On the Importance of Parallel Application Placement in NUMA Multiprocessors

Timothy Brecht[†]
Department of Computer Science, University of Toronto
Toronto, Ontario, CANADA M5S 1A4
`brecht@cs.toronto.edu`

Abstract

The thesis of this paper is that scheduling decisions in large-scale, shared-memory, NUMA (Non-Uniform Memory Access) multiprocessors must consider not only how many processors, but also which processors to allocate to each application. We call the problem of assigning parallel processes of an application to processors *application placement*.

We explore the importance of placement decisions by measuring the execution time of several parallel applications using different placements on a shared-memory NUMA multiprocessor. The results of these experiments lead us to conclude that, as expected, in small-scale mildly NUMA multiprocessors, placement decisions have only a minor affect on the execution time of parallel applications. However, the results also show that placement decisions in large-scale multiprocessors are critical and localization that considers the architectural clusters inherent in these systems is essential. Our experiments also show that the importance of placement decisions increases substantially with the size and NUMAness of the system and that the placement of individual processes of an application within the subset of chosen processors also significantly impacts performance.

---

† Author's current address: Department of Computer Science, York University, 4700 Keele Street, North York, Ontario, CANADA M3J 1P3 email: `brecht@cs.yorku.ca`

This paper appears in the Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), San Deigo, CA, September, 1993, pp. 1-18

1. Introduction

Small-scale, shared-memory multiprocessors based on a single shared bus have become prevalent and the number of manufacturers building and selling such systems continues to rise. The success of such systems can be partially attributed to the simple parallel programming model they present, relative to strictly non-shared-memory systems. This simple programming model has allowed many applications to achieve substantial increases in performance by making effective use of all of the processors in the system.

The considerable improvements in parallel application performance attained using small-scale multiprocessors have fueled the desire for even greater performance improvements. One approach to increasing performance is to simply build larger systems while maintaining the shared-memory model. Single-bus systems, however, are not scalable because the bandwidth of the bus limits their size. As a result research and design efforts in shared-memory multiprocessors have focused on scalable architectures. These architectures distribute memory modules throughout the system in order to optimize access times to some memory locations. The result is an important class of scalable shared-memory systems known as Non-Uniform Memory Access (NUMA) multiprocessors. Alternatively, all memory accesses could be made uniform, but then they would be uniformly slow.

The emergence of large-scale, shared-memory multiprocessors presents a number of new opportunities and challenges. The opportunities are to solve much larger problems than previously possible, with applications that use more processors, and to solve many problems concurrently, by simultaneously executing multiple parallel applications. The challenges are to effectively utilize the processors while enabling the efficient execution of multiple applications. The multiprogramming of parallel applications is required because not all applications will be capable of effectively utilizing all processors in a large-scale system.

An obvious but critical difference between scheduling in this new class of NUMA (Non-Uniform Memory Access) multiprocessors and small-scale UMA (Uniform Memory Access) multiprocessors, is that in UMA systems all processors can be treated equally (aside from cache contexts). This is because in a UMA system the time to access any memory location is the same from any processor. NUMA system designers must, however, consider the time it takes to access different memory locations from different processors. Therefore, an important aspect of scheduling in large-scale, shared-memory, NUMA multiprocessors is application placement. That is, how should the parallel processes of an application be placed in a NUMA multiprocessor?

This paper shows that in large-scale, shared-memory, NUMA multiprocessors the execution time of a parallel application is directly related to which processors it executes on. As a result, efficient and effective placement decisions become critical to processor scheduling and overall system performance. In fact, it is likely to be a contributing factor in ultimately determining the success or failure of large-scale NUMA multiprocessors.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 describes three existing shared-memory NUMA multiprocessors along with their architectural and NUMA characteristics. This is followed in Section 4, by a description and explanation of the importance of localization during application placement. Section 5 describes the environment and Section 6 the applications used in the experiments. In Section 7 we present the results of a series of experiments designed to demonstrate the the importance of localization. The paper is concluded in Section 8 with a summary of the results, conclusions and a brief discussion of future work.

2. Related Work

As microprocessor technology continues to improve at a faster rate than memory or interconnection network technology, the relative increase in communication costs in multiprocessors has become a topic of increasing importance. A number of recent studies consider the importance of memory access costs when making scheduling decisions in shared-memory multiprocessors. An important and common goal of this research is to reduce the number of cache-misses or remote memory references by co-locating lightweight threads or kernel processes with the data being accessed, thus reducing the time spent loading data into the local cache or memory.

