.
4. Aggregates can be sliced (look at Syntactic properties below).

Parallelism is achieved by executing parallel member functions of aggregates. The function is a large grain data par-

allel operator which is applied atomically and in parallel to each element in the Aggregate. The semantics of the call are

- Read all referenced locations into a purely local copy.
- Compute using local copies.
- Write all modified variables back so that the global value is changed.

The user can, within a parallel function, find out using the pseudo variable #1, #2 etc., which array member the current operation is on.

A parallel function can return a scalar (this is typically done by “reducing” (a parallel prefix type operation) the values returned by individual invocations of the parallel function) or it can return an aggregate.

4. Data Partitioning

Data partitioning is handled by the compiler.

5. Parallel Constructs

The parallel member functions of aggregates is the only parallel construct. As mentioned earlier the semantics of the call are different from the C++ semantics. It is not, for example, possible to perform iterations within a parallel function, changing a local value and expecting other processors to read the new value any time they access it before the completion of the function. The reason for this is that all required remote values are initially copied and maintained in a purely local environment. While this does limit the power of the parallel function, it is possible to overcome this by calling the function itself in a loop, from outside.

6. Synchronization issues

The user does not have to deal with any synchronization issues.

7. Portability

C** seems suited to large and medium grain machines and the code produced by it will not perform efficiently on fine grain machines.

8. Semantic properties.

There are a few semantic differences between C** and C++. Firstly the semantics of a parallel function call are not the same as a method of C++ as discussed before. Other semantic differences include

- If a parallel function returns an aggregate, that aggregate's constructor never gets called whereas in C++ it does.
- C** overloads the += and similar operators for reductions when the left operand is scalar and the right a parallel operation.

The address space is essentially global. During the execution of a parallel function, the values of the referenced variables can be different on different processors, depending on whether a particular processor modified that variable or not (remember that each processor gets a local copy of all variables it references) - but, one can think of this more as local variables rather than a non global space.

The language does not assure determinacy. The parallel functions can cause non-deterministic behavior. If two processors modify the same variable during a parallel function call, the value the variable will have at the end of the parallel function's execution is not determinate.

9. Syntax.

C** has a few syntactic extensions to C++. Instead of C++ classes, one deals with aggregates (which have properties like inheritance, virtual functions etc. just like classes).

Another important extension to the syntax of C++ is Slices. A slice selects a subset of an aggregate along an axis introduced by a subscript dimension. A slice is not a copy, it shares all selected elements with the full aggregate. For example, if one makes the rows and columns of a matrix to be slices, it is then possible to write data parallel operations on those. This makes the language more expressive and the style of programming more elegant and readable. The dot product operation, needed for matrix multiplication, for example is easily writable in terms of row and column slices

of the matrix aggregate.

10. Ease of Programming.

C** is designed to make the user's task of programming easier. The capabilities provided through aggregates and slices make parallelism easy to express.

11. Efficiency

What C** gives up, in making the user's task of programming much simpler, is efficiency. In the performance results given the code written in C** performed much worse than the hand coded parallel program. The reasons for this are

- The user has no control over data partitioning and hence locality cannot be exploited.
- The overhead for the parallel functions is likely to be high compared to the amount of computation they perform.

Since the parallel functions cannot see changes in the outside world once invoked, it may be necessary to call them within a loop rather than have a loop within their body. Further every time a parallel function is called, the values of the variables it references need to be sent to it, and after the execution, the variables it modifies need to be written back. Hence the overhead of invoking a parallel function is likely to be high compared to the amount of computation performed by the functions.

12. Interesting features

1. The language does make the task of programming easy for the user.

13. Problems with the language which make it less suitable to use.

1. Efficiency is a major concern. In the figures provided, code generated by the C** compiler performed four times worse than the hand coded one.
2. The semantics of the parallel function calls are non-intuitive and make programming a little harder.

2.3.2 Jade (M. S. Lam, Stanford)

1. The Model (presented to the user)

The model Jade[8, 10, 11] provides the user with is of a shared memory machine. There is no explicit message passing and the user does not have to deal with synchronization. Parallelism is implicit.

2. Target architecture

The current implementation of Jade is for coarse and medium grain machines. The philosophy seems extendable to fine grain machines, but the overheads of the run time system in the current implementation are inefficient for fine grain systems.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

The user writes sequential programs and annotates call-sites of procedures with statements describing how the procedure accesses data (i.e which “objects” does it write and read). The compiler uses this information to extract coarse grain parallelism by doing a remote procedure call when a given procedure is independent of all other currently running procedures (Two procedures are independent if and only if they don't have a write-read, read-write or write-write dependency).

A task (procedure) must wait for all tasks it depends upon to complete before it can begin execution. In order to improve the parallelism, the user can define reads and writes to be deferred. In this case, the task attempts to synchronize only just before it accesses the objects on which it performs the deferred accesses, and not before it begins its execution. Further the user can also indicate to the compiler within the body of a procedure that a task will no longer access a particular object. This further lessens the number of dependencies.

The developers of Jade believe that a compiler is better than the user at exploiting fine grain parallelism and that the user is better at exploiting coarse grain parallelism. Further if the user writes an explicitly parallel program, it becomes

extremely difficult for the compiler to exploit fine-grain parallelism. This indeed does seem true in practice.

Hence by giving the user these constructs, the language allows the user to indicate to the compiler, dependencies at a very coarse level (task-level) while yet retaining sequential semantics and thus making it possible for the compiler to exploit fine-grain parallelism.

The current implementation of the compiler does not attempt to extract fine grain parallelism from user code.

4. Data Partitioning

The user has no control over data partitioning.

Any data (object) which is shared across multiple tasks has to be defined as shared. To minimize data dependency between tasks, the user must define objects in as fine grain a fashion as possible. For example if two tasks access two different parts of an array, they are independent of each other, but if the user defines the whole array to be a single shared structure, the compiler will assume that there the tasks are dependent.

5. Parallel Constructs

There are no special parallel constructs - all parallelism is implicit.

6. Synchronization issues

The user does not have to worry about synchronization issues as the compiler takes care of preserving sequential semantics.

7. Portability

The code is easily portable and will work efficiently without any rewriting on medium to coarse grain machines. The current implementation of the Jade compiler will make code ported to a fine grain machine inefficient - but there is nothing in the philosophy of the language which prevents future implementations from overcoming this problem.

However it must be pointed out that Jade compilers will need a lot of run time analysis like reporting access violations

which may prove to be a pretty large overhead when really small threads are spawned in a fine grain machine.

Jade can also be used for heterogeneous multiprocessor systems.

8. Semantic properties.

The language assures determinacy. The style of programming is sequential and there is a global address space.

9. Syntax.

Jade is an extension of C. It preserves the semantics of C.

10. Ease of Programming.

The language seems to be reasonably simple to program in. The authors describe numerous examples in which they take existing C programs, refine the data structures and introduce the required annotations to produce reasonably efficient Jade code.

The refining of data structures of an existing program may prove tedious, but it seems to be one of the simpler ways to convert sequential code to parallel code.

11. Efficiency

The performance results provided by the authors seem to indicate that Jade performs well on medium grain and coarse grain machines.

12. Interesting features

1. The user deals with no synchronization issues and the compiler takes care of all parallelism.
2. The current implementation works well for coarse and medium grain machine and the philosophy of the language does seem extendable to fine grain machines.

13. Problems with the language which make it less suitable to use.

1. The efficiency is highly dependent on the way the user defines data structures and also on how well s/he uses the constructs for deferred accesses.
2. The user has no control on data partitioning and hence the language makes no use of locality.

2.3.3 Mentat (Andrew Grimshaw, University of Virginia)

1. The Model (presented to the user)

Mentat[8, 12, 13] is an Object Oriented Parallel language. It is an extension of C++. The user does not have to deal with a machine model. The user declares some objects to be distributed(default) and some to be shared (globally available and operations on which are synchronized).

The user programs in a sequential manner and the compiler extracts the parallelism and hence determinism is assured.

2. Target architecture

The language works well for machines which are medium to coarse grain. It has been implemented for iPSC/2, iPSC/860(Gamma), SGI Iris, Sun 3 network, Sun 4 (Sparc) network, (In progress: Paragon, Cm-5, RS/6000). The language works well for a network of workstations too, and is thus suitable for a large segment of the market.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

The parallelism is implicit. The user can define Mentat objects which indicates to the compiler that the methods of this class are worth parallelizing. (In medium to coarse grain, the overhead of parallelizing being high, the methods of some classes are not worth parallelizing).

The compiler uses this information to make Remote Procedure Calls (RPCs) for the methods of Mentat objects and making the calls for non Mentat objects locally.

The compiler detects data and control dependencies between Mentat class instances and hence automatically provides data and functional parallelism. There is no construct for loop parallelism.

4. Data Partitioning

Data Partitioning is essentially the compiler's responsibility. The user may however, at the time of creation give directives to the compiler which cause the object to be instantiated by a particular processor, based on the location of other objects or a particular processor number. The user can also inform the compiler that a particular object's methods have a high a computation to communication ratio (for load balancing). This allows the user to have some control over data partitioning.

5. Parallel Constructs

The user can define 3 types of objects

1. Default objects retain no state information and their methods are not worth parallelizing as the amount of computation is negligible.
2. Mentat Regular retain no state information, but have computationally intensive methods and which are hence worth parallelizing.
3. Mentat Persistent retain state information. All operations on these are synchronized and the compiler attempts to parallelize the methods of these classes.

