MCC Technical Report Number: PP-163-87

Performance of Symbolic Applications on a Parallel Architecture

MCC Non-Confidential

Adolfo Guzman, Edward Krall, Patrick McGehearty, Nader Bagherzadeh

May, 1987

The results of a study of several parallel applications on a parallel symbolic architecture are presented. The architecture being simulated is characterized as a shared memory, hierarchical, and clustered architecture. Speedup measurements were obtained from six different application kernels. Measurements were also performed to assess the degradation of speedup as interconnection delays increase. These measurements allowed understanding of some of the interactions that take place among architectural parameters and the specific applications, and suggested ways for improving of the parallel architecture. The results presented highlight the manner in which application characteristics place limits on achievable parallel performance, given particular architectural features. The paper discusses processor starvation, fine grain parallelism, uneven loads, foreign reference, costs scheduling and indeterminate computation with respect to the applications chosen.

Artículo 70

Performance of Symbolic Applications on a Parallel Architecture

Adolfo Guzman, Edward J. Krall, Patrick F. McGehearty, Nader Bagherzadeh

Parallel Processing Architecture
Advanced Computer Architecture Program
MCC

ABSTRACT

The results of a study of several parallel applications on a parallel symbolic architecture are presented. The architecture being simulated is characterized as a shared memory, hierarchical, and clustered architecture. Speedup measurements were obtained from six different application kernels. Measurements were also performed to assess the degradation of speedup as interconnection delays increase. These measurements allowed understanding of some of the interactions that take place among architectural parameters and the specific applications, and suggested ways for improving of the parallel architecture. The results presented highlight the manner in which application characteristics place limits on achievable parallel performance, given particular architectural features. The paper discusses processor starvation, fine grain parallelism, uneven loads, foreign reference, costs scheduling and indeterminate computation with respect to the applications chosen.

1. OVERVIEW

It is generally agreed that parallel computers will be one of the master pillars in the design of advanced architectures capable of supporting the symbolic processing tasks of future generation computing systems. Thus, the Advanced Computer Architecture Program of MCC proposes, studies and evaluates parallel architectures. Often, these evaluations are done using existing symbolic applications, converted into parallel programs.

This paper reports results obtained from running six symbolic processing application kernels, in an architecture that can be characterized as "shared memory." The tool used for the measurements was the FutureLisp Emulator running on a Lisp machine.

1.1. FutureLisp

FutureLisp is a dialect of Common Lisp [Steele, 1984], augmented by the addition of *futures*. The *future* is a construct of MultiLisp, a parallel dialect of Lisp devised by [Halstead, 1984]. A *future* returns a pointer to the eventual value of its argument. In effect, it is a promise to compute the value asynchronously with the calling routine. If the computation is not yet completed when that value is needed by the caller, then — and only then — will the caller wait.

FutureLisp contains a number of Lisp functions built on *futures,* such as *pmapcar*, the parallel counterpart of *mapcar*. It also contains primitives for synchronization, such as *wait-sema* and *signal-sema*.

1.2. The Architecture and the Experiments

The applications were simulated on an architecture (See Figure 2-1) with the following characteristics.

- Each processor has access to three types of memory: **private, common**, and **shared**. The private memory is used for instructions, stack, cache, etc. Private memory is only accessed by the processor *owning* it; the union of private addresses does not form an address space. The common memory repeats itself among each processor. A given location of common memory contains the same information in each module. Common memory contains program constants and other rarely changing global values. Common memory could be visualized as a single address space that is physically repeated among the processors. In contradistinction, shared memory forms a single address space that is physically split among the processors (See Figure 2-1). Thus, each processor has quick access to a portion of the shared memory (as well as slower access to the remainder). That portion is frequently referred to as its *local shared* memory, or — if no confusion arises — simply as its *local* memory. If a processor wishes to access a portion of shared memory that is not local to it, such processor must communicate through the interconnection network.
- The architecture under study presents two level of interconnections; those processors connected through the lower level are said to form a *cluster;* also by *cluster memory* the paper refers to the part of shared memory located within a cluster.
- The individual parts of the architecture are implementable in current technology.

The experiments measured

- the speedup obtained for six different application kernels with various processor/cluster combinations,
- the effect on speedup of varying the delays of interconnection network.

The graphs and charts in this paper illustrate the following statements: (1) application characteristics have a significant impact on obtainable speedup for any given combination of architecture and application; (2) system performance is very sensitive to interconnection delay.