Using an analytic model of a time-sliced, central ready-queue, scheduling environment and experimental evaluation on a UNIX based multiprocessor, Squillante and Lazowska [11] [10] argue and demonstrate that applications can build considerable cache context, or footprints [13]. Recognizing that it may be more efficient to execute a process on a processor that already contains relevant data in that processor's cache, they design and examine techniques that consider the affinity a process has for a processor. They observe that the performance of applications, which release a processor because of quantum expiration, preemption, or I/O, can be significantly reduced by making use of processor-cache affinity information.

Subsequent studies have arrived at different conclusions. Gupta, Tucker, and Urushibara [5] also consider the importance of cache-affinity techniques but in a space-shared multiprocessor environment. They simulate a number of scheduling techniques and, using processor utilization as a performance metric, conclude that improvements due to processor-cache affinity are quite small, improving mean processor utilization by only 3%. Vaswani and Zahorjan [16] draw similar conclusions in their study of the importance of cache affinity. They also suggest, using an analytic model, that even with faster processors and larger caches the benefits due to cache affinity will be minimal.

The apparent difference between the conclusions drawn in these studies is largely due to the difference in the scheduling policies used to multiprogram applications. While Squillante and Lazowska use time-sharing, the study by Gupta, Tucker, and Urushibara and the study by Vaswani and Zahorjan both use space-sharing. The time-sharing policy employs a small reallocation interval. This results in relatively frequent context switches and ensures that processes do not run long enough to interfere with each other significantly. Vaswani and Zahorjan found that with space-sharing the frequency of context switches is reduced and that intervening applications ran long enough to significantly disrupt the cache context of the previous process, thus greatly reducing the benefits of processor-cache affinity.

While cache affinity studies investigate the benefits of reusing cached data when executing more than one application on the same processor (across applications), related work at the University of Rochester concentrates on the benefits of reusing cached data when executing lightweight threads of the same application on the same processor (within applications). The Rochester work also extends the notion of locality management to include one more level in the memory hierarchy by considering systems that may have local-cache, local-memory and remote-memory, such as the BBN TC2000. Markatos [7] first demonstrates that fine-grain parallel programs, because of the overhead required to load data into the local cache, or memory, typically perform much worse than coarse-grain implementations even though the cost of thread management is negligible. This motivates the need for techniques that consider locality when scheduling lightweight threads within an application. Markatos then develops a technique called memory-conscious scheduling which, when used with fine-grain applications, yields execution times that are comparable to coarse-grained implementations.

Markatos and LeBlanc [8] consider the conflicting requirements for load balancing and locality management when scheduling lightweight threads of an application. They conclude that of the two important considerations, locality management should be the primary factor influencing the assignment of threads to processors. The importance of locality management is also explored in their work on loop scheduling [9]. They demonstrate how traditional loop scheduling techniques incur significant performance penalties on modern shared-memory multiprocessors. They then propose and compare new loop scheduling algorithms that consider the requirements of load balancing, minimizing synchronization, and co-locating loop iterations with the data being referenced. These new algorithms are shown to improve performance by up to 60% in some cases.

Our work is complementary to processor-cache affinity and lightweight thread scheduling techniques for improving locality of data references. While these previous studies investigate the importance of scheduling techniques for reducing the number of non-local memory accesses by co-locating processes with the data being accessed, our work investigates the importance of scheduling techniques for reducing the cost of required non-local memory accesses in environments where processes and data cannot be co-located. We have conducted a preliminary simulation study [18] which indicates that placement is an important aspect of scheduling in large-scale, NUMA multiprocessors and has motivated the need for experimentation on a real NUMA multiprocessor.

In this paper we experimentally investigate the problem of scheduling parallel processes of an application that concurrently access shared data in an environment in which there is no a priori knowledge of sharing or communication patterns. The complexity of the problem is increased by the architectural trend to cluster processors and memory elements and to connect clusters together in a hierarchical fashion in order to build larger systems. This results in systems with a number of levels in the memory hierarchy and memory access latencies that vary with the number of levels of the hierarchy that must be traversed. Therefore, the placement problem becomes one of placing processes of an application onto processors such that the costs of required accesses to shared data are minimized.

