MCC Technical Report Number: ACA-262-87

The Effect of Application Characteristics on Performance in a Parallel Architecture

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

August 7, 1987

A shared memory, hierarchical, and clustered architecture is reported here. Measurements presented (including speedups) were obtained by running six different symbolic application kernels, with varying architectural parameters. The results and graphs shown 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, cost scheduling and indeterminate computation with respect to the applications chosen.

These measurements allow understanding of some of the interactions that take place between the architectural parameters and the specific applications, and suggest ways for improving of the parallel architecture. So far, the architecture has provided reasonable speedups on the chosen applications.

1.1. FutureLisp

FutureLisp is a dialect of Common Lisp [Steele], augmented by the addition of *futures* [Halstead]. A *future* returns a pointer to the eventual value of its argument: 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 (briefly described in Section 3) 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 architecture under study presents two levels of interconnections, with processors connected through the lower level forming a *cluster.* By *cluster memory* the paper refers to that part of shared memory located within a given cluster.
- Each part of the architecture is implementable with current technology.

The experiments measured the speedup obtained for six different application kernels with various processor/cluster combinations. The graphs and charts in this paper illustrate the following statement: application characteristics have a significant impact on obtainable speedup for any given combination of architecture and application.

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**. The 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.

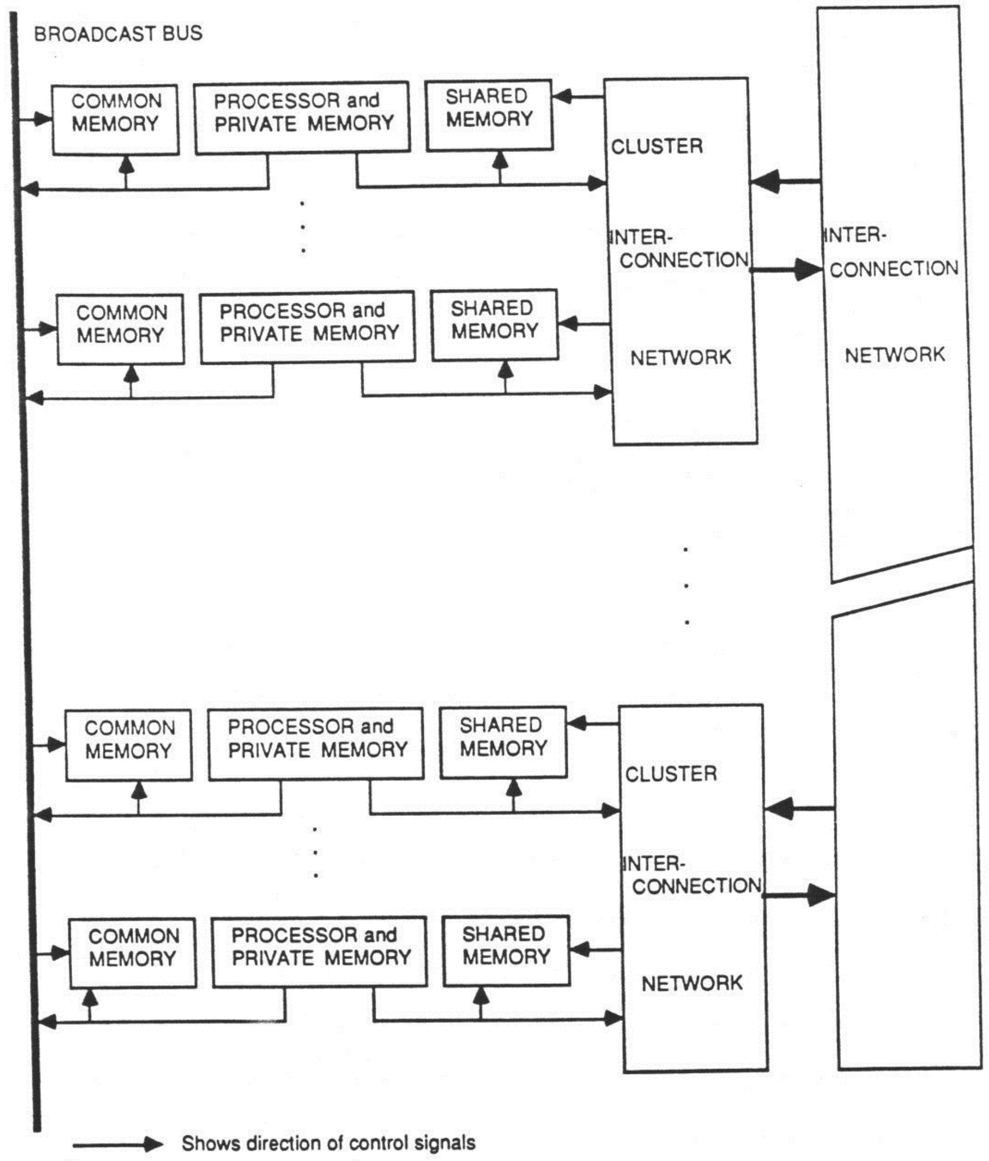
BROADCAST BUS
COMMON MEMORY
PROCESSOR and PRIVATE MEMORY
SHARED MEMORY
CLUSTER INTER-CONNECTION NETWORK
INTER-CONNECTION NETWORK
Shows direction of control signals