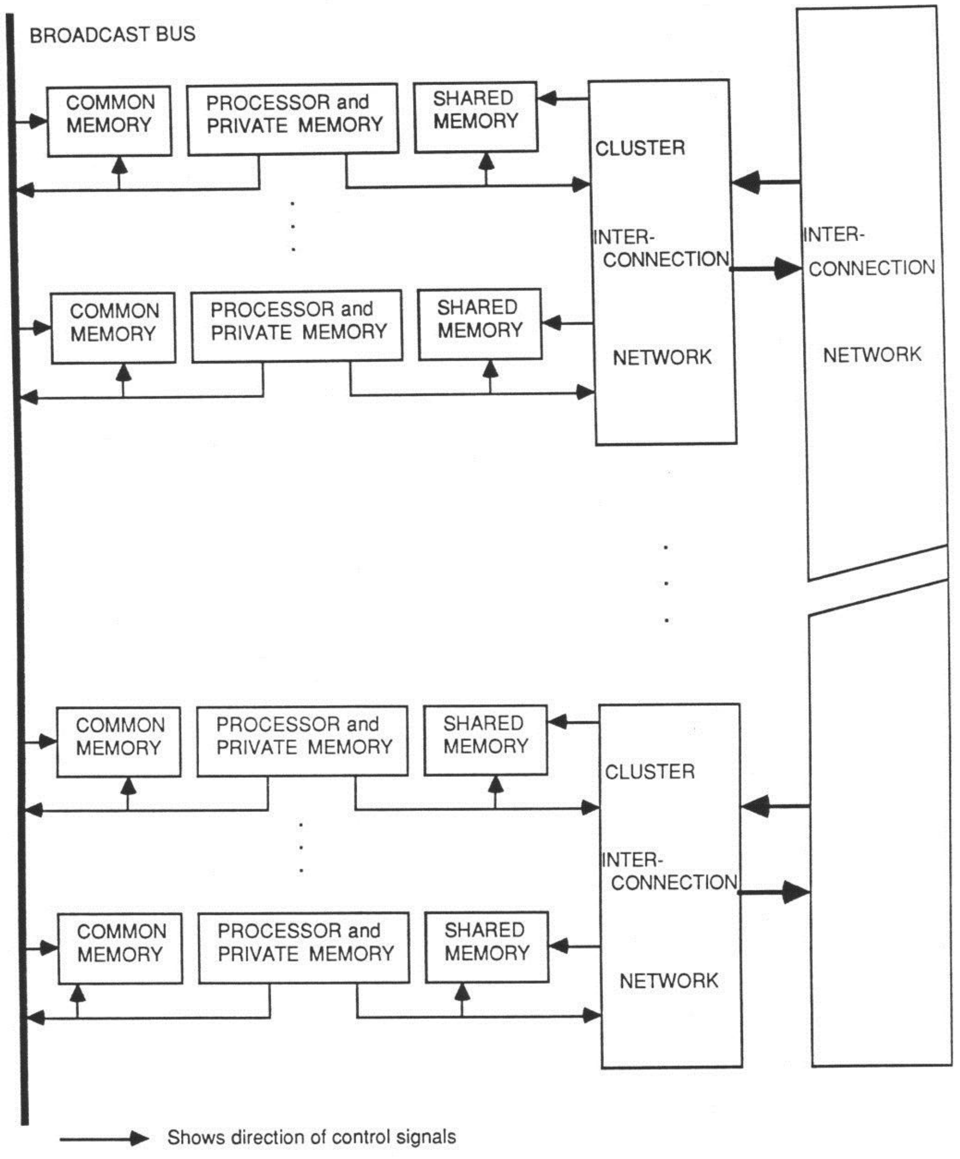
BROADCAST BUS
COMMON MEMORY
PROCESSOR and PRIVATE MEMORY
SHARED MEMORY
CLUSTER INTER-CONNECTION NETWORK
INTER-CONNECTION NETWORK
Shows direction of control signals

Figure 2-1: FutureLisp Cluster Architecture

2. BACKGROUND

This section describes the main characteristics of the architecture and applications used in the experiments.

2.1. Definitions

The following parameters are of use in discussing the power of a parallel processor, when executing a particular program:

Speedup: is the time required for the parallel processor (which contains **n** processors) to execute the whole job or program, relative to the time it takes *one* processor using the serial version of the program to do the same job. Note that the serial version of the program generally requires less execution time than the parallel version of the program executing on one processor. This difference is due to overhead required for parallel task management even when no parallelism results. Thus, the speedup is typically a number $\leq$ **n**.

Efficiency: is the speedup divided by **n**. Efficiency is a number ≤ 1. Generally, as **n** grows, the speedup grows, but the efficiency drops. The efficiency of a parallel program running on a parallel processor gives an indication of how well utilized are the processing resources *by such program*. The same parallel program running on a single processor can be used to estimate the overhead incurred by the parallel constructs.

Relative Execution Time: It compares the time taken by a program running on the actual system against the time taken by the same program running on an ideal system equal to the actual system but with no interconnection delays. It is ≥ 1.

Relative Execution Time = (time on real syst)/(time on zero-delay syst)
= (speedup on real syst)/(speedup on zero-delay syst)

The relative execution time (not speedup) of a given application running on a particular architecture is given by:

$$\text{Relative Execution Time} = 1 + C_o D_o + C_i D_i$$

where

C_o = out of cluster references / total cycles
C_i = in cluster references / total cycles
D_o = out of cluster delay
D_i = in cluster delay

Note that if the delays are zero, the size and diameter (how far the memory modules are) of the memory system are irrelevant.

Slowdown Factor: If the ideal speedup of a given program is divided by its real speedup, the slowdown factor is obtained.
Slowdown Factor = (speedup on zero-delay syst)/(speedup on real syst)

= 1/(Relative Execution Time)

The slowdown factor ranges between 0 and 1; the closer to 0 this factor, the greater is the loss due to the interconnection delay.

The above terms always define relations between a particular program and a particular architecture.

2.2. Architecture Summary

Figure 2-1 shows a diagram of the FutureLisp Cluster Architecture. The processors are connected through a two-level interconnection network. Each processor has three kinds of memory associated with it, ***private, common,*** and ***shared*** memory. The shared memory may be further categorized according to whether it is local to a processor (*local shared*), in the same cluster as a processor (cluster memory), or in a different cluster from a processor (non-cluster memory). Thus each processor sees five different types of memory.

No statements are made concerning the technology of hardware implementation except to assume that comparable technology is used throughout the system. Indeed, the performance figures are quoted in terms of master clock cycles, not microseconds. The emulator assumes that the processor is faster than the rest of the system, so that it may execute two typical operations in a single master clock cycle. With the master clock cycle set equal to that of a local shared memory reference time, the switch delay is specified in units of master clock cycles, thus allowing the emulator to function independently of implementation technologies.

The private memory is used for code and for execution stack data. It is accessible only by its associated processor. Two accesses to private memory may occur in a single master clock cycle.

Each processor has the whole contents of common memory immediately available for itself. Thus, read accesses to common memory are always immediate and require only a master clock cycle. Writes originating from a processor to its common memory also require a common memory cycle from the other common memories; thus no common memory is available to any processor during a write. The FutureLisp Emulator "charges" a master clock cycle to every processor, when common memory is written, in order to be conservative. In a real implementation, processors not requiring access to common memory will suffer no delay. This "cycle stealing" by common memory from the processors is done through the broadcast bus of Figure 2-1. Several processors simultaneously wanting to write to common memory will have their requests queued; each processor blocks until its write access is granted. In the applications studied, common memory was used for global variables and property lists. Writes to common memory were rare. Defining which data should be classified "read-mostly" in general is a open research topic.