3. Scalable Shared-Memory Multiprocessors

Examples of three existing scalable shared-memory multiprocessors are the KSR1, from Kendall Square Research [2], DASH, developed at Stanford University [6], and Hector, developed at the University of Toronto [17]. Each of these systems incorporates a hierarchical design to build larger systems by using small-scale multiprocessor components as building blocks. In Hector and DASH the base component is essentially a bus-based multiprocessor containing a small number of processors (they are called stations and clusters, respectively). In the KSR1 the base component, called Ring:0, is a unidirectional ring connecting up to 32 processors. Both the KSR1 and Hector use a ring to connect base components together to form larger systems. The Hector design provides for another level in the hierarchy by connecting a collection of rings together with what is called a global ring. The DASH system uses a mesh interconnection network to connect base components together. The processing modules in the KSR1 and Hector, besides containing a processor and associated cache, also contain local processor memory, which is used to further optimize access times to some memory locations and reduce contention for the base component interconnection network. In DASH each cluster is essentially a Silicon Graphics multiprocessor which does not contain localized processor memory but instead contains a secondary shared cache. The processor used in the KSR1 is a 20 MHz RISC processor developed by Kendall Square Research. DASH uses the 33 MHz MIPS R3000 processor while Hector uses the 16.67 MHz Motorola MC81000.

System and CPU	Memory Level	Memory Location	Processor Cycles	Approx. System Size
KSR1	1	Local Memory	18	1
	2	Ring 0	126	32
	3	Ring 1	600	32–1088
DASH	1	Secondary Cache	15	1
	2	Local Bus Memory	29	4
	3	Remote Cluster Memory	132	16–64
Hector	1	Local Memory	19	1
	2	On Station Memory	29	4
	3	On Ring Memory	37	16
	4	Off Local Ring Memory	46	256

Table 1.1: Memory reference hierarchies and latencies of some NUMA multiprocessors

Table 1.1 shows some memory latency times in processor cycles for each of these systems. The times for the KSR1 are in 50 nano-second cycles and are the times required to read one 128 byte cache line [3]. DASH and Hector have 30 and 60 nano-second cycle times respectively and the latencies shown in Table 1.1 are for loading one 16 byte cache line [6] [12]. This table illustrates two of the key issues related to the use of shared-memory NUMA multiprocessors:

1) The time to access remote memory can be significant.
2) The time to access remote memory depends on the distance to the location being accessed (the number of levels of the hierarchy that must be traversed).

It is therefore quite natural to hypothesize that placing the parallel processes of an application close to each other in order to reduce communication costs will be essential for their efficient execution. The degree to which the execution time of an application will benefit from a localized placement depends on the number, frequency and latency of remote communication.

4. Application Placement for Localization

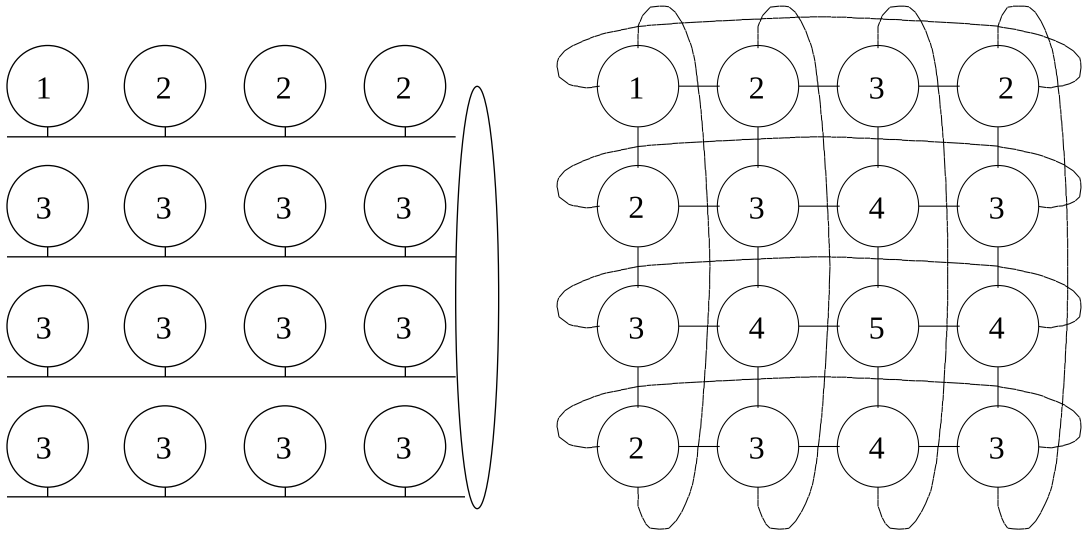
In this section we describe how a scheduler might choose a ‘‘localized’’ subset of processors on which to execute an application. Fortunately, most scalable shared-memory architectures adhere to a hierarchical design and as a result determining a ‘‘localized’’ subset of processors is not difficult. Note that processes of an application must be placed individually, since we are assuming a dynamic scheduling environment in which there is no a priori knowledge of the number of processors an application will be allocated.