Figure 2-1: FutureLisp Cluster Architecture

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 **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 is larger if the processor belongs to another cluster. A single shared memory module services one request per master clock cycle. If both local and non-local requests are present, the module alternates their service.

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 interconnection networks are assumed to be non-blocking with no con-

tention. 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 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. Application Characteristics

Each of the six application kernels used (See Section 3) had different characteristics (summarized in Table 2-1) relating to parallel symbolic computation. They 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.

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 2-1: Application Characteristics

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. The rule sets in Rewrite, Resolve, and EMY are considered system constants, because they 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.

Table 2-1 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 (Shown in column "1 Pc Eff" of Table 3-2).

2.4. Limitations on Obtainable Speedup

The effective speedup attained depends on many features of both algorithm and architecture. The most obvious cause of limited effective parallelism is a lack of tasks to keep the parallel processors busy. This limitation will be referred to as **processor starvation:** a given application may not use all the processors available, due to lack of enough parallelism in the application. For example, on a system with 256 processors, the "Image" application could only use 128 of them since it only creates 128 tasks (See Figures 3-1a,b).

A related problem arises when there are approximately as many tasks as processors, but the tasks do not map evenly to the processors. For instance, the tasks may be of uneven size so that some processors finish early while others still have much work to do. Or there may be 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].

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, and 1-S is the fraction that is parallel. By extension, 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 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 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.

If the tasks created by a parallel application are small, then system and creation overhead dominates useful computation. This will be referred to as the **fine grain** problem. Solutions to it 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.

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 (see below under the task and data assignment problem) and computes until it requires a 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.

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.

Some applications require an indeterminate amount of computation, depending on the order in which subportions of the application are executed. These applications are frequently referred to as having "OR" or "search" parallelism. As will be seen in Section 4, this **indeterminate computation** problem interferes with architectural studies significantly. For instance, consider that a search application may be implemented with a depth-first 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.

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), where one of the main measurements was to ascertain the speed-up potential of the selected applications.

The applications for this study use the default parameters listed in Figure 3-1.

Processors	Processors in Cluster	Delay within cluster	Delay out of cluster
1	1	0	0
2 .. 8	8	1	1
9 .. 16	8	1	2
17 .. 64	8	1	5
65 .. 256	16	2	7
257 .. 1024	32	3	9

Figure 3-1: Default Parameters

These cluster sizes and delays are reasonable and are further discussed in [Guzman *et al*].

3.1. The Speed-Up Potential in the Applications

The speedup obtained by each application (see Figures 3-1a and 3-1b) varied widely. For the EMY application, additional processors beyond 16 provided little useful speedup. At the other extreme, Mandel shows continued speedup beyond 256.