Objects of type 1 and 2 are maintained in the local memory of processors and hence their scope is local i.e if two processors have objects with the same name of type 2 and 3, they are both looking at their local instances only.

Hence parallelism is achieved by

1. Non blocking RPCs.
2. Multiple creations of objects of type 1 and 2 (there is no synchronization needed for these objects).

6. Synchronization issues

Any Mentat persistent object's methods and operations are performed synchronously (i.e the operations are performed in the user specified order and exactly one at a time, the RPC however is non-blocking). The responsibility for synchronization thus rests with the compiler making the users task much simpler.

7. Portability

The code developed in Mentat is easily portable. The Mentat design can be presented in 3 layers

1. The user's view - the language and the compiler which does part of the dependence analysis.
2. The portable run time system
3. The underlying computational model (the macro data flow model, a medium grain, data driven computational model that represents programs as data dependence graphs, constructed dynamically at run time).

8. Semantic properties.

The language assures determinacy, the style of coding is sequential and the address space is not global.

9. Syntax.

The language is an extension of C++. Semantically however, there are 4 differences

1. Mentat member functions are always call by value.
2. Mentat member function calls are non-blocking whenever the data dependencies permit.
3. Each invocation of a regular Mentat object instantiates a new object to service the request.
4. There is no global address space.

10. Ease of Programming.

Programming in this paradigm is fairly straightforward. The user has to design the application keeping in mind the properties of Mentat regular and persistent objects. Once these design decisions have been made, the coding task looks reasonably easy.

11. Efficiency

Efficiency is largely dependent on the compiler as all parallelization is performed by the compiler. The user can perform some data partitioning and be helpful in deciding which classes' methods to parallelize etc.

It is difficult to look at a piece of code and analyze its complexity as the compiler does data flow analysis and some optimizations.

12. Interesting features

1. Mentat keeps the users job of programming reasonably simple without having him/her worry about synchronization or communication issues. Users can further simplify matters for themselves by letting the compiler take care of data partitioning as well.
2. Portability across a fair spectrum of parallel architectures.

13. Problems with the language which make it less suitable to use.

1. Efficiency - the user is not motivated towards thinking parallel.
2. As the compiler does the parallelizing, the complexity of the program is not clear to the user and hence it is not immediately obvious whether an algorithm change will improve or degrade performance. But this is clearly a claim to be made of the whole class of languages which take the responsibility of parallelizing off the user's shoulders.
3. Some run time data dependence analysis is required because of the nature of C++ (it is not always possible at compile time to determine which objects a pointer points to) - and this will slow down code written in this language.
4. Further it is not possible to label individual methods as compute intensive (entire classes are labeled Mentat or non-Mentat).

2.3.4 pC++ (Dennis Gannon, Indiana University)

1. The Model (presented to the user)

pC++[8, 14, 15] is a data parallel, object oriented parallel language. The model presented to the user is of a distributed memory machine. The user does not have to deal with message passing and in that sense the parallelism is implicit.

2. Target architecture

The language seems designed for medium to coarse grain machines. It will work well for distributed as well as shared memory machines, though the programming model is that of a distributed memory machine.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

pC++ is inherently data parallel. Methods are applied concurrently in an SPMD style on “collections” of homogeneous objects.

pC++ does not extract functional or loop parallelism.

4. Data Partitioning

pC++ has borrowed data partitioning directives from Fortran 90 and HPF. The user defines templates which are representations of the virtual machine the user is working on. The user can use the align directive to place the “collection” of objects at a specific place on the template. The user can also do block, cyclic or whole distributions of data.

pC++ provides these directives as part of a Processor class. The partitioning is hence completely handled by the user. pC++ uses the owner compute rule (i.e the processor on which a variable is resident is responsible for all operations which modify the value of that variable).

5. Parallel Constructs

A user programs in terms of classes and collections in pC++. A class in pC++ is like a class in C++. An object is unaware of other objects in the collection. Hence when the user programs things specific to the object, s/he is likely to make them part of the class and when s/he codes things pertinent to all the objects in the collection, s/he makes them part of the collection i.e local data is stored in the class and global data in the collection. A collection, like a class has private, protected and public members. The constructor for a collection is invoked with an alignment and a template and hence a collection also has physical meaning i.e it represents the topography of the set of objects as well. There is also a special “MethodOfelement” type that members of a collection can have. Methods which are defined as “MethodOfelement” are virtual and protected. These methods when invoked, are run in parallel on all processors which are part of the alignment with which the constructor of the collection was invoked.

The “MethodOfelement” methods of a collection can be overridden by the definition in the class (which is an element of the collection). The class has information of the global world because of the “MethodOfElement” data members it inherits from the collection it is a member of.

Hence parallelism in pC++ essentially stems from the invocation of a method of a collection (which causes all elements of the collection to execute the method in parallel). One can alternatively invoke a “MethodOfElement” type method of the collection itself which again causes the code to be executed in a SPMD type fashion on all processors.

6. Synchronization issues

For element class functions and “MethodOfElement” functions, the compiler performs synchronization at the call level, i.e all processors synchronize after executing the code associated with the call. All other calls are executed asynchronously. What this means is that the user is responsible for the synchronization of various processor threads during the execution of a private or public method of a collection. These methods are executed by all processors simultaneously.

7. Portability

The pC++ user deals with a virtual machine model. It is the compilers responsibility to map this onto a real machine.

Hence the code should be portable across architectures.

pC++ has been implemented on CM-5 and the Intel Paragon.

8. Semantic properties.

pC++ does not assure determinacy. The sources of non-determinacy are the public and private methods of a collection which are executed asynchronously and in parallel.

The address space is not global. When a method is executed in parallel on different processors, one object cannot access the private data members of other objects.

9. Syntax.

pC++ is an extension of C++. The extensions made are that of collections and of alignment functions.

10. Ease of Programming.

pC++ is a reasonably simple language to program in and get fairly good performance results. What makes the language some what difficult is the fact the user has to take care of some synchronization issues.

11. Efficiency

It is possible to write reasonably efficient code in pC++. Since the compiler does not perform any transformations and the user does not have low level control, it sometimes not possible to program in the most efficient manner possible.

12. Interesting features

1. pC++ is the natural data parallel extension to object oriented programming.

2. The ideas of global and local data are elegantly encapsulated.

13. Problems with the language which make it less suitable to use.

1. The compiler does not perform sophisticated transforms and the user does not have any low level control and hence it is not always possible to program in the most efficient fashion.

2.3.5 μ C++ (P. A. Buhr, University of Waterloo)

1. The Model (presented to the user)

μ C++ [8, 16] is an extension of C++. It is an object oriented concurrent language. The user is made unaware of the underlying message passing model. All communication takes place through procedure calls.

The language does not assure determinacy.

2. Target architecture

The language has been designed for a shared memory multiprocessor. Communication in μ C++ takes the form of procedure calls which often (in μ C++) involve the movement of entire data structures. This is extremely costly in a distributed memory machine, but is a reasonable scheme for a shared memory machine.

μ C++ allows the user to program in a fashion which ensures that a large number of lightweight threads are available to the run time system to extract fine grain parallelism. However, the μ C++ compiler does not attempt to extract fine grain parallelism from the users program. The user can express coarse grain parallelism easily in the language.

μ C++ is a fairly complicated language and hence seems targeted at a sophisticated user who wants to write as efficient code as possible.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

The user programs in terms of tasks which are an expression of functional parallelism. The user also writes “co-routines”, which are used to generate lightweight threads¹, which could be used to generate fine grain parallelism. The fine grain parallelism is achieved by scheduling these threads when other threads are blocked.

The μ C++ compiler does not extract any loop parallelism. All parallelism is programmed in by the user. The compiler does not perform any sophisticated transforms. The language itself is not explicitly parallel. Parallelism is achieved through method invocations on tasks which are running in parallel.

4. Data Partitioning

The user has no control on data partitioning. Since the target machine is shared memory (and so is the user’s view), data partitioning is not too important for μ C++. The user can however migrate processes (tasks) from one processor to another.

5. Parallel Constructs

The following is the type of classes one can create in μ C++ and the kind of parallelism they generate

1. Class Objects are not owned by any processor and do not have any state associated with them. All operations on them proceed without synchronization. They are essentially run on the processor on which they are invoked. They produce light weight threads and can be used for fine grain parallelism.
2. Monitors are like class objects in that they do not retain state information and have no home processor. The operations on them, however, are synchronized. Since the μ C++ run time system attempts to extract fine grain parallelism by scheduling new threads when the currently executing one blocks, it is possible that two methods may execute concurrently on a given object and hence a user may require to synchronize between them to ensure mutual exclusion.
3. Coroutines retain state. They do not have a separate processor allocated to them and have no synchronization primitives associated with them. Hence co-routines generate lightweight threads as well.
4. Monitor coroutines are coroutines with monitors associated with them which ensures that only one method can

1. Threads which perform a task which requires very little CPU time.

execute at a given time. Since coroutines have state and no processor, they can only be accessed by the processor which creates them.

5. Tasks - are objects with state, synchronization and with a host processor. Tasks can communicate with each other through procedure calls and are globally visible.

These 5 abstractions provide the user with an elegant scheme in which to represent the program.