From any one processor, remote memory accesses can have successively higher and higher costs as the distance from the requesting processor increases. These costs can be thought of as forming a hierarchy of levels, where the access time from a given processor to any memory module within the same level is the same. If from each processor we define M_l to be the time to access memory at level l in the hierarchy and $l = 1,2,3,...,L$, then:

$$M_1 < M_2 < \cdots < M_l < \cdots < M_L .$$

One method of building large-scale, shared-memory multiprocessors which is currently popular is to connect processors in a clearly hierarchical fashion, as is shown is Figure 1.1a. This is the type of interconnection scheme used in the DASH and Hector systems. Figure 1.1b is an example of an alternative interconnection scheme that is not strictly hierarchical by design. It is presented as an example of how memory access times can be organized into a hierarchical structure when viewed from individual processors.

Figures 1.1a and 1.1b assume that the system is built from processor/memory pairs. The labels indicate the level of the memory access hierarchy that each processor/memory pair belongs to. Labels are assigned relative to the specified source processor, which is labelled and belongs to level 1. If we assume that the first process of an application is placed randomly on the processor/memory pair labelled 1 a localized placement places the next process of that application on any one of the processor/memory pairs labelled 2, since they can all be accessed from the first processor with the same latency. Future placements for the same application continue to use processor memory/pairs labelled 2, until they are all used, at which point level 3 is used. The levels in Figure 1.1a adhere to the hierarchical structure of the system while the levels in Figure 1.1b are determined by simply counting the number of hops required to reach each processor/memory pair from the first processor.



(a) (b)

Figure 1.1: Memory access hierarchy in two multiprocessor designs

A rough classification of memory reference times, such as the hierarchical view just discussed, provides for localized processor allocations by choosing processors from the lowest level possible, relative to the first processor chosen, then moving up the hierarchy once all processors at the current level have been allocated. This is repeated until the number of desired processors has been obtained.

5. The Experimental Environment

The experiments presented here were conducted using a prototype of a scalable shared-memory NUMA multiprocessor called Hector, developed at the University of Toronto [17]. Each processing module in the Hector prototype contains a 16.67 MHz Motorola MC81000 CPU, a 16 Kbyte instruction cache, a 16Kbyte data cache, and 4 Mbytes of globally addressable memory. The hierarchical design used in Hector connects a number of processing modules with a bus to comprise a station, several stations are connected with a bit-parallel slotted ring, and rings can be further connected using a hierarchy of rings to easily support up to 256 processors. The prototype used consists of 4 stations, each containing 4 processor modules, for a total of 16 processors and 64 Mbytes of globally addressable memory. There is no hardware support for cache coherence, thus permitting a simple and elegant design that has relatively mild NUMA characteristics. Cache coherence is enforced in software, by the HURRICANE operating system's memory manager at a 4Kbyte page level of granularity, by permitting only unshared and read-shared pages to be cacheable [14] [15]. HURRICANE also supports page migration and replication but these features were disabled in order to conduct these experiments.

The parallel processes comprising each application are placed on processors in the system by a system scheduler. This is accomplished by having each HURRICANE process or thread creation first contact a user-level scheduler to determine which processor to execute on. The scheduler is designed for a dynamic multi-purpose, multiprogrammed environment so it is assumed that the desired number of processors is not known a priori. As a result processor requests and placement decisions occur one at a time, at the time of process creation. The scheduler was configured in such a way that the desired placements were obtained. Applications were linked with a special library that directs creation calls to the scheduler and notifies the scheduler whenever a process is finished executing.