Table 3-2 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 3-2: Application Behavior

The details of the application analysis may be found in [Guzman *et al*]. The applications are briefly described here, however.

3.1.1. Mandel

Mandel is the program which determines whether points on the complex plane are members of the Mandelbrot set [Dewdney]. 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.

3.1.2. Image

Image implements a simple gray level shift or scaling on a 128×128 image array. Because the amount of computation for each point is very small, each

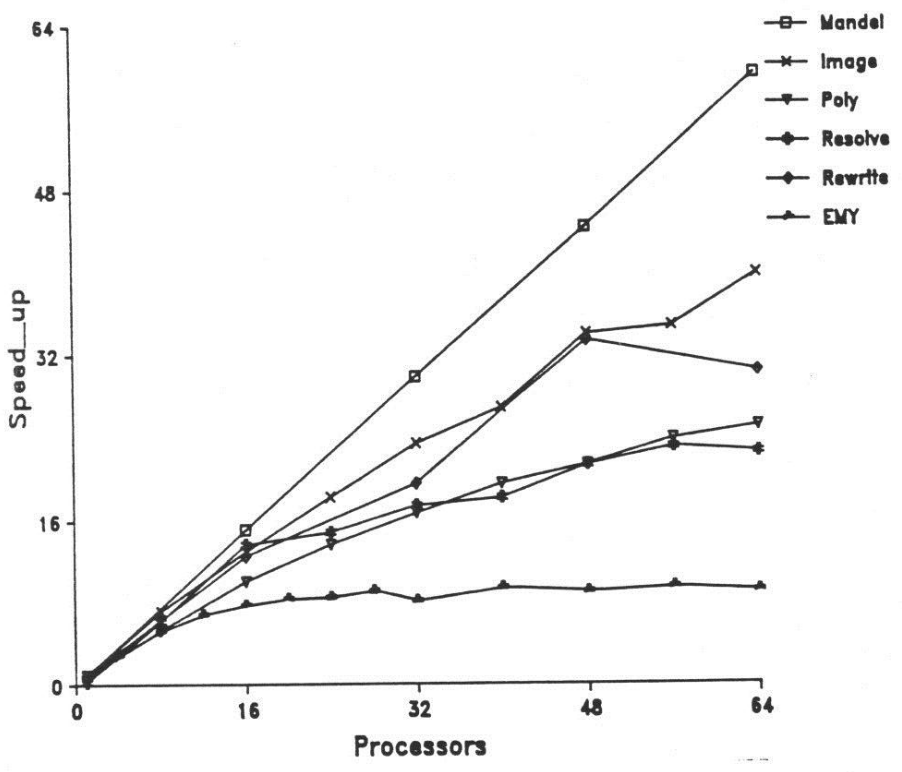

Figure 3-1a: Application Speedup for up to 64 Processors

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.