The shared memory on each processor may be read by any processor in the system at varying costs. Access by the local processor typically takes one master clock cycle given that no conflicts occur. Access by other processors incur an interconnection switch delay in addition to the master clock cycle required for the read or write operation. This delay is smaller if the accessor is in the same cluster as the local memory being accessed (it will be called *delay for cluster memory reference*); it is larger if the processor belongs to another cluster (it will be called *delay for non-cluster memory reference*). A single shared memory module services one request per master clock cycle. If both local and non-local requests are present, it alternates their service.

The interconnection networks are assumed to be non-blocking with no contention. The communication delays for a cluster reference or non-cluster reference are system parameters that may be set independently for purposes of simulation. Each processor may have only one request to other than private memory outstanding at a time; that is, a processor request (read or write) to shared memory waits or blocks until that access is granted. When multiple requests for the same shared memory are made by different processors through the interconnection networks, the interface between the interconnection networks and the memory is assumed to queue requests from the interconnection networks that cannot be serviced immediately. No other assumptions are made about the structure of the interconnection networks.

There are four main parameters that characterize a given architecture:

1. Total number of processors
2. Number of processors per cluster
3. Delay for cluster memory reference
4. Delay for non-cluster memory reference

The number of processors could be set to any value, but the experiments generally used numbers between 1 and 256. The number of processors in a cluster could also be set to any number. However, to reduce the complexity of the experimental space and improve comparability, the number of processors in a cluster was set to 8 if the number of processors in the total system was less than or equal to 64, otherwise the number of processors in a cluster was set to 16. Besides being "reasonable" numbers, the interconnection systems to support these cluster sizes are feasible with current interconnection technology. Detailed study† beyond the scope of this paper would be required to determine optimal configurations for any given implementation technology and a particular program; the above numbers were chosen based on preliminary data.

The cluster and non-cluster delay parameters do not include the time required to access the shared memory. They refer only to the delay imposed by the interconnection network. Thus, if cluster delay were set to 2, a cluster memory access would require 3 master clock cycles if no other processors had pending memory accesses on the requested memory during the master clock cycle that the memory request arrived at the requested memory (2 to travel the cluster and 1 to access the memory). If the non-cluster delay were set to 6, then a non-

† Very small cluster sizes increase the percentage of slower out-of-cluster accesses; very large cluster sizes increase bottlenecks in that cluster. Thus, the detailed study should describe ways to allocate tasks and data to processors and memory. Section 2.5 "Limitations on Obtainable Speedup" touches some of these issues.

cluster shared memory access would require 7 master clock cycles if no other processors were accessing the requested non-cluster memory when the memory request arrived. In this way, it is possible for an experiment to have non-cluster delays relatively independent of cluster delays, instead of the more restricting rule

$$\text{non-cluster delay} = \text{cluster delay} + \text{outer-delay} + \text{cluster delay}$$

which Figure 2-1 indicates.

Table 2-1, "Default Parameters for Multiprocessor Configuration," shows the values for the default system configurations. Note that if default parameters are specified, the only independent variables are the number of processors and the application selected. When the number **n** of desired processors is not found in the table, use the row corresponding to $\lceil \mathbf{n} \rceil$. Thus, for 128 processors, use the parameters 128, 16, 2 and 7.

Number of Processors	Processors in a Cluster	Cluster Delay	Non-Cluster Delay
1	1	0	0
8	8	1	1
16	8	1	2
64	8	1	5
256	16	2	7
1024	32	3	9

Table 2-1: Default Parameters for Multiprocessor Configurations

The details of the processor used in the FutureLisp Emulator are not critical to these experiments. Briefly, it is stack-based machine with hardware assist for discriminating among the various data types ("tag bits") and for low-level scheduling operations. Each processor is assumed to have substantial private memory and enough registers to hold the context of the computation.

2.3. Methodology

Since the parameterized architectures of Table 2-1 (as well as any others) were supported by the FutureLisp Emulator, the experiments proceeded as follows:

1. Select suitable applications and data for the measurement study. The applications are selected to represent different aspects of the behavior of symbolic processing programs. See [McGehearty and Krall, 1986] for a discussion on the parallel execution of Common Lisp programs.
2. Modify the source code of the application program to execute in parallel in the FutureLisp Emulator. This step also included separation of the initialization routines from the body of the code to be measured. The initialization routines are omitted from the measurement phase to allow these studies to focus on the parallel kernel of computation. For completeness, a detailed

study focusing on the limits of parallelism in applications would need to study parallelism in the initialization phases as well.

3. Compile the application using the FutureLisp Compiler, built into the FutureLisp Emulator.
4. Invoke the FutureLisp Emulator on the object code, with a specific set of emulation parameters, often taken from Table 2-1.
5. Understand emulator output, and then show results and graphs.
6. Using the understanding from (5) above, modify emulator to measure specific information or to implement new architectural features. This suggests several iterations of points 1-6, as opposed to knowing in advance exactly what to measure; the architecture of Figure 2-1 was an act of evolution, not of creation.

2.4. Application Characteristics

The experiments use six application kernels, each having different characteristics relating to parallel symbolic computation. The application kernels were tested over several data sets, and a typical data set was selected for each application. While each of these programs is relatively small, varying from 60 to 600 lines of source code, they do cover a range of characteristics. Some of these characteristics are summarized in Table 2-2. A brief description of each application kernel can be found in Section 3.

Application	Independent Subtasks	Dependent Stages	Serial Time	Number of Tasks	Grain Size
Mandel	yes	no	3,160,000	2550	1400
Image	yes	no	940,000	128	7400
Rewrite	yes	yes	1,340,000	9040	157
Poly	yes	yes	970,000	5720	207
Resolve	no	yes	9,200,000	3640	2500
EMY	no	yes	66,000	187	425

Table 2-2: Application Characteristics

In Table 2-2, the "Independent Subtasks" column refers to whether a computation at the lowest level of parallelism is self-contained except for system wide constants once it has its initial values, or whether its behavior depends on other values that are being dynamically computed. Note that the rule sets in Rewrite, Resolve, and EMY are considered system constants since the rule sets do not change during the computation. The "Dependent Stages" column refers to whether there are stages in the computation which depend on earlier results. The "Serial Time" column contains the number of master clock cycles required to execute the serial version of the application (that version written without futures) on a single processor with all data local. The Number of Tasks counts how many executable tasks (in FutureLisp, an executable task is a Future) were created during the parallel execution of the application. The "Grain Size" column is the average size, measured in master clock cycles, of each task. Since a master clock cycle represents a local shared memory reference time, or the time for two private memory accesses (usually one for instruction fetch and other for operand fetch), it is an approximate measure of "RISC" instructions. (The latest RISC from Stanford University gives 1.2 clocks/RISC instruction+result, in the average [Agarwal]).