6. Synchronization issues

The implementors of μ C++ have tabulated eight possible kinds of objects based on whether or not an object retains state, whether or not it has a host processor and whether or not accesses to its data is synchronized. Three of these objects they reject as useless and the other five (mentioned above) they provide to the user.

The user can thus direct the compiler take care of all synchronization for an object. The user can also refine the definition of synchronized objects by defining certain methods to be non-synchronized.

However, if the user wants s/he may attempt to manage the synchronization issues himself/herself. For this event, μ C++ provides primitives which the user can use to get some control in an asynchronous environment. The user can do things like

1. define the stack size for various objects
2. Use accept commands to decide which method to execute when more than one is pending.
3. Relinquish control of the processor for use by other tasks.

and several other low level commands.

7. Portability

The language is meant for shared memory machines and performance will degrade if a compiler for the language, given its present semantics, is developed for a distributed memory machine. The language has currently been implemented on the Sequent Symmetry.

8. Semantic properties.

The language does not assure determinacy. However if a user were to use only synchronized data structures, the program produced would be deterministic.

The address space is not global in that class objects, monitors, coroutines and monitor coroutines created by a task are visible only to that task.

9. Syntax.

The language is an extension of C++. The extensions to C++ are essentially the 5 different kinds of object types.

Inheritance poses a big problem to the designers of μ C++. If a synchronized object inherits from a non-synchronized object, what are the properties of the resultant object? (especially in terms of virtual functions). As of now the language disallows inheritance from an object of a different type - but the implementors expect to define the semantics of inheritance soon.

10. Ease of Programming.

μ C++ is a difficult language to learn and use in its entirety.

11. Efficiency

It is possible to write fairly efficient code in μ C++. The language relies on the users ability to program in a fashion which allows the run time system to extract fine grain parallelism. This is a difficult task and most user programs are not likely to program in such a fashion. The efficiency of the language could suffer from this.

12. Interesting features

1. The 5 different kinds of objects provided are elegant abstractions.
2. μ C++ allows a lot of low level control which is useful for generating efficient code.

3. The abstractions of class objects and monitors produce light weight threads which are useful in generating fine grain parallelism.

13. Problems with the language which make it less suitable to use.

1. The compiler does not attempt to transform the program and extract fine grain parallelism and leaves the task to the user.

2. Providing a large number of low level commands makes the language appear more difficult and may inhibit users from using it initially.

2.3.6 CC++ (Mani Chandy, Caltech)

1. The Model (presented to the user)

The user's view is of a distributed memory machine. The language motivates the user to think about locality issues. The parallelism is explicitly programmed by the user. The language does not assure determinacy. The user does not have to deal with message passing and shared variables can be accessed directly.

CC++[8, 17] makes 6 extensions to C++ of which two constructs deal with expressing parallelism, and two are for synchronization.

2. Target architecture

The language seems suited for medium to coarse grain architectures. It is possible to use the language to express fine grain parallelism, but users would normally find it difficult to think in the low level of detail needed to exploit fine grain parallelism to its fullest.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

The parallelism achieved is essentially functional. The compiler does not extract loop parallelism. Since the language is object oriented, it has some inherent data parallelism. When the user invokes methods on different objects, he can choose to have them execute in parallel.

4. Data Partitioning

The user has some control on data partitioning. Similar to the new operator in C++ which allows the user to specify where to place the newly allocated data, the new operator in CC++, allows the user to specify which physical processor to place the data on. The control, however is limited. The user can only direct the compiler to place new objects, not relocate older ones and further the user can only direct the compiler to place a new object on the same processor as an old one, not in an adjoining processor or processor number "x", etc.

Instead of giving the compiler directives for data partitioning, the user can choose to view the machine as composed of a set of logical processors and then proceed to distribute data in any way he deems fit. He cannot however direct the compiler to make certain processors close to others etc.

5. Parallel Constructs

The parallel constructs available in CC++ are

1. **Parallel Block:** This construct is similar to a co-begin. The set of statements comprising the parallel block operate concurrently, and control passes onto the line of code after the parallel block only when all the component statements of the parallel block have finished executing. A parallel block in CC++ cannot
 - Have local variables
 - Have gotos from within the parallel block to out of it and vice versa
 - Have a component statement which executes a return.
2. **Spawn:** When a function call is annotated by a spawn, the execution of that function takes place remotely. This ensures parallelism as once the remote procedure call is made, the caller continues execution without waiting for the callee to complete execution i.e the call is non blocking.

6. Synchronization issues

CC++ provides two constructs to the user to handle synchronization issues.

1. Atomic Functions: When a user declares a method to be atomic, no other operation can occur on that object in parallel with that method. An atomic function

- Can only refer to its own data members and static variables. It cannot reference other objects' members.
- Cannot reference a “sync” (look below) value.
- Must terminate execution in a finite number of steps - this is checked for by the compiler.

2. Sync variables: A “sync” variable is written to exactly once and read from an arbitrary number of times. If an attempt is made to reference an uninitialized “sync” variable, the process doing so will suspend until the variable is initialized. Hence in CC++ we have “sync”, “const” (constant) and mutable data.

7. Portability

Code written in the language can be ported across machines with relative ease. However, since the user exploits parallelism, the code may tend to be specific to a particular architecture.

8. Semantic properties.

The language preserves the semantics of C++ completely and makes a few extensions of its own. Any valid C++ program is a valid CC++ program.

The language does not assure determinacy. CC++ is an attempt to unify the framework of writing deterministic and non-deterministic programs. The implementors believe that in a parallel language, it should be easy to write a deterministic program (and it should be easy for the user to see that the program is deterministic), but there should be non-deterministic constructs available.

When the user has a parallel block, or executes a spawn, it is the users responsibility to make sure that conflicting writes or reads do not occur. There are no run time or compile time checks to ensure that the code is deterministic

which makes it hard for the user to be sure that the code is indeed deterministic.

The address space is not global. Local pointers (look below) point to different pieces of memory on different processors.

9. Syntactic - is it an extension of C/C++ or is it more. Are the semantics any different

There are 6 extensions CC++ makes to C++, 4 of which have been discussed earlier. The last two extensions are

1. Logical Processors: This is an abstraction of a processing resource and can be used by the user

- As a mechanism to separate algorithmic concerns from resource allocation.
- as a way of expressing locality.
- to describe heterogeneous computation.

A private data member of a logical processor is local and is not accessible by other logical processors and hence the user is motivated towards improving locality of accesses. Public members are accessible globally. Hence if the user wants, he can program in an explicitly parallel fashion directly involving logical processors. The logical processor is an elegant method of expressing RPCs on other processors, and accessing data members from other processors.

2. Global pointers: CC++ deals with the problem of remote procedure calls whose parameters are pointers (a pointer on one processor may point to something totally different from a pointer with the same value on a different processor) by defining global pointers. A global pointer's value is understood across processors. A user can also define sync pointers (for sync variables), the default is local non-sync.

10. Ease of Programming.

Certain aspects of the language, like the global pointers, parallel blocks, spawn etc. make it easy for the user to program in a parallel environment. However, since the language is non-deterministic, it makes the user's task of programming difficult.

11. Efficiency

The language should perform reasonably well.

12. Interesting features

1. Logical processors are an elegant idea. Since logical processors are objects, CC++ has within the framework of C++ given the user a nice abstraction through which to view the machine. It is possible to express locality and heterogeneous computing easily using this.

13. Problems with the language which make it less suitable to use.

1. Data partitioning: The user does not have complete control over data partitioning.

2.3.7 Charm++ (L. V. Kale, University of Illinois at Urbana Champaign)

1. The Model (presented to the user)

The model presented to the user is of a distributed memory machine. Charm[8, 18] introduces “Chares” which are essentially processes. The user programs in terms of distinct processes which run independently and communicate with each other using messages.

The message passing in Charm++ is quite different from most message passing systems. Each chare the user writes has a procedure (user defined) that services received messages. The user “sends” messages essentially by invoking remote procedures. Hence, while the user is made aware of the existence of an underlying message passing system, he does not have to directly program in terms of sends and receives. The parallelism is quite explicit and the language does not assure determinacy.

2. Target architecture

Charm looks like a reasonable language for the whole spectrum of parallel machines. Data accesses are performed in a split phase¹ fashion. Further since a large number of chares (usually more than one per processor) execute concur-

rently, it is possible to exploit split-phase transactions. Hence Charm does exploit fine grain parallelism.

Coarse grain parallelism is inherent in the languages as the user programs in terms of chares which are concurrently executing threads.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

The parallelism is achieved through functional and data parallelism. Chares are essentially objects which encapsulate data parallelism. Further all function (method) calls in Charm++ are non blocking and remotely executed.

Charm does not attempt to parallelize loops.

4. Data Partitioning

Data partitioning is essential taken care of by the run time system of Charm. The run time system attempts to do efficient load balancing and gives the user a choice between a number of strategies

- Random
- Adaptive
- Central Manager
- Token

The last of which is the most efficient. The user may however override the load balancing schemes by giving the processor number on which to create a chare. The user cannot however do any sophisticated data partitioning (like the block or cyclic partitions). The compiler also does not attempt to do data partitioning based on the structure of the program.

5. Parallel Constructs

Each chare is a process which is potentially executed concurrently with all others. There are three kinds of chares

-
1. Split phase transactions are performed to hide latency. In the typical case, when a process requests a remote data fetch, it suspends and allows other processes to use the CPU. Upon the completion of the fetch, the suspended process may resume, thus hiding the latency of the data access by allowing the CPU to be used.