6. The Applications

Since the current Hector system is a prototype, and because much of the work being conducted on the system consists of operating systems research and performance evaluation, there is not a large body of regular users. As a result the applications collected for experimentation consist mainly of applications that were written either to evaluate this type of research or as part of a course project on parallel programming. Consequently some of the applications used are really kernels of what would be considered real parallel applications.

All of the applications are of the data parallel or single program multiple data class of applications, which means that each process executes the same computational kernel on different portions of the data space. The data access patterns of each application are different, so the importance of the placement of parallel processes of the application should vary with each application. Since the placement of each parallel process of an application is important relative to the data that is being accessed, the HURRICANE operating system permits the application writer to roughly control where data will be located by specifying the policy to be used when requesting memory. In the applications used most of the shared data is allocated to memory according to a first-hit policy. That is, data will be physically located in the memory of the processor module that first touches the page containing that data. Some applications specify a round-robin policy for some of the shared data so that frequent access of the data by many processors reduces the likelihood of hot-spots and also reduces remote memory access costs when executing on all 16 processors. Even though we do not use all of the processors in the system we have not modified the memory allocation policies used by the applications (currently there is no policy for allocating memory on a round-robin basis from the subset of processors assigned to the application).

For each application the main process creates a number of children which act as slaves. Since each process of the application is allocated to a separate processor and we do not want processors to be unnecessarily idle, some applications were modified so that the master process not only controls and synchronizes the children but also performs its share of the computation, rather than simply waiting for the children to perform the computation.

The applications are listed in Table 1.2 along with the problem size, precision used, the number of lines of C source code, and the speedup measured using four processors of one station, S(4). The speedup values shown were computed by comparing the execution times of the parallel application using one and four processors (since a serial version was not available for all applications). The number of source code lines may be slightly high due to the large number of timing, tracing, and debugging calls used when tuning the applications. More detailed descriptions of each application can be found in [1].

Name	Application / Problem Size	Precision	Lines of C	S(4)
FFT	2D Fast fourier transform 256x256	Single	1300	2.9
HOUGH	Hough transformation 192x192, density of 90%	Double	600	3.4
MM	Matrix multiplication 192x192	Double	500	3.4
NEURAL	Neural network backpropagation 3 layers of 511 units, 4 iterations	Single	1100	3.8
PDE	Partial differential equation solver using successive over-relaxation 96x96	Double	700	3.7
SIMPLEX	Simplex Method for Linear Programming 256 constraints, 512 variables	Double	1000	2.4

Table 1.2: Summary of the applications used

The size of the system used is relatively small, and in order to evaluate different application placements, each application is executed using four processors (four processors was also chosen because some applications constrained the number of processors used to a power of two or to a number that divides evenly by the size of the data set used). Although the size of the data sets may appear to be small, they were chosen for a number of reasons:

1) They should execute on four processors in a reasonable amount of time since multiple executions of each application are used to compute means and confidence intervals.

2) The size of the data cache on each processor is relatively small (16 Kbytes). Consequently cache misses and memory accesses will occur, even with a relatively small sized problem.

3) The amount of memory currently configured per processor is relatively small (4 Mbytes). If problem sizes are too large data structures that are designed to be allocated to the local processor (by using the first-hit allocation policy) may have to be allocated to a different processor, resulting in remote memory references where the application programmer had not intended. That is, once all of the physical memory of the local processor has been allocated, the memory manager will allocate memory from a neighbouring but remote module.

The seemingly poor speedup of some applications is the result of the small data sets used to perform these experiments, since most of the applications were designed to be used with larger data sets on more processors (*i.e.* the parallelism is relatively coarse-grained).

7. Impacts of Placement on Performance

In order to examine the importance of localization in shared-memory NUMA multiprocessors we conduct a series of experiments using the Hector multiprocessor and six parallel applications (or application kernels) each executing on four processors. The main purpose of these experiments is to determine the importance of application placement. That is, the importance of localization. The experiments are conducted by running each application in isolation under different placement strategies. The execution times of the localized placement are then compared with the non-localized placements.