Table 2-2 does not show the parallel execution times; these are shown and discussed in Section 3. Note that the "Number of Tasks" multiplied by the "Grain Size" is larger than the "Serial Time". The difference is the overhead required to initiate the parallel tasks using the scheduler. If only one processor is used by the parallel versions of the programs, the program still incurs a measurable overhead of task creation and completion (Column "1 Pc Eff" of Table 3.1 shows the efficiency loss due to these overheads). This study compares the execution time obtained by the parallel version of the programs with that of the serial version to compute speedup.

A given application may not use all the processors available for its use. For example, on a system with 256 processors, the "Image" application could only use 128 of them since it only creates 128 tasks. In other words, there are some problems in which the parallelism is so shallow that some processors remain idle during the entire execution (See Figures 3-1a,b).

2.5. Limitations on Obtainable Speedup

Many features of an algorithm and architecture determine the effective speedup obtained. The most obvious cause of limited effective parallelism is a lack of tasks to keep the parallel processors busy. This limitation on speedup will be referred to as **processor starvation.**

A related problem arises when there are approximately as many tasks as processors, but the tasks do not map evenly to the processors. Either the tasks are of uneven size so that some processors finish early while others still have much work to do, or there are slightly more tasks than processors, so that some processors must execute two tasks while others execute one. Without prior knowledge about the execution requirements of the tasks, it might occur that the longest task is started last. This type of problem will be referred to as the **uneven load** problem.

Both processor starvation and uneven load can be alleviated by larger data sets in which the number of tasks is determined in part by the size of the data set. With more tasks, processors are not starved and the laws of large numbers tends to smooth out uneven task sizes [Lundstrom].

If the tasks created by a parallel application are small, then system overhead dominates useful computation. This limitation on useful task creation will be referred to as the **fine grain** problem. Solutions to the fine grain problem may conflict with solutions to the processor starvation and uneven load problems. Nevertheless, if the architect expects most of the tasks to be of fine or very fine grain size, then his system will likely be designed with a smaller system overhead.

Another application problem, called the **serial fraction**, occurs when a part of the application does not allow for parallel execution. For example, if twenty percent of an application's execution time must be performed serially, then more than five times speedup will never be obtained. This limit has been expressed previously as Amdahl's Law [Amdahl]:

$$\text{Maximum Efficiency} = 1/([S \times n] + 1 - S)$$

where:

n is the (total) number of processors
S is the fraction of code that is serial; 1-S is the fraction that is parallel.

The term serial fraction will also be used to cover the case when a portion of the application allows only a limited degree of parallel execution which is much less than the parallelism obtained in the main body of the application. This type of problem frequently occurs during task initiation and completion.

A problem related to the serial fraction is that of **dependent stages**. Some solution methods allow a limited amount of parallelism, followed by a sequential synchronizing step, then more parallelism followed by another synchronization step. Each additional stage contributes synchronizing overhead and increases the serial fraction. Further, limitations or slowdowns at any stage, which operate asynchronously, affect the entire pipeline. Thus a problem that has many dependent stages will be limited primarily by the least parallel or slowest stage and secondarily by the number of stages.

The **foreign reference** problem refers to tasks which require non-local or foreign data to execute. These references are considered part of the task creation overhead. The reason is that when a Future creates a task, this task usually runs in a processor different from the creator. Thus, the Future has to make foreign references to the data residing in the creator. The creator needs the value that the Future will produce; in addition, the creator may modify the data it is sharing with the new task; for these reasons, FutureLisp keeps these data (generally the arguments to the function created by the Future) local to the creator. The created task is thus penalized with foreign references each time it uses them.

In the current implementation of the FutureLisp Emulator, tasks assigned to a processor can not "migrate" to another processor, even if the processor it is assigned to has several tasks ready to run while a neighbor is idle. This specific condition can arise when a number of tasks are created each of which needs the result of some task which has not completed. Each task is assigned to a processor† and computes until it requires the result that is not yet ready. Then the task blocks and the processor becomes available for the next task. When the result is finally available, all of the tasks which have blocked on it become ready to run. However, if they happen to be assigned to the same processor, they will run sequentially, even if other processors are idle. This problem will be referred to as the **scheduling** problem.

† How this assignment is made and what problems are found is discussed below under the **task and data assignment problem.**

Some applications require an indeterminate amount of computation, depending on the order in which subportions of the application are executed. This type of application is frequently referred to as having "OR" parallelism. As will be seen in the experimental results section, this **indeterminate computation** problem interferes with architectural studies significantly. As an example of the problem, consider that search application may be implemented with a depth-first search strategy or with a breadth-first search strategy. In a sequential implementation, the order of search is strictly controlled, whereas a parallel implementation can put control of the order of search in the hands of the system task scheduler. Thus, the time to find an acceptable solution becomes dependent on the order in which tasks happen to be scheduled.

The **task and data assignment problem** refers to the criteria used by the run-time system environment to assign tasks and data to processors. In the examples presented here, a task was assigned to the first processor available. If no processor is available, the task was kept in a queue waiting for an available processor. In general, data frequently used by a processor should be located next to it, perhaps in its local shared memory. Tasks working closely together should be assigned to a single cluster. This study did not consider the issues involved when data and tasks are allowed to migrate.

While there are other types of problems which interfere with effective parallel execution, the ones discussed above cover the problems observed in the selected application set.

3. MEASUREMENTS

This paper studies six applications (3.1.1 to 3.1.6) against a variety of parallel architectures (given by Figure 2-1 and Table 2-1), where the main measurements seek to find out two important performance parameters: (a) the speed-up; and (b) the sensitivity of the applications to the interconnection delay.