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

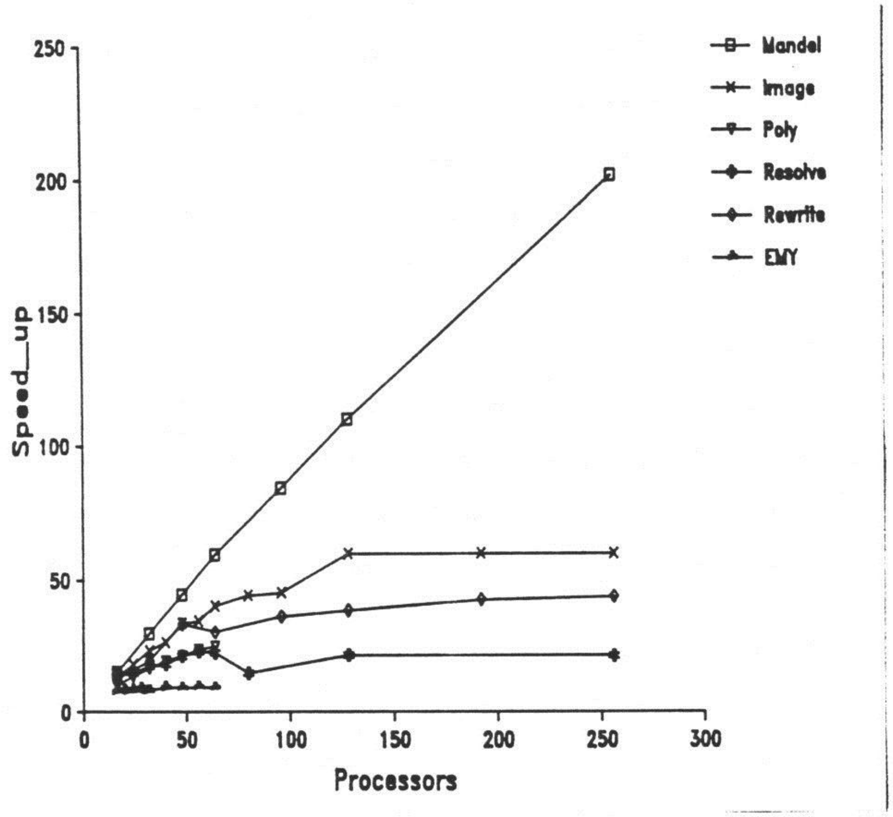

Figure 3-1b: Application Speedup for up to 256 Processors

[Gabriel]. 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.

3.1.4. 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.

3.1.5. 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.

3.1.6. EMY

EMY implements a kernel of EMYCIN, 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 in [Krall and McGehearty].

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.

4. DISCUSSION OF RESULTS

4.1. Influences of Application Behavior on Architecture

The application experiments demonstrate (See Table 4-1) that application style and algorithm technique influence attainable parallel speedup.

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.

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

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. Other Effects Observed

System performance is very sensitive to interconnection delay too, but the results are reported elsewhere [Guzman *et al*].

Very small cluster sizes increase the percentage of slower out-of-cluster accesses; very large cluster sizes increase bottlenecks in that cluster. Further studies are needed to adequately quantify these effects.

4.3. 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 show that application characteristics have a significant impact on obtainable speedup for any given combination of architecture and application.

There are of course other factors that influence the degree of parallelism: basic architecture of the system; scheduling disciplines; specific interconnection networks, and delays. Such features are orthogonal to those reported here.

5. BIBLIOGRAPHY

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.

Guzman, A., Krall, E., McGehearty, P., and Bagherzadeh, N. "Performance of Symbolic Applications on a Parallel Architecture," MCC Technical Report PP-163-87, Austin, Texas. Also submitted to *International Journal of Parallel Programming.*

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*

[redacted]

Processing, St. Charles, Illinois, August 1986.

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

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.

4. DISCUSSION OF RESULTS

4.1. Influences of Application Behavior on Architecture

The application experiments demonstrate (See Table 4-1) that application style and algorithm technique influence attainable parallel speedup.

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.

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

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. Other Effects Observed

System performance is very sensitive to interconnection delay too, but the results are reported elsewhere [Guzman *et al*].

Very small cluster sizes increase the percentage of slower out-of-cluster accesses; very large cluster sizes increase bottlenecks in that cluster. Further studies are needed to adequately quantify these effects.

4.3. 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 show that application characteristics have a significant impact on obtainable speedup for any given combination of architecture and application.

There are of course other factors that influence the degree of parallelism: basic architecture of the system; scheduling disciplines; specific interconnection networks, and delays. Such features are orthogonal to those reported here.

5. BIBLIOGRAPHY

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.

Guzman, A., Krall, E., McGehearty, P., and Bagherzadeh, N. "Performance of Symbolic Applications on a Parallel Architecture," MCC Technical Report PP-163-87, Austin, Texas. Also submitted to *International Journal of Parallel Programming.*

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.