1. Main chare: This is where the execution of the program begins
2. Branch Office Chares: One exists on each processor and they execute concurrently.
3. Other chares:

In Charm, when a chare is created, a message is sent to it. Messages are also to a chare whenever a method is invoked on it.

Each Chare has

1. Entry point: This is a set of methods which can be executed by other processors. So when a message arrives, the function corresponding to the name in the message is executed.
2. Public and private data members and methods.

Data assignments are split phase and therefore hide latency and increase parallelism.

6. Synchronization issues

There is no explicit synchronization between various chares. The user can however use the following language features to communicate and synchronize

1. Read-Only Objects: Can be accessed from any processor.
2. Write Once Objects: These objects are written to exactly once after which they behave like read-only objects.
3. Accumulator Objects: An accumulator object has two operations “add” which adds to the object in a user defined manner, and combine which combines the two objects. The code for these functions is provided by the user. The operations are commutative and associative.
4. Monotonic Objects: Operations on these are commutative-associative and idempotent.
5. Distributed Tables: Essentially a database. The operations Insert Delete and Find are provided by the language.

7. Portability

The language does not seem particularly dependent on any OS or architecture feature. and should therefore be portable.

8. Semantic properties.

The language does not assure determinacy. Messages received by a chare are serviced as they are received making the language non-deterministic.

The address space is not global in that private members of logical processors cannot be accessed from other processors.

9. Syntax.

While syntactically the language is very similar to C++, programming in terms of chares instead of normal objects makes the user aware that the target machine is parallel. The other extensions to the language, have been discussed above.

10. Ease of Programming.

The style of programming in Charm++ is very different from most parallel and sequential languages and will take some getting used to. Otherwise Charm++ appears to provide a reasonable model to program in.

11. Efficiency

It is possible to write reasonably efficient code in Charm++. Since the compiler is not primarily responsible for extracting parallelism, performance is dependent on the user's ability to define chares and program in a fashion which makes good use of the synchronization primitives available.

12. Interesting features

1. The concept of logical processors elegantly encapsulates ideas like locality etc.
2. The run time system uses load balancing instead of data partitioning to get better performance.
3. The idea of programming chares, makes the user aware that the target machine is parallel which is useful in making him/her think parallel.

13. Problems with the language which make it less suitable to use.

1. The user has no control over data partitioning. While the run time system may do a good job, it is useful to give the user some power to over ride the default behavior.
2. The language will take some getting used to as it is very different from most sequential and parallel languages.
3. While multiple chares can reside on a single processor, a single chare cannot be parallelized further (i.e executed across a set of processors).

2.3.8 HyperTool (Min-You Wu, State University of New York at Buffalo)

1. The Model (presented to the user)

Hypertool[8, 19] is a system which parallelizes sequential code. The user codes the application in a sequential fashion, and then the hypertool system parallelizes the code using data flow analysis.

Hence the parallelism is implicit and the programmers view is essentially of a sequential machine. The user does not have to deal with message passing.

Since sequential semantics are preserved, determinacy is assured.

2. Target architecture

Hypertool seems targeted at users who want to use parallel machines but do not want to go through the trouble of learning a new language or a new paradigm of programming. Further Hypertool provides an easy way to parallelize existing sequential code.

The target architecture seems to be coarse grain distributed memory machine.

3. How is Parallelism achieved: data/loop/functional/implicit/etc.

Hypertool considers every procedure to be a indivisible unit of computation and represents it as a task. Based on this, a macro dataflow graph is formed with the dependencies between tasks represented as arcs. Tasks which don't have

read-write or write-write dependencies can be scheduled in parallel. The user annotates parameters as read-only or read-write to aid the compiler in assessing dependencies.

Hypertool carries out performance estimation and gives the user some performance results based on which the user may choose to modify the granularity of the procedures or change the algorithm.

Hence the parallelism that hypertool extracts is functional. It makes no attempt to exploit loop or data parallelism.

4. Data Partitioning

Based on the macro data flow graph, certain tasks are performed by certain processors. When a processor begins executing a certain task, all the objects the task references are sent to the processor at the beginning of execution.

Hence, neither the user nor the compiler tackle any data partitioning issues.

5. Parallel Constructs

There are no parallel constructs.

6. Synchronization issues

The user does not have to tackle synchronization issues. The compiler generates all the sends and receives required for the program to function and the run time system manages the scheduling.

7. Portability

The language appears to be portable across architectures. It has been implemented on the iPSC/2, iPSC/860 as well as a network of sparc stations.

8. Semantic properties.

The address space is global. Sequential semantics are preserved. Hence, the language assures determinacy.

9. Syntax.

The user programs in C. The only extensions are the directives IN and OUT which are used to tell the compiler whether parameters are read-only or read-write. The language preserves the semantics of C. This is very useful as the user can actually debug the code on a sequential machine before doing a parallel run.

10. Ease of Programming.

The language is very easy to program in.

11. Efficiency

The performance of a hypertool program depends largely on how well the user defines his/her procedures. Since a procedure is regarded as an indivisible unit of computation, it is necessary that the user define his/her procedures in a fine grain fashion to extract maximal parallelism.

Further, since the parallelism is functional, hypertool does not seem suited to fine grain machines.

For medium to large grain machines, within the paradigm provided, the tool performs well. But, the paradigm of programming itself does not allow the user to express, or the compiler to exploit, most of the parallelism inherent in a program.

12. Interesting features

1. Parallelizing existing sequential programs should be relatively easy using hypertool. If the user has some knowledge of the program, he can even modify it based on the performance results given and improve the running time.

13. Problems with the language which make it less suitable to use.

1. Unlike Jade, Hypertool does not allow the user to annotate call-sites of procedures. (Jade allows the user to annotate

each call-site with a list of variables read and written by that invocation of the procedure)

2. Again unlike Jade the run-time system does not check for possible violations of the IN and OUT directives.
3. The procedure is regarded as a task which can be a very coarse level of granularity.
4. If the user uses global variables and pointers, it may be difficult for the compiler to determine the data dependencies.

4. Conclusions

The survey of languages we conducted gave us some interesting insight into how languages tackle the various issues we brought up in the first two sections of this chapter. In particular languages take varied approaches to tackling implicit versus explicit parallelism in programs. In brief we can summarize how each of the languages tackles this issue

- C^{**}: Only procedures are labelled parallel, the compiler does the rest. So the actual RPC is performed implicitly.
- Jade: The user annotates call-sites and they are performed in the form of an RPC implicitly. Synchronization is not a concern.
- Mentat: Methods of certain objects are performed in a synchronized fashion implicitly through RPCs.
- pC⁺⁺: Synchronous parallel operations are implicitly done, while the user explicitly conducts the asynchronous ones through RPCs.
- CC⁺⁺: The user explicitly uses RPCs through spawn.
- Charm⁺⁺: The user programs at the process level and programs RPCs through entry points.
- μ C⁺⁺: is meant for shared memory multiprocessors and the RPCs are performed implicitly.
- Hypertool: RPCs are performed implicitly.

Hence at one end we have Jade, Mentat, Hypertool and μ C⁺⁺ which use RPCs implicitly, and at the other end we have Charm⁺⁺ and CC⁺⁺ where the RPCs are completely explicit. pC⁺⁺ and C^{**} are somewhere in the middle. In C^{**} the user has to think in the RPC paradigm but doesn't need to actually perform them, whereas in pC⁺⁺ the user performs some RPCs explicitly and the language performs others implicitly.

3. The Test Language.

3.1 Introduction

In order to examine the RPC paradigm we developed a language which allowed us to test how easy it was to code in this paradigm and also obtain some performance results concerning the code produced. We debated about using one of the languages we surveyed as the medium to test the entire paradigm, but decided against it as none of the languages we surveyed was a pure form of the RPC paradigm. What we needed to test the paradigm was a language that had all the RPC functionality and no other parallel constructs, i.e. a pure RPC language.

3.2 Language Description

We designed and implemented an Object Oriented parallel language. The language is an extension of C++. The classes the user defines have local and global components (each with their respective private, public and protected sections). The user defines a template which is a representation of the virtual machine s/he is working on. The virtual machine can currently only be a host processor coupled with a multidimensional array of local processors (extensions could include trees, hypercube, butterfly etc.). The local component of each class is resident on each of these processors, whereas the global section is resident on one processor (typically the front end) only. To each processor, what is visible is its own local portion of the class, the public methods of the global section and the public methods of the local sections of other processors. RPCs can be made on the public methods of other processors as well as the public methods of the global section of the class. In some sense the global data and methods are "shared" by all the local segments. Since we have attempted to develop a pure RPC language, the user cannot make remote data accesses directly in the test language. All such data accesses are made through procedure calls. For example if a processor were to access a global variable it would have to do so through an accessor function.

3.2.1 The Remote Procedure Call; syntax and semantics

The syntax for a RPC is similar to a C++ procedure except that after the semicolon, the user must mention, in square brackets ([]) the virtual processor number (e.g “1, 2” in a two dimensional array) on which to perform the RPC. If the remote procedure call is to be made on the global data, the user annotates the call with a “[host]” after the semicolon. The compiler inserts the appropriate sends and receives in order to perform the RPC. The semantics of the RPC are as follows

- RPCs are blocking (i.e the caller must wait till the callee services the request).
- While waiting for the callee to service its request, the caller will service pending RPC requests which were made on it.
- The caller is informed by the callee of the completion of the call, after which the caller resumes its work.