3.1. The Speed-Up Potential in the Applications

As discussed in Section 2.4, "Application Characteristics," six applications were selected for this case study. Default parameters described in Table 2-1 were used. As Figures 3-1a and 3-1b "Application Speedup" show, the speedup obtained by each application varied widely. For the EMY application, additional processors beyond 16 provided little useful speedup. At the other extreme, Mandel shows continued speedup beyond 64 processors.

Table 3-1 shows additional information about each application. The column labeled "Number of Tasks" refers to the number of separately executable tasks in the application. The column labeled "Grain Size" is the average number of master clock cycles in each task. The column labeled "Task Suspends" refers to the number of times a task suspended waiting for another task to complete or for a semaphore to be unlocked. The column labeled "1 Pc Eff." states the efficiency of the parallel version of the application when running on a single processor as compared to the sequential version of the application. Finally, to simplify comparisons, the speedup obtained with 64 processors is listed for each application. The behavior of each application will be discussed in turn.

Application	Number of Tasks	Grain Size	Task Suspends	1 Pc Eff.	64 Pc Speedup
Mandel	2550	1400	51	0.99	59
Image	128	7400	2	0.99	40
Rewrite	9040	157	1958	0.92	33
Poly	5720	207	2409	0.82	25
Resolve	1785	2500	1660	0.30	23
EMY	187	425	279	0.83	9

Table 3-1: Application Behavior

3.1.1. Mandel

Mandel is the program which determines whether points on the complex plane are members of the Mandelbrot set [Dewdney 1985]. The computation is non-trivial. The number of points selected to be computed was a 50×50 grid in the complex plane. Mandel comes the closest to linear speedup, since the basic algorithm allows each point to be computed independently of every other point.

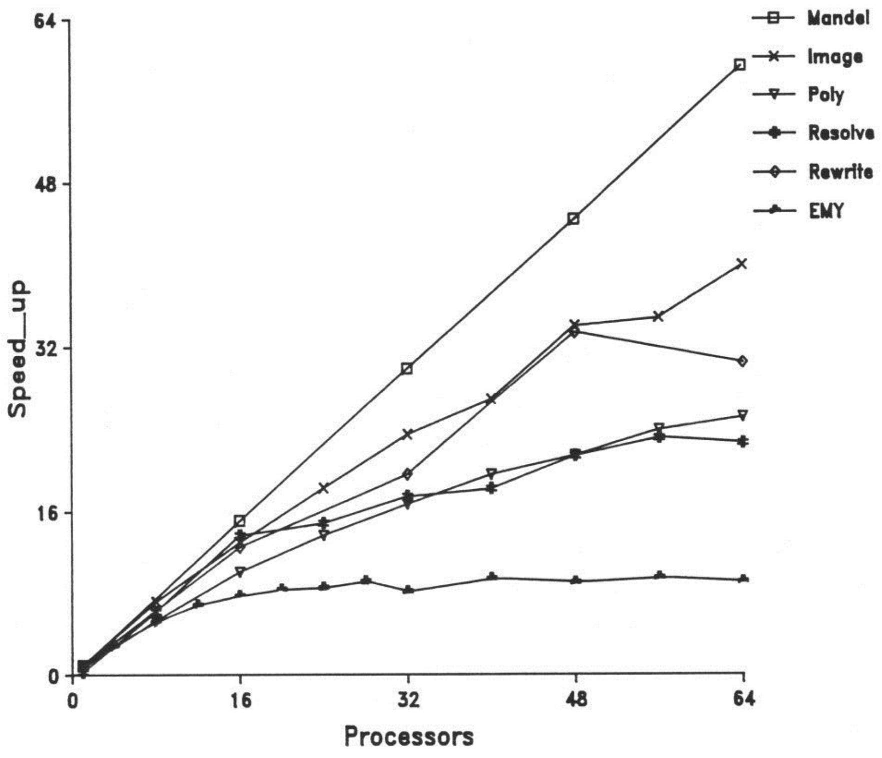

Figure 3-1a: Application Speedup for up to 64 Processors

The only communications required by each task are input of the initial point coordinates and output of the final boolean determination. Since fourteen hundred master clock cycles are required to make the determination for a typical point, and less than twenty-five are required for communication and task initiation, overhead is minimal. The test data reported here is for a 50×50 array of points. A startup task is created for each row in the array, which starts a subtask for each point on the row. Thus, fifty support tasks are created, each of which in turn create a total of 2500 working tasks. The working tasks then determine whether each point is in the Mandelbrot set. The number of tasks far exceeds the number of processors, thereby preventing processor starvation.