The prototype Hector system used to conduct the experiments is configured with sixteen processors. To avoid interference caused by system processes and the workload generator, thus ensuring that differences in execution times are due solely to different placements, we dedicate one station (the four processors in Station 0) to their execution. That is, only stations 1, 2, and 3 are used to execute the applications being tested. Figure 1.2 illustrates a localized and a non-localized placement of four processes of an application and the notation used to represent these placements. In the localized placement, the four dashes ‘‘−−−−’’ above Stations 0, 1, and 2 indicate that the four processors in each of these stations are not used, while the numbers ‘‘1234’’ above Station 3 indicate where each of the four processes of the application is placed. The first process (1) being the main (master) process of the program and the remaining three (2, 3 and 4) being the child (slave) processes. The placement is localized because all four processes of the application execute within one station and the notation is ‘‘−−−− −−−− −−−− 1234’’. The non-localized placement spreads the processes across the twelve processors being considered (as mentioned previously, Station 0 is not used, in order to to avoid interference with system and workload generating processes which are restricted to that station). The first process executes on Station 3, the second and fourth on Station 2, and the third on Station 1 and the notation is ‘‘−−−− −−−3 −−42 −−−1’’.

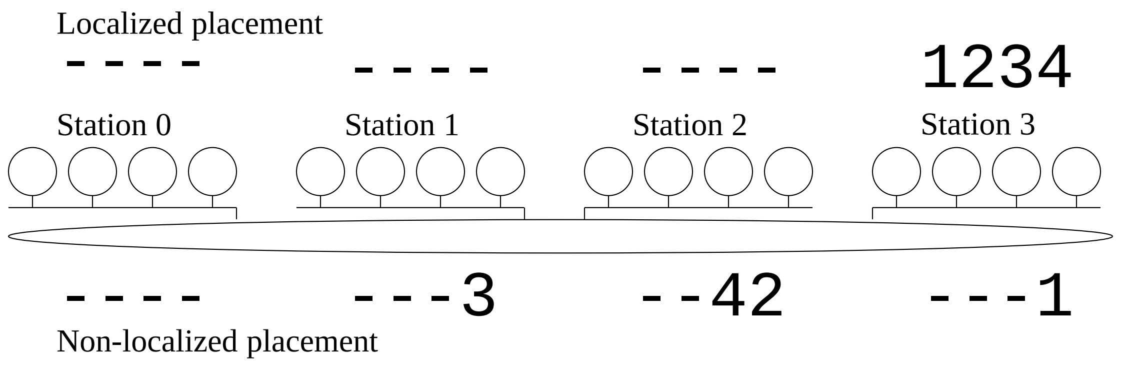


Figure 1.2: Localized and non-localized placements

Figure 1.3 shows the normalized mean execution times of the six applications when executed using the localized and non-localized placements. This graph along with the detailed results in Table 1.3 show that localized application placement does improve the execution time of some of the applications examined. The table was constructed by executing the applications eight times for each placement. The table contains the mean execution time (Mean) and 90 percent confidence intervals (CI), for each of the placements, as well as the improvements obtained by using the localized placement (% Impr). This is given as the percentage by which the mean execution time was improved by using the localized placement rather than the non-localized placement, expressed as a percentage of the mean execution time of the non-localized placement. Times are measured in seconds.