The user may choose to perform a non-blocking RPC(i.e the caller does not block) by entering the asynchronous mode. The user has two calls at his/her disposal the begin and end asynchronous calls (“begin_asynchronous()” and “end_asynchronous()”). In the asynchronous mode, the caller and callee work in parallel and this thus improves performance. It is however not always possible perform RPCs in the asynchronous mode as there may exist data dependencies which need to be resolved in a particular fashion. Further if the RPC expects a return value, the call cannot be non-blocking. The begin_asynchronous() call starts the asynchronous mode and the end-asynchronous() call waits for all RPCs made in asynchronous mode to complete and then flips to non asynchronous mode. For example, consider the following piece of code,

```
foo();[2,1]      /* Perform “foo” on processor 2,1 in a blocking fashion */  
  
[begin_asynchronous]  
  
foo();[2,1]  
  
bar();[1,2]     /* “foo” and “bar” are performed in parallel */  
  
[end_asynchronous] /* Wait for “foo” and “bar” to complete */
```

3.2.2 Synchronization Primitives

The user is also provided with synchronization primitives which can be used to enforce the correct order of operations and therefore get deterministic behavior. The primitives provided are “barrier” and “synchronize”. Both primitives

must be executed by all threads else there will be a deadlock. The semantics of the barrier are as follows

- Upon reaching the barrier the thread informs the host thread that it is at the barrier.
- The host thread waits till all threads have informed it of reaching the barrier and then informs all the threads that they may continue.
- All threads wait till they receive the go-ahead signal from the host thread and then they continue their work.
- During the time that a thread is waiting for the go-ahead signal from the host thread, it services pending RPCs.

The synchronize primitive is different from the barrier in that the threads do not service pending RPCs while they wait. This primitive must be used with caution as no RPCs are performed while the thread waits for the go ahead and it may happen that one process is waiting for the completion of an RPC on another process which is performing a synchronize. The thread performing the synchronize cannot complete till the caller also performs a synchronize and the caller cannot perform the synchronize as it awaits the result of the RPC on the callee, thus resulting in a deadlock.

The language also allows the “accept” operation (this is not part of the pure RPC paradigm and we will motivate its need in the next section). The accept operation causes the thread performing the operation to block till some other thread performs an RPC on it (if there is a pending RPC, the thread performs that RPC and returns immediately).

3.2.3 Syntactical extensions to C++

The syntax of the language is similar to that of C++. The few differences come from the extensions made to C++.

- The user defines in the main file an object of type Object. As parameters to the constructor of this object, the user passes the dimensions of the multidimensional array s/he wishes the virtual machine to be.

For example,

```
Object(2,2);    /*describes a 2 x 2 array of processors */
```

```
Object(2,3,2); /* describes a 2 x 3 x 2 array of processors.*/*
```

This instantiates a virtual machine. The second example of an instantiation of an object of the class Object creates 12

processes and the user can think of these as a 3 dimensional array of processors.

- RPCs are made by indicating the virtual processor on which the RPC is to be performed in square brackets after the semicolon. The square brackets could either contain the keyword “host” (which implies that the RPC is performed on the host thread) or it could contain a set of integers separated by commas. The number of integers should be the same as the dimension of the virtual machine and the integers uniquely identify the virtual processor on which the RPC is to be performed. For e.g.

```
Object(2,2);    /* definition of virtual machine */
```

```
foo();[0,1]    /* RPC of “foo” on virtual processor 0,1 */
```

- The user can describe the beginning and end of asynchronous operations, a barrier or a synchronization operation by indicating the operation to be performed within square brackets. The user can also demarcate parts of the file as containing local or global functions. For e.g.,

```
[global]      /* Global functions follow */
```

```
[local]      /* local functions follow */
```

```
[barrier]    /* Perform a barrier */
```

```
[synchronize] /* Perform a synchronization operation */
```

```
[begin_asynchronous]/* Begin performing RPCs in a non-blocking fashion */
```

```
[end_asynchronous]/* Wait for RPCs performed in asynchronous mode to complete and then begin performing RPCs  
in a blocking fashion */
```

- A fundamental problem in parallel languages is pointers. Pointers are advantageous in that
 - They make it easy for the user to pass as a parameter to a procedure a set of elements.
 - The calling procedure may not even need to know the number of elements the invoked procedure is going to access.
- It is probably too restrictive to disallow the use of pointers in RPCs (whether implicit or explicit). Our language does allow the use of pointers in RPCs, however the user must know how many elements the invoked procedure may access. Therefore we allow the user to exploit the first advantage, but do not allow him/her to use the second. The syntax of an RPC with a pointer parameter is as follows,

```
foo(i,data{3},j); /* where i and j are integers and data is an integer pointer and foo is a method of type void foo(int, int  
*,int). The {3} implies that foo will access no more than the three integers following data. */
```

If a parallel language allows the user the use of pointers and yet provides both the advantages of pointers, it will have to implicitly take care of mapping memory from one processor onto another. This will involve initially sending a certain amount of memory to the callee (from the caller), and then if the callee ever accesses memory which has not been brought over from the caller, hooks will have to be called which will perform the accesses (from the caller). This scheme though more transparent and easier to use than the previous scheme suffers from being less efficient (as multiple communications may be needed between caller and callee for a single RPC). For example if as before we call "foo" with an integer pointer "data", and do not specify the amount of memory foo may access, then the system will send to the callee a certain amount of memory (that is pointed to by "data"). If "foo" accesses more than is sent to it, the system will have to, at run time, get more of the memory that is pointed to by "data", from the caller.

- A user's class definition looks like the following

```
class Matrix{  
    global:  
        private:  
            /* All global data, private methods */  
        public:  
            /* Constructor, destructor, accessor functions, etc.*/  
        protected:  
            /* protected global methods and data */  
    local:  
        private:  
            /* all local data, local private method */  
        public:  
            /* Constructor, destructor, accessor functions, data etc.*/  
        protected:  
            /* protected local methods and data */  
};
```

3.2.4 Semantics differences with C++

The user writes code by defining the objects and then writing a main function which instantiates these objects and performs operations on them, like in C++. There are however a few restrictions imposed upon the user in order to get a pure RPC paradigm

- Global methods cannot access or modify local data directly. All such accesses must be performed through RPCs.
- Local methods cannot access or modify another processor's local data or the global data directly. Once again such accesses are performed through RPCs.
- A local method cannot return an object (or a pointer to one) which has a global part.
- If a class is defined without the keywords local and global i.e in a purely C++ fashion, it is accessible only by the virtual processor which instantiates it. All other processors have no knowledge of its existence.
- An RPC can only contain pointers to objects, not references or copies of them. The reason for this restriction is the way in which we implement distributed objects.
- Further, an object cannot perform an RPC on another object's methods.

In general, to perform any significant computation a program will usually call a global function which performs some operations on global data and which then calls local functions to perform local work. The local functions will communicate with each other using the RPC paradigm.

3.3 Implementation

We used a package called PVM (parallel virtual machine) as the base for developing our language. This package has a C and Fortran interface and allows the user to spawn multiple tasks and perform sends and receives between them. It further takes care of mapping multiple tasks onto multiple CPUs. For example if a user defines 8 processes and there are only 4 CPUs, PVM maps the processes such that each processor handles 2 processes (Brent's principle[20]).

Our compiler (it's actually a source to source translator) converts the users code into C++ with PVM calls. Two executables are produced, one for the global thread and one for the local threads (an SPMD program for the local threads).

The user's ".H" file is parsed using flex¹ and bison² and is converted into a local ".H" file (for the threads for each local process) and a global ".H" file (for the host thread). The user's ".C" file is parsed using C++ and converted into a local ".C" file and a global ".C" file. These are then compiled using "gcc³".

The parsers for the ".C" and the ".H" files report errors specific to the parallel language and let the C++ compiler handle errors specific to C++. Our emphasis in this project has been to develop a platform to test the RPC paradigm; we have not therefore concentrated on improving the interface or the robustness of the parallel language.

3.3.1 An example

We present below an example program of matrix multiplication written in our language.