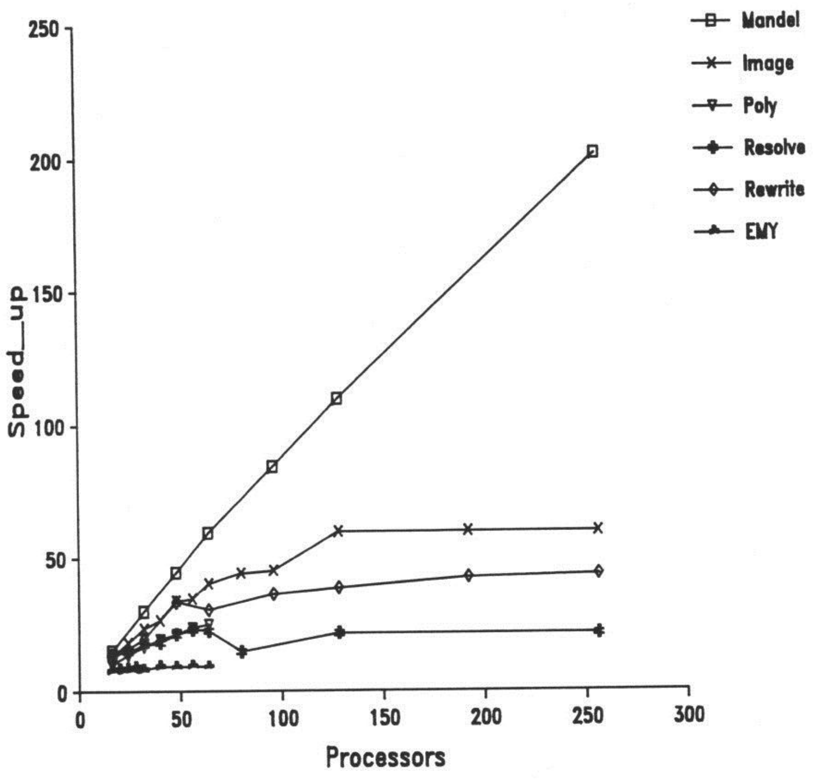

Figure 3-1b: Application Speedup for up to 256 Processors

3.1.2. Image

Image implements a simple gray level shift or scaling on a 128 $\times$ 128 image array. Because the amount of computation for each point is very small, each parallel task performs the smoothing operation for an entire row. This approach keeps efficiency high, but yields only 128 tasks to be executed, leading to the uneven load problem. Note that minimal increase in speedup is obtained between 64 and 96 processors in the Image program. With 64 processors, each processor handles two rows of the image. With 96 processors, each processor handles an average of one and one-third rows. However, the task size equals one row. So, some processors handle one row and some handle two rows. Since the computation is not complete until all rows are processed, the speedup is limited by the time to handle two rows, just as in the case for 64 processors. Some speedup is gained by spreading the array references over 96 memories instead of

64 memories. No speedup is gained after 128 processors, since there were only 128 tasks (each row was a task).

When the image size goes up to 256 $\times$ 256, slightly better speedups were obtained: the size of the task was larger (256 instead of 128, so there is less start-up overhead), and there are more tasks, thus allowing slightly better smoothing over the 64 processors. These results were not included in Figure 3-1b, since the figure refers to Image working on a 128 $\times$ 128 array.

Another problem for Image relates to startup activity. From the data in the previous section, it can be determined that 3000 master clock cycles are required to start 128 tasks. This time is a significant fraction of the 7400 master clock cycles required by the average task. Therefore, the last task starts to work 3000 cycles later than the first task. The same reasoning imply that the last task ends 3000 cycles later than the first task — but the total time spent by the multiprocessor is counted from the begining of the first task to the end of the last task —. Thus, if it were possible to start all 128 tasks in parallel, a significant amount of computer power could be put to use earlier and saved at the end. This behavior can be classified as a serial fraction problem. It is reasonable to speculate that breaking the startup task into parallel pieces could reduce the startup time, and further, breaking the computation of each row of tasks into several parts would ease the load balancing problem. The limiting factor on these approaches is the overhead of task initiation and completion.

Foreign references also affect the Image application. Like the Mandel application, the operation on each point in the Image application is independent of every other point. However, unlike the Mandel program, each task also requires the initial value of its point in the image. These points are stored in a two-dimensional array. Two types of memory references are required for operations on arrays in the current FutureLisp implementation. First, the array descriptor must be accessed to check array bounds and compute the linear index of the referenced value in the array from the index values supplied in the array reference. Then the appropriate array value is read or written depending on the type of memory reference required. For this experiment, the array descriptor is placed in common memory and the array itself spread over shared memory in an interleaved fashion. That is, the first word of the array is in the local shared memory of processor 1, the second word is in the local shared memory of processor 2, etc. Since each processor maintains a copy of common memory, no memory contention or non-local delays are incurred by the accesses to the array descriptor. With a single processor, the array data is also local, leading to very high efficiency. As the number of processors increases, accesses to the array data are more likely to be non-local, with associated delays. In the 64 processor case, this delay time accounts for approximately 15% of the execution time of a typical task. Solving this problem would require some method of associating the tasks for processing portions of an array with the processors that those portions of the array are stored in. This is a fine example of the task and data assignment problem.

3.1.3. Rewrite

Rewrite is a small theorem prover based on the Boyer-Moore theorem prover. It was adapted to FutureLisp from the version of Rewrite published in [Gabriel, 1985]. A short theorem was selected to be proven. Similar behavior was observed with other theorems, except that the maximum speedup observed was dependent on the complexity of the theorem to be proven. Rewrite was selected as a benchmark because theorem proving is a critical part of the planning process in many Artificial Intelligence programs.

Since over nine thousand tasks are created, processor starvation and uneven load are not the limiting problems for this application. However, each task is quite small, so the fine grain problem comes into play. Fine grain overhead accounts for an 8% loss of efficiency in Rewrite.

Another problem is in the area of dependent stages. The parts of the initial theorem to be proven are rewritten several times before the final proof checking step occurs. Before a given part of the theorem can be rewritten, the previous rewriting of that part must be complete. In the final stage, the rewritten theorem is checked. This stage has the least available parallelism and acts as a further limit on the speedup obtained.

The variation observed in the speedup of Rewrite in the range from 48 to 128 processors is due to the indeterminate computation problem. In the sequential implementation, all rewriting is completed before any checking is performed. In the parallel implementation, checking begins as soon as any rewriting has completed. Some optimizations exist which allow portions of the rewritten theorem to be ignored by the checking process. Since these portions are not needed by the check phase, it does not matter whether they complete their rewriting operation. Thus, if they happen to be scheduled before the required portions, total execution time is greater than if they happen to be scheduled after the required portions of the rewriting and checking process.

3.1.4. Resolve

Resolve is a resolution theorem prover. It was included in the benchmark as another type of theorem prover, because it uses "OR" parallelism to search the tree of possible proofs. Since its method of computation is considerably different from Rewrite, it provides an alternate view of theorem proving.

Resolve is strongly affected by the indeterminate computation problem. Its basic method of operation is "generate and test," combining previous solution attempts (clauses) to be checked as future solutions. When it combines the right set of clauses, it determines that the theorem to be proven is correct. If the theorem is not correct, it will never terminate. The order in which new clauses are added to the trial list greatly affects the time required to achieve a solution. If a complex clause which does not lead to the solution is added to the trial list, many related clauses will also be added to the trial list. The computation related to these subclauses detracts from the desired computation. This effect is so strong for this application that it overrides other problems.