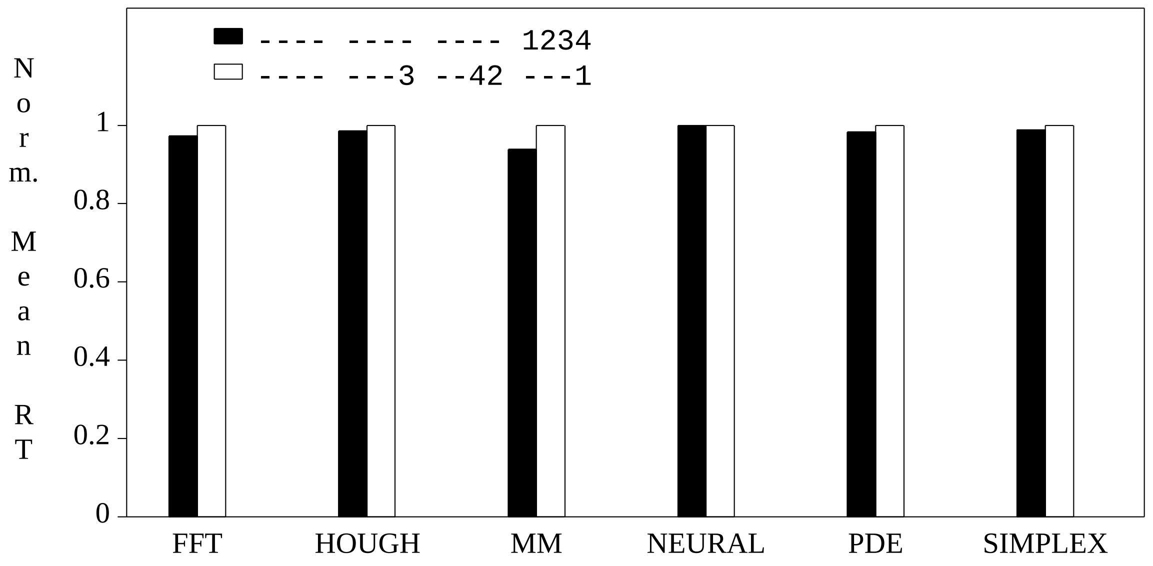


Figure 1.3: Normalized execution times using localized and non-localized placements

Notice that improvements obtained here are not large. This is due to relatively small-size and mild NUMA structure of the prototype sixteen processor Hector system. We hypothesized that, under a heavy multiprogrammed workload, contention for shared-resources such as the interconnection network (ring) would be reduced under a localized placement, resulting in even greater benefits. However, preliminary experimental results indicate that, under the multiprogrammed workloads tested, contention is not significant (further experimentation with different workloads is ongoing). The remainder of the experiments are therefore conducted in a uniprogrammed setting.

Appl	Localized		Non-Localized		% Impr
	Mean	CI	Mean	CI	
FFT	4.84	0.00	4.71	0.00	2.7
HOUGH	5.01	0.00	4.94	0.01	1.4
MM	5.25	0.01	4.93	0.01	6.1
NEURAL	4.83	0.01	4.83	0.00	0.0
PDE	5.45	0.00	5.36	0.01	1.7
SIMPLEX	19.27	0.06	19.06	0.07	1.1

Table 1.3: Mean execution times, in seconds, using localized and non-localized placements

The following sections consider larger systems, systems with different architectures, and future multiprocessors, by studying the effects of NUMAness on the importance of localization.

7.1. Increasing Memory Latencies

The NUMAness of a system can be thought of as the degree to which memory access latencies are affected by the distance between the requesting processor and the desired memory location. It is determined by:

- The differences in memory access times between each of the levels in the memory access hierarchy.
- The number of processors that can be accessed in the time determined by each level.
- The number of levels.

In order to study the effects of changes in the NUMAness of the system, Hector features a set of switches, called delay switches, that add additional delays to off-station memory requests. The range of settings possible are: 0, 1, 2, 4, 8, 16, 32, and 64 cycles. Every packet destined for a memory module not located on the same station is held up at the requesting processor for the number of selected cycles. The delay switches are used to emulate and gain insight into the performance of:

1) Larger systems — since increases in the system size will result in increased memory latencies.
2) Systems of different designs — because some systems have larger memory latencies due to the complexity of the interconnection network or hardware cache coherence techniques.
3) Future systems — because processor speeds continue to increase at a faster rate than memory and interconnection networks.

	32bit load	32bit store	cache load	cache writeback	Delay
local	10	10	19	19	
station	19	9	29	62	
ring	27	17	37	42	0
	35	21	49	58	4
	43	25	61	74	8
	59	33	85	106	16
	91	49	133	170	32
	155	81	229	298	64

Table 1.4: Memory reference times, in processor cycles, on a 16 processor Hector system

Table 1.4 shows latencies for local, on-station, and off-station (or ring) memory accesses in units of 60 nano-second cycles. Off-station requests, or those requiring the use of the ring are shown for 0, 4, 8, 16, 32 and 64 cycle delays. The values shown are pessimistic values in the sense that the true values depend on the relative positions of the source and destination stations, and the values shown represent worst case relative positioning. This is because even though the system is symmetric, asymmetry is introduced, since cache line reads consist of one request packet but two reply packets (in order to return the entire 16 byte cache line). Note that the delay switches have no affect on local or on-station requests. For more detailed descriptions of the Hector see [4] [17] [12].

To provide insight into the importance of localization on a slightly larger system and in other shared-memory multiprocessors we set the delay switches to 16 and conduct the same localized versus non-localized placement experiment. The results of this experiment are shown in Figure 1.4. Note that with a delay of 16 cycles the sixteen processor Hector system used has memory access latencies that are roughly equivalent to other existing shared-memory multiprocessors.