The "matrix.H" file

```
/* This file defines the class Matrix, its methods and the data it encapsulates */  
/* The Matrix class has information regarding the number of rows and columns in the matrix as well as the elements  
of the matrix. The number of rows and columns are maintained globally, however the matrix entries are maintained  
locally. Locally, information is also kept about the row and column number of the entry Accessor functions are  
defined for accessing local and global data. C++ allows users to define operations (*, /, + etc.) on different classes.  
For example, the multiplication (*) operation has no meaning for a matrix class, but the user can define it to mean  
matrix multiplication. In C++ terminology, this is known as "operator overloading". In the example shown below,  
the operator * (multiplication) has been overloaded to symbolize matrix multiplication. The definition of this over-  
loading is also given. What follows is the declaration of the class with all the data elements and member functions  
which comprise it.*/
```

-
1. A Lexical Analyzer.
 2. A parser.
 3. A C++ compiler

```
class Matrix{
```

```
    global:
```

```
    /* What follows are declarations of global data and methods. */
```

```
        private:
```

```
        /* The private section of the global section of the class is accessible only by the  
        global methods */
```

```
            int _rows;        /* The number of rows in the Matrix */
```

```
            int _cols;        /* The number of columns in the Matrix */
```

```
        public:
```

```
        /* The public methods of the global section are accessible by other objects, by the  
        local section and by any method which has an instance or a pointer of the class  
        */
```

```
            Matrix(int rows, int cols, int *data); /* Global constructor */
```

```
            ~Matrix();        /* Global destructor */
```

```
            Matrix operator *(Matrix &B); /* Overloaded operator * */
```

```
            int get_cols() {return _cols;} /* accessor function */
```

```
    local:
```

```
    /* What follows are declarations of local data and methods. These local data and methods are repli-  
    cated on all local processors */
```

```
        private:
```

```
        /* The private section is visible only to the local processor on which it resides */
```

```
            int *_value;        /* Local value of matrix entry */
```

```
            int _row_val;        /* The row to which the entry belongs */
```

```
            int _col_val;        /* The column to which the entry belongs */
```

```

public:
/* The public section is visible to other local parts and the global section of the
same object only */

Matrix(int row_val, int col_val, int *data); /* Local construc-
tor*/

~Matrix(); /* Local destructor */

void local_multiply(Matrix *B, Matrix *C); /* local multiplica-
tion function */

int get_data(); /* Accessor function*/

void set_data(); /* Modifier */

int get_remote_data(int i,int j); /* Does a remote access */

};

```

The “matrix.C” file

```

/* This file defines the methods of the class Matrix (both global and local). */

```

```

#include “matrix.H” // Include the definition of the class Matrix.

```

```

[global] // i.e. what follows are the definitions of global functions.

```

```

/* The global constructor, initializes the private members of number of rows and number of columns and calls the local
constructors to initialize the local values of the class. If there is no data (data == 0) then the local values are initial-
ized to 0 (default). The local constructors are invoked in parallel to improve performance. */

```

```

Matrix::Matrix(int rows, int cols, int *data){
    int i,j; // declaration of local variables

    _rows = rows; // Initialize data.
    _cols = cols;

```

```

    _data = data;

    [begin_asynchronous] // Call local constructors in parallel.
    for(i=0;i<_rows;i++)
        for(j=0;j<_cols;j++)
            if(!_data)
                Matrix(i,j,_data[i*_cols+j]);[i,j]
            else // If there is no data initialize local elements to default.
                Matrix(i,j,0);[i,j]

    [end_asynchronous] // Wait till the object is created.
}

/* The global destructor de-allocates the memory chunk with the elements of the matrix. Thereafter local destructors
are called in parallel. */

Matrix::~Matrix(){
    int i,j;

    delete _data; // De-allocate memory.

    [begin_asynchronous]
    for(i=0;i<_rows;i++)
        for(j=0;j<_cols;j++)
            ~Matrix();[i,j] // Call local destructors in parallel.

    [end_asynchronous]
}

/* This is the overloaded operator *. This method gets called when a statement like A*B appears, where A and B are
both matrices. The method is called on object A, with object as parameter. The method works as follows. A new
matrix is created with default values of elements. Then, local multiplication operations are called in parallel, at the
end of which the correct values of the newly created matrix are present in the local processors. The method then
returns the newly created matrix.*/

```

```

Matrix *Matrix::operator *(Matrix &B){
    int i,j;

    Matrix *C = new Matrix(_rows,B->get_cols(),(int *) 0); /* Initialise new matrix. */
    [begin_asynchronous]// Do local work in parallel.
    for(i=0;i<_rows;i++)
        for(j=0;j<_cols;j++) // For all processors
            local_multiply(&B,C);[i,j] // Call local multiply in parallel.
    [end_asynchronous] // Wait for completion.
    return C; // return created matrix.
}

```

[local] // i.e. the definition of local methods follow.

/* The local constructor initializes the row and column number and the value of the element of the element of the matrix, the local processor is responsible for. */

```

Matrix::Matrix(int row_no,int col_no,int data){
    _row_no = row_no; // Initialise data.
    _col_no = col_no;
    _data = data;
}

```

/* The local destructor performs no work. */

```

Matrix::~Matrix(){
}

```

/* The local multiplication operation. This essentially computes the dot product of the object it is called on with the object which is given to it as first parameter. The result is stored in the object which is the second parameter. The dot product is computed by requesting for non-local elements from remote processors and multiplying them and adding the result. The local value of the result matrix is updated. */

```

void Matrix::local_multiply(Matrix *B, Matrix *C){
    int i,j;
    int sum = 0;
    int num_cols;
    int temp1,temp2;

    num_cols = get_cols();[host]
    for(i=0;i<num_cols;i++){// Compute inner product.
        if(i == col_no)                // then data is local.
            temp1 = data;
        else
            temp1 = get_data();[_row_no,i] // access remote data.
        temp2 = B->get_remote_data(i, col_no);
        sum+=temp1*temp2; // Add to get dot product.
    }
    C->set_data(sum); // modify value.
}

/* Modify the local value of the element */
void Matrix::set_data(int sum){
    _data = sum;
}

/* Accessor function to get the value of the element */
int Matrix::get_data(){
    return _data;
}

/* To fetch the data of an element, which may not be local to the current processor. This method is typically called
when a remote (or local) value of another object may be needed. */

```

```

int Matrix::get_remote_data(int i,int j){
    if((i == _row_no) && (j == _col_no)) // data is local.
        return _data;
    return get_data()[i,j]
}

```

The “main.C” file

/* This file instantiates the objects and calls the required functions on them. It must also create the virtual machine class.

```

main(int argc,char **argv){
    Objects *__objects;
    int *data;
    int rows,cols,i;

    /* Read in and initialize Matrix A and Matrix B. Code is not provided for this part.*/
    __objects = new Objects(rows,cols); /* Creates a virtual machine of rows x cols processors*/
    Matrix *C = A*B; /* Creates a new Matrix * C which is A*B */
    __objects->terminate(); /* Indicates end of parallel part of program. */
}

```

The execution of the program begins in the “main” function. This function begins by reading in the values of matrices A and B and initializing them (code for this is not shown). Then it instantiates a virtual machine. It then calls the matrix multiplication operation for the Matrix class on object A, with object B as parameter.

Control now passes onto the global matrix multiplication method. This method creates a new matrix with default values (0) for elements and then calls the local matrix multiplication routines in parallel on object A, with object B and the newly created object as parameters.

The local matrix multiplication routines compute the dot product (of vectors of A and B) by requesting remote values of A along the row (of the two dimensional mesh (the virtual machine)) and remote values of B along the columns.

The result is stored as the local value of the newly created matrix.

At the completion of this task, the local multiplications return and control passes back to the global matrix multiplication routine. The global routine waits for all local multiplications to complete and then returns to the “main” function the newly created object. The “main” functions then terminates the parallel operations.

3.3.2 Conversion to C++ with PVM calls

We implement distributed objects by giving each object created a unique object identifier. This object identifier is passed to the local objects at the time of creation (constructor call) and thereafter every time a remote call is made, in the converted code we add the object number of the calling object as part of the message sent to the local processor. The message also includes the procedure number to be called and the parameters to be passed to it. A library function at the local processor will strip the headers, identify the object on which the call is to be made, the procedure which is to be called and the parameters it is to be invoked with. The call is then performed and the return value (if any) is added to the packet which is sent back to the caller informing it of the completion of the call. The library function for handling receives is invoked whenever a processor is waiting for the completion of an RPC or a barrier. The library function performing the send is invoked when an RPC is made.

Given this scenario, the implementation essentially required maintaining a list of global and local procedures (at parse time), assigning them procedure numbers and creating the library module (the library module is unique for each program) to deal with the RPCs. The “.C” and “.H” files are then split and all RPCs are converted into library calls.

We shall not go into the details of the implementation as the essence of the report is the exploration of the paradigm.

4.0 Conclusion

Implementing the test language was surprisingly easy. We believe that the availability of tools like PVM will stimulate a great deal of research into the field of parallel languages. Further, while the scientific community is unable to reach a consensus on a standard parallel C or C++, it is tools like these which will be increasingly used.

4. Results and Conclusions

We used the test language we developed to write a number of programs. This gave us an idea of how easy it was to express parallel algorithms in the pure RPC paradigm. We also took some performance results and made some interesting discoveries about the ability of the RPC paradigm to produce efficient parallel code. We ran our tests on a network of sparcstations, which is an extremely coarse grained parallel processor, as communication delays are high.

4.1 Matrix Multiplication

The first application we programmed was matrix multiplication. In the previous section, we presented some sample code for matrix multiplication. That is probably the most natural way in which to program matrix multiplication for a parallel machine as it ensures equal load distribution. When we took performance results on the running time of that program and compared it to a sequential version of matrix multiplication, we were surprised to find that the parallel program (on 16 processors) performed 8 times slower (real time) even on 256 x 256 size matrices (as the size of the matrices increases performance of the parallel program becomes better). When we looked closer at the program, we found that we were performing a lot of communication (n^2 RPCs for multiplying $n \times n$ matrices). Given that communication delay times are so large, an efficient algorithm for a network of sparcstations needs as little communication taking place as possible. Further the initial cost of spawning processes is fairly high, and in the example shown, we spawn n^2 processes.

As a first improvement to this application, we worked with as many processes as there were CPUs (so no CPU handled more than one process), by giving each process more than one element to work on. This reduced the number of processes spawned and improved performance to a point that the parallel program performed only twice as slow as the sequential one for 256 x 256 matrices.