3.1.5. Poly

Derived from the MACSYMA program widely used for symbolic mathematics, Poly provides symbolic manipulation of polynomials. The data set selected causes two large polynomials to be multiplied together. Other data sets yield varying amounts of speedup which are proportional to the complexity of the polynomials in a non-linear manner.

Like Rewrite, it tends more toward the fine grain problem than the processor starvation problem. In particular, task creation and communication take approximately 18% of the total execution time.

It also suffers from the problems of dependent stages and serial fraction. Polynomial multiplication has four stages. First, the terms of one polynomial are distributed over the other polynomial for multiplication. Second, the multiplication occurs. Third, the terms of the same degree for each variable in the polynomial are matched. The fourth stage adds these terms to compute the final result. The first and third stage tend to be serial in nature thus limiting the obtainable parallelism.

3.1.6. EMY

EMY implements a kernel of the EMYCIN expert system. EMYCIN is a backward chaining expert system. The rule set selected, called fevers, is based on disease diagnosis expertise; the specific disease to be diagnosed was rheumatic fever. More information about EMY may be found in [Krall and McGeheartу, 1986].

The primary problems with obtaining speedup with EMY are dependent stages and serial fraction. EMY contains a concept known as parameters. The value of a parameter is determined by executing rules relating to that parameter. After all such rules have completed execution, conclusions about the value of a parameter are made. Note the implicit sequentiality in this process. A further serialization may be caused by executing a rule which requires that the value of another parameter must be determined. Then another set of rules must be executed and conclusions made about the new parameter before the value of the first parameter is known. The parallelism is limited by the number of active rules, which is between 60 and 95 in the chosen test data.

However, there is another sequential step relating to the termination of the rules associated with a single parameter. Each rule must access a shared value sequentially. The combination of dependent stages and serialization are the primary limits on the speedup of the EMY application. Adding more processors will not result in a significant speedup since there are not enough parallel tasks to keep the processors busy.

3.1.7. Summary of Application Behavior

While each application differs in its parallel behavior, there are some common trends. Serial fractions and dependent stages of computation cause the most severe limits in speedup. The overhead associated with fine-grain tasks causes a loss of efficiency, which reduces speedup by a constant factor. Non-local references can also reduce speedup. Note that the parallel versions of the applications were carefully modified to minimize non-local references, and the default architecture assumes near-optimal interconnection performance to minimize the cost of non-local references. The final problem in measuring speedup relates to the problem of indeterminate computation. This report does not have a solution to this problem, but will discuss its effects where they are significant.

It is important not to make predictions about likely speedups to be obtained in real systems over broad ranges of applications from the limited data presented here. The mix of behaviors and parallel algorithm problems will vary widely from one application to another. Rather, these application kernels are intended to be used to identify common styles of programming and potential weaknesses of parallel architectures and their implementations in supporting these programming styles.

3.2. Switch Delay Study

The next aspect of the architecture to be studied was the interconnection delay characteristics of the system communications. Varying the interconnection delay provides a sensitivity analysis of each application with respect to the interconnection delay. This analysis is useful for the following reasons:

1. Implementation of the interconnection system with different component technologies or topologies yields different average delay times for the system. By examining a range of values for the delay, one may view how the choice of underlying implementation can alter overall response times of applications. In addition, one could analyze the cost versus performance properties of the system as faster switches are incorporated into the design, and extrapolate the system performance for various delays.
2. The application sensitivity can be analyzed by observing the effect of interconnection delays on execution time of different problems. Hence, one can evaluate how sensitive a given problem is to different communication delays.

In the results discussed in this section, the number of processors was fixed at 64 processors, with 8 processors in each cluster. To provide a regular comparison of increasing interconnection delay, the cluster interconnection delay was chosen as the independent variable. The inter-cluster interconnection delay was set to be three times the cluster delay [Recall from Section 2.1 that in addition to these, there may be some further delays queuing at the target memory]. This value is justified by two observations: (a) a non-cluster reference must traverse two cluster interconnects and the system interconnect (See Figure 2-1), and (b) the inter-cluster interconnect ties together some number (eight) of processors, while the inter-cluster interconnects ties together the same number (eight) of clusters. Other system parameters were kept at the default values.

Figure 3-2 shows the speedups (for 64 processors) of the various application kernels for the selected range of cluster interconnection delays, while Figure 3-3 gives the "Slowdown Factor" of the applications, dividing their speedups for a given delay by their speedups when the delay is zero (See Section 2.1). The closer

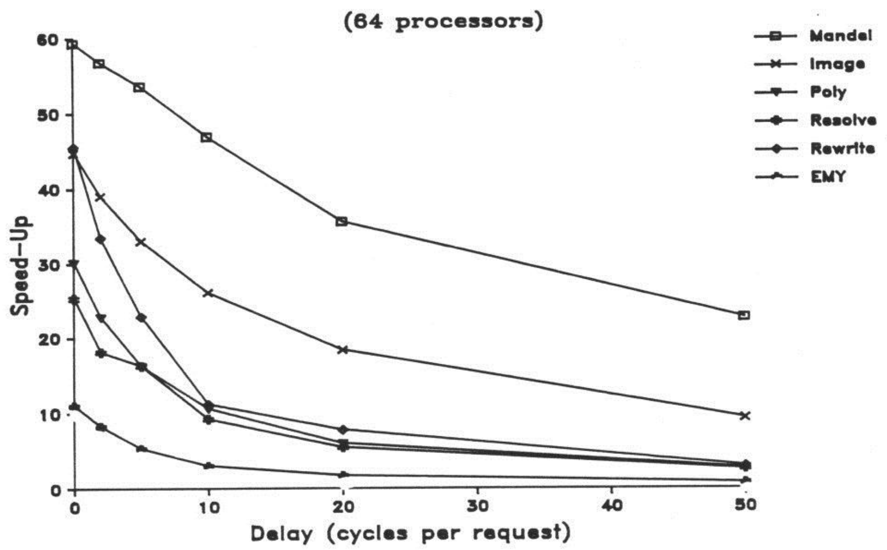

Figure 3-2: Effects of Interconnection Switch Delay on Speedup

to zero this slowdown *factor* is, the worse the speed loss due to the interconnection delay is.