We then reduced the communication between the processes, by sending each process all the information it needs, in a

single RPC at the beginning of the algorithm. While this reduced the number of times communication took place (to a constant), it in no way reduced the number of items to be communicated. Yet this scheme worked extremely well as can be seen by the attached table (Table 1) and graph (figures 1 and 2). The reason for the drastic improvement in performance is that the communication delay time for a network of workstations being high, the dominant factor in the running time of the algorithm is the number of communication events. The improved parallel algorithm was more than 3 times faster than the sequential one for a 256 x 256 size matrix on 16 processors. Figure 1 compares the running times of the parallel algorithm on 1, 2, 4 and 16 processors with the sequential algorithm. Figure 2 highlights the start up costs by showing the initial part of figure 1 in greater detail. What follows are the performance results of the improved algorithm on 1, 2, 4 and 16 processors.

Table 1:

Size of Matrix	Sequential	1 processor Parallel	2 processor Parallel	4 processor Parallel	16processor Parallel
32 x 32	0.1	0.7	1.3	1.5	3.9
64 x 64	0.8	1.8	2.1	2.7	4.0
128 x 128	6.0	9.5	6.2	6.1	6.0
256 x 256	46.9	57.9	41.1	27.6	17.0
512 x 512	421.2	662.2	345.1	213.3	76.1

Some observations:

- The graph and the table of results clearly illustrate the start up cost of a parallel program. As can be seen from the graph until when the size of the matrices are 128 x 128, the overheads of creating the parallel processes outweigh the advantages of performing the computations in parallel
- It is further evident that as the number of processes increases the start up time increases as well.
- The one processor parallel algorithm is slower than the sequential counterpart as in the one processor parallel case, one CPU simulates the local and global actions of the user's class
- What is gratifying to note however, is that as the size of the matrices to be multiplied increases, the parallel program

performs far better than the sequential one.

- Further, users are unlikely to use parallel programs to do extremely small tasks like multiplying 32×32 matrices, and are more likely to use parallel programming to perform really large tasks where the real gain of parallelism is evident.

As mentioned earlier, the most natural way of programming matrix multiplication was not the most efficient. The example shown in the previous section would have worked well for a fine grained machine. This clearly shows that the algorithm the user needs to write is architecture dependent. The RPC paradigm does not make the architecture transparent to the user and hence

- The user can write code targeted at a particular architecture and thus get better performance.
- However, the code written will perform badly on other architectures and in that sense it ports poorly.

4.2 Gaussian Elimination:

The next application we looked at was gaussian elimination. Unlike matrix multiplication, gaussian elimination requires a great deal of synchronization between processors. The most natural way to program gaussian elimination is using barriers. The first gaussian elimination algorithm (Gauss 1) we wrote did the following

- Distributed an equal number of rows to each processor (with no duplication).
- Iterate as many times as there are rows
 - The processors which contain rows whose indices are smaller than the iteration index perform a barrier.
 - The processor which contains the row whose index is the same as the iteration index performs a barrier and then carries out its local work (i.e local steps of gauss elimination).
 - All other processors, get the required values from the processor which has the row number same as the iteration number, perform their local work and then perform a barrier.

Hence in this implementation n barriers are performed for $n \times n$ matrix. A barrier involves the sending and receiving of $2n$ messages, and further it causes a bottleneck at the global thread which needs to receive and process all the barrier requests and is therefore a costly operation.

If one were to hand code gauss elimination, using sends and receives, the processor with the row with the same number as the iteration index would begin by sending the required information and then doing its local work. Other processors merely perform receive so that they receive the values from that processor and then do their local work. In the RPC paradigm, since the caller has control over when to execute the RPC, and not the callee, we need barriers to synchronize.

Given that barriers are costly and should therefore be used sparingly, we devised a method using which the callee also has control over when an RPC is performed (thus obviating the need for a barrier in this algorithm). We extended the RPC paradigm to include the accept statement. An accept has the following semantics. If an RPC is pending on the processor, it is performed and the call returns immediately, else the processor waits till an RPC is performed on it.

Based on this we modified the algorithm so that the processor which has the row with the same number as the iteration number, performs as many accepts as there are active processors. There is no longer a need for barriers and the algorithm therefore becomes:

- Distribute an equal number of rows to each processor (with no duplication).
- Iterate as many times as there are rows
 - The processors which contain rows whose indices are smaller than the iteration index do nothing.
 - The processor which contains the row whose index is the same as the iteration index performs as many accepts as there are processors which need information from it and then carries out its local work.
 - All other processors, get the required values from the processor which has the row number same as the iteration number and then perform their local work.

As can be seen by the table (Table 2), we register a slight improvement in performance, but this algorithm (Gauss 2) yet performs far worse than the sequential version. What is happening in this algorithm is that at each stage, all active processors perform an RPC on one single processor thus increasing the computation along the critical path and also creating a bottleneck for servicing requests.

Gauss 3 is an algorithm which remedies this. Rather than have all processors perform RPCs on a single processor, we make that processor (the one which has the row with the same number as the iteration index) perform RPCs on all other

processors informing them of the required values (i.e rather than other processors requesting a value, this processors informs other processors). This effectively removes the bottleneck and decreases the computation along the critical path (as now all the RPCs informing other processors are performed in parallel). The algorithm then becomes:

- Distribute an equal number of rows to each processor (with no duplication).
- Iterate as many times as there are rows
 - The processors which contain rows whose indices are smaller than the iteration index do nothing.
 - The processor which contains the row whose index is the same as the iteration index performs RPCs informing the processors that need information from it of the values of the required variables and then carries out its local work.
 - All other processors, perform accepts in a loop till they get the required information and then perform their local work.

The table (Table 2) does indeed show that Gauss 3 performs far better than either Gauss 1 or Gauss 2 and does better than the sequential version for matrices of size greater than 512 x 512. The results that follow are for programs executed on 8 processors.

Table 2:

Size of Matrix	Gauss 1	Gauss 2	Gauss 3	Sequential
32 x 32	60.1	56.1	2.6	0.1
64 x 64	429.6	412.4	3.6	0.5
128 x 128	*a	*	6.6	3.5
256 x 256	*	*	30.0	27.4
512 x 512	*	*	136.6	218.3

a. Timing not carried out as running time expected to be greater than 25 minutes.

The Gauss elimination application program demonstrated to us the need for the accept statement, as it allows the callee to determine when an RPC takes place thus making the costly synchronization less needed. It must be pointed out, that the running times of the 3 algorithms would not be so drastically different on a fine grain machine (Gauss1 and Gauss2 should do better than the sequential algorithm on a fine grain machine). One problem with the accept statement is that

it leads to unstructured programming and exposes the user to the send - receive paradigm. However, if the statement is used judiciously, the user gains a lot in efficiency and loses only a little in the elegance of the programming model.

4.3 Fast Fourier Transform

Interestingly, our first attempt at a parallel implementation of the FFT was an efficient one. It was based on the experience we had gained in designing the previous algorithms. Our implementation is specific to polynomial multiplication. A broad outline of the algorithm follows

- We create as many processes as there are CPUs. We distribute the coefficients equally between the processors.
- Split the coefficients into their odd and even parts and perform polynomial multiplication recursively on them. The base case is when the size of the polynomials to be multiplied is the original size divided by the number of processors.
- Do the conquer step and add up the coefficients.

This algorithm ran 3 times faster on 8 processors than the sequential one for polynomials of size 256.

4.4 Conclusions:

In the RPC paradigm only the caller has control over when an RPC is made; the callee has no control over when the RPC is executed. Thus the RPC paradigm, in screening from the user the underlying send-receive paradigm, has given him/her less power to express parallel algorithms. By extending the paradigm to include the “accept” statement, the user retains the expressive power of the send-receive paradigm and also has the transparency provided by the RPC mode of programming.

In an implicit parallel language, since the RPCs are handled by the compiler and run time system, the user cannot fine tune his algorithms and make them more efficient. The RPC paradigm, in not abstracting away the architecture of the underlying system, gives the user the power to express algorithms in an efficient fashion.

The flip side of exposing the user to the underlying architecture is that the code written in a pure RPC language is specific to an architecture and will perform poorly if ported to a different architecture.

Given the fact, that parallel compilers today are not capable of extracting the desired performance from user code, most applications still need to be hand coded. The RPC paradigm provides an elegant method of hand coding parallel applications. However, if and when compilers do become sophisticated enough to do a good job, implicit parallel

languages are likely to replace explicit ones completely.

5.0 References

- [1] Bilardi, Gianfranco and Franco P. Preparata, "Horizons of Parallel Computation," *CS-93-20*, Brown University.
- [2] Netzer, Robert H. B. and Barton P. Miller, "What are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Languages and Systems I*, 1, March 1992.
- [3] High Performance Fortran Forum, "High Performance Fortran Language Specification," Version 1.0 Draft, Jan. 25 1992.
- [4] Tanenbaum, Andrew S., "Modern Operating Systems," Prentice Hall, pp. 417-445.
- [5] Sunderam, V. S., "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practise and Experience*, vol. Vol. 2 No. 4, pp. 315-339, Dec. 1990.
- [6] Lamport, Leslie, "Time, Clocks and the ordering of Events in a Distributed System," *CACM* 21(7) pp. 558-565(July 1978).
- [7] Netzer, Robert H. B., and Barton P. Miller, "Experience with Techniques for Refining Data Race Detection," *Fifth Workshop on Languages and Compilers for Parallelism*, New Haven, CT, August 1992.
- [8] Cheng, Doreen Y., "A Survey of Parallel Programming Languages and Tools", *Report RND-93-005*, March 1993.
- [9] Larus, James R, "C**: A Large Grain, Object Oriented, Data Parallel Programming Language", *5th Workshop on Languages and Compilers for Parallel Computing*, Aug. 1992.
- [10] Rinard, M. C, D. S. Scales and M.S. Lam, "Heterogenous Parallel Programming in Jade," *Proceedings of Supercomputing '92*, pp. 245-256, Nov. 1992.
- [11] Lam M. S., M. C. Rinard, "Coarse-Grain Parallel Programming in Jade," *Proceedings of the third ACM SIGPLAN Symposium on Principles of Programming Languages*, pp 105-118, Jan 1992.
- [12] Grimshaw A. S, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, May 1993.
- [13] Grimshaw A. S, W. Timothy Strayer and Padmini Narayan, "The Good News About Dynamic Object-Oriented

- Parallel Processing," *University of Virginia, Computer Science Report TR-92-41*, 1992.
- [14] Bodin, Francois, Peter Beckman, Dennis Gannon, Srinivas Narayana and Shelby Yang, "Distributed pC++: Basic Ideas for an Object Parallel Language", *Proceedings of Supercomputing'91 (Albuquerque, Nov. 1991)*. *IEEE Computer Society and ACM SIGARCH*, pp. 273-282.
- [15] Gannon, Dennis "Object Oriented Parallel Programming: Experiments and Results".
- [16] Buhr, P. A, "uC++: Concurrency in the Object-oriented Language C++",
- [17] Chandy, Mani K and Carl Kesselman, "CC++: A Declarative Concurrent Object Oriented Programming Notation."
- [18] Kale, L.V, "The Chare Kernel Parallel Programming Language and System," *Proc. of the International Conference on Parallel Processing*, Aug. 1990.
- [19] Wu, Min-You and Daniel D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Systems* vol. 1 No. 3 pp. 330-343, July 1990.
- [20] Brent, R.P., "The Parallel evaluation of general arithmetic expressions," *Journal of the ACM*, 21, 2 (1974), pp. 201-206.

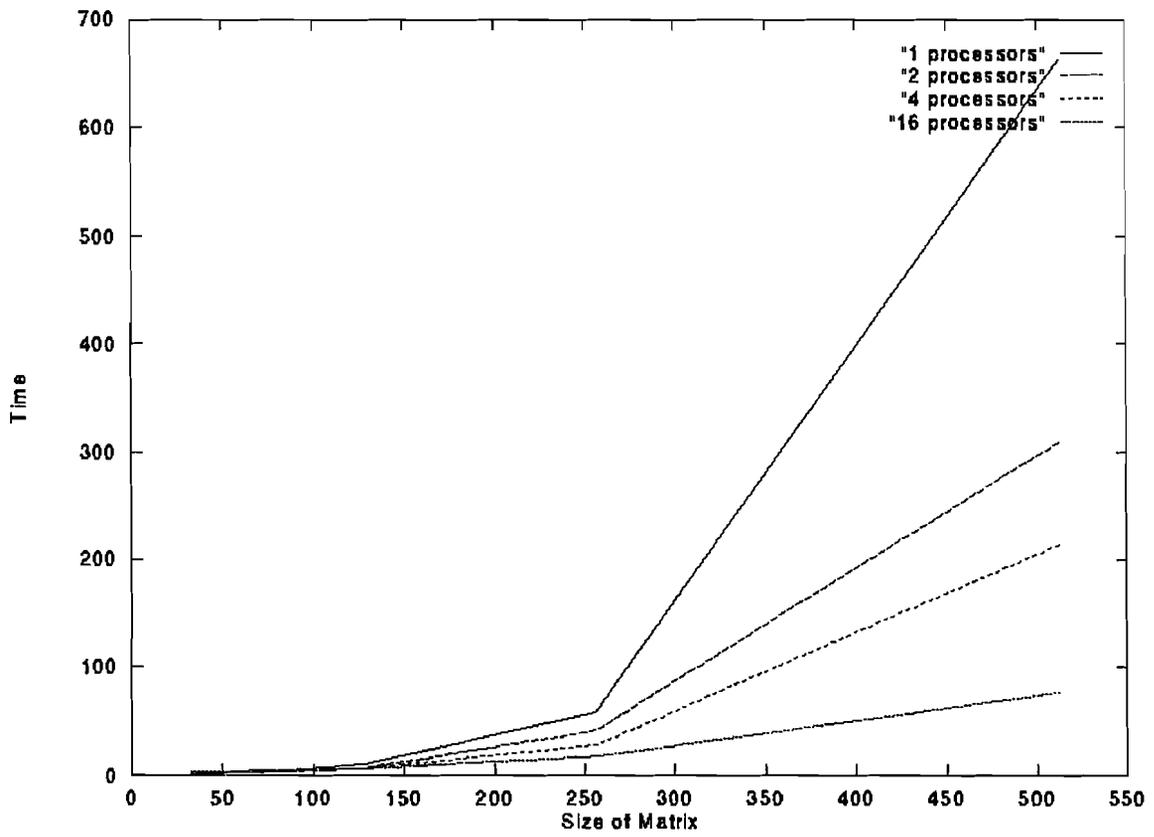


Figure 1.

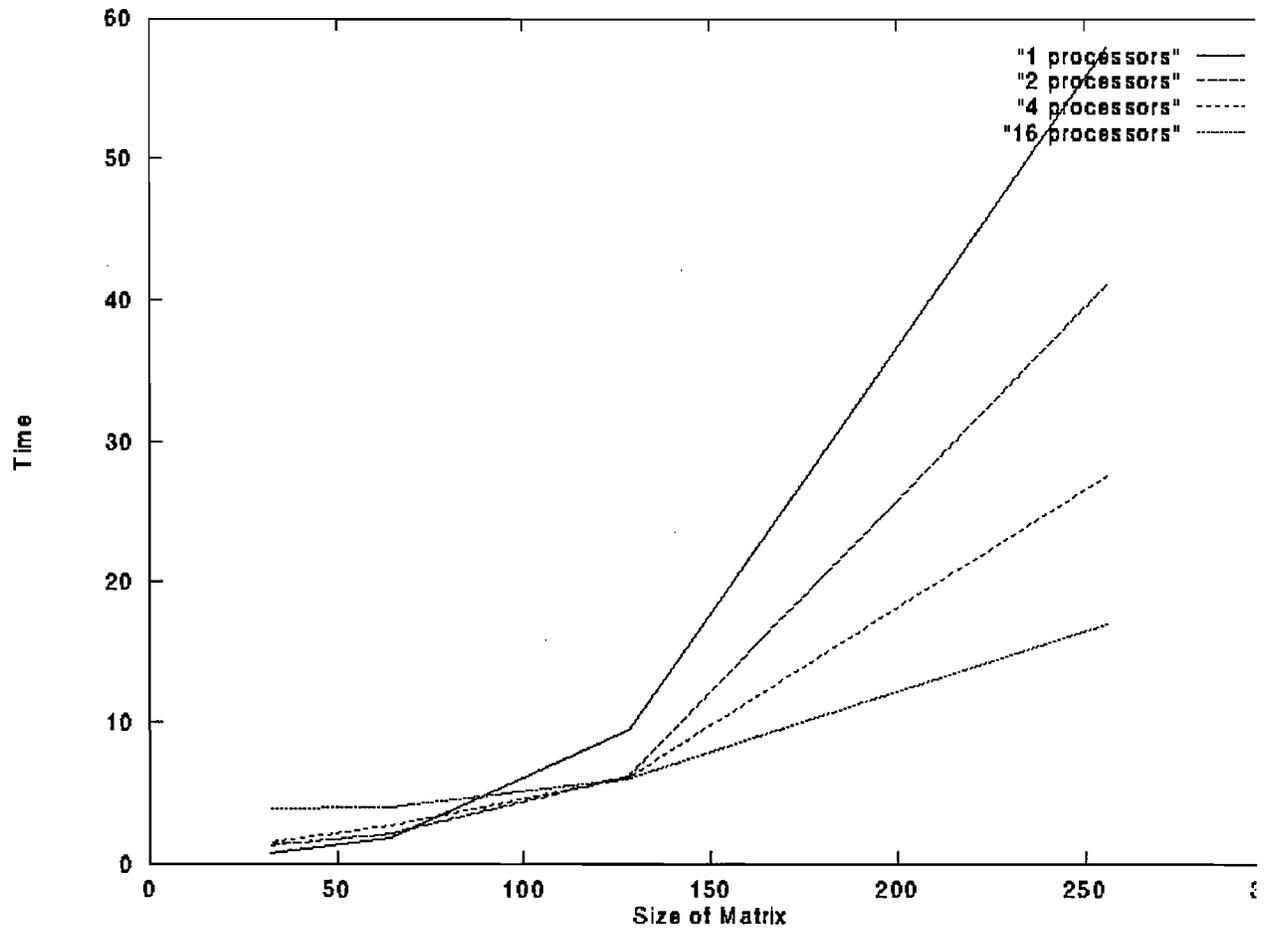


Figure 2.