The loss of speedup in the various applications is dependent on the effect of locality. As the percentage of the non-local references (see Table 3-2) increases, the delay graphs tend to take on a sharper slope. The Mandel program with less than 1% non-local memory references was least sensitive to interconnection delay. The image application showed the second best results with 3% non-local references. The application programs which have approximately 10% non-local memory references (Rewrite, Poly, and EMY) constitute the worst delay graphs in the experiments. Indeed, with the cluster interconnection delay set at 50, they show almost no speedup when executing in parallel on 64 processors.

While some applications are much more sensitive than others to interconnection delay, large interconnection delay, in all cases, causes a substantial reduction in performance. Therefore, interconnection delay is a critical system parameter for applications designed to execute on the class of architectures studied here. If interconnection delay is large, then major redesign of applications and algorithms may be necessary to achieve useful levels of performance improvement.

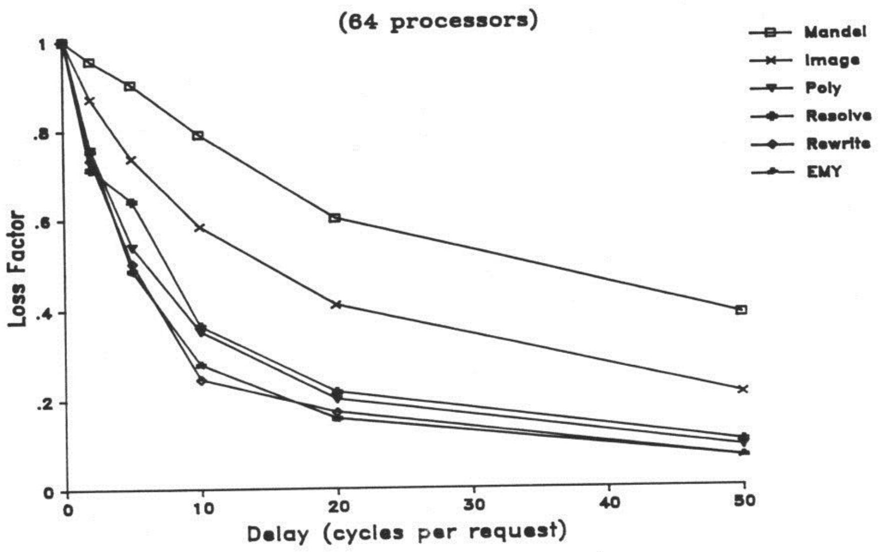

Figure 3-3: Slowdown Factor as a Function of Switch Delay

Application (64 processors)	Fraction of Data References	
	Cluster	Non-Cluster
Mandel	0.005	0.004
Image	0.004	0.026
Rewrite	0.073	0.053
Poly	0.049	0.048
Resolve	0.015	0.043
EMY	0.029	0.061

Table 3-2: Application Effects on Locality of Data

4. Discussion of Results

4.1. Implications of Application Behavior on Architecture

The application experiments demonstrate that application style and algorithm technique influence realizable parallel speedup. These influences are summarized in Table 4-1.

Application	Salient Features	Critical System Requirements	Observable Parallelism
Mandel	independent computation	none	great
Image	independent computation, but shared data	rapid network	large
Rewrite	search algorithm with fine grain tasks	fast scheduling	moderate
Poly	shared data fine grain tasks	rapid network fast scheduling	moderate
Resolve	OR-parallelism algorithm	runtime priority methods	moderate
EMY	synchronization fine grain tasks	rapid network fast scheduling	limited

Table 4-1: Algorithm and Architecture Interactions

Applications with fine grain tasks require support for fast scheduling to prevent scheduling overhead from dominating useful computation. Without such support, their algorithms must be redesigned to increase the grain size of their tasks. In some cases, such redesign results in too few tasks being created to provide a smooth load to the entire system. In summary, the efficient support for fine grain tasks reduces the programming burden on algorithm design, parallel compiler support, and load balancing schedulers.

Those applications which communicated frequently among their various tasks showed rapid loss of performance when the cost of communication was high. Such high communication costs would place heavy burdens on algorithm designers to minimize communication among subtasks.

Some classes of symbolic applications have significant components of indeterminate computation. Frequently several paths of computation are pursued and the partial results on one path can be used to optimize the computation on another path. In addition to requiring efficient communication methods, these applications might benefit from methods of adjusting the priority of different subtasks dynamically during a computation. No such method is currently available in the system used for these measurements.

4.2. Summary

The results presented show that both application characteristics and architectural features place limits on achievable parallel performance. In particular, the graphs and charts in this report illustrate the following statements:

1. Application characteristics have a significant impact on obtainable speedup for any given combination of architecture and application.
2. System performance is very sensitive to interconnection delay.

There are of course other factors that either contribute to or detract from optimal parallelism. Other researchers have investigated the basic architecture of the system, the scheduling disciplines, and specific interconnection networks. Such features are orthogonal to those reported here.

5. BIBLIOGRAPHY

Agarwal, A., Chow, P., et al. "On-Chip Instruction Caches for High Performance Processors." In *Advanced Research in VLSI* (Proceedings of the 1987 Stanford Conference), P. Loslebel (ed). MIT Press, 1987.

Amdahl, G. M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings 1967 Spring Joint Computer Conference,* 483-485.

Dewdney, A. K. "Mandelbrot Sets" (in the column "Computer Recreations"), *Scientific American*, July 1985.

Gabriel, Richard. *Performance and Evaluation of Lisp Systems*, The MIT Press, Cambridge, 1985.

Halstead, Robert. "Implementation of Multilisp: Lisp on a Multiprocessor," *ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

Krall, E. and McGehearty, P. "A Case Study of Parallel Execution of a Rule-Based Expert System", *International Journal of Parallel Programming*, Volume XV, Number 1, February 1986.

Lundstrom, Stephen F. "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," submitted to *IEEE Trans. on Computers.*

McGehearty, P. and Krall, E. "Potentials for Parallelism of Common Lisp Programs," *Proceedings of the Sixth International Conference on Parallel Processing*, St. Charles, Illinois, August 1986.

Steele, G. *et al.* *Common Lisp*, Digital Press, Hanover, Ma., 1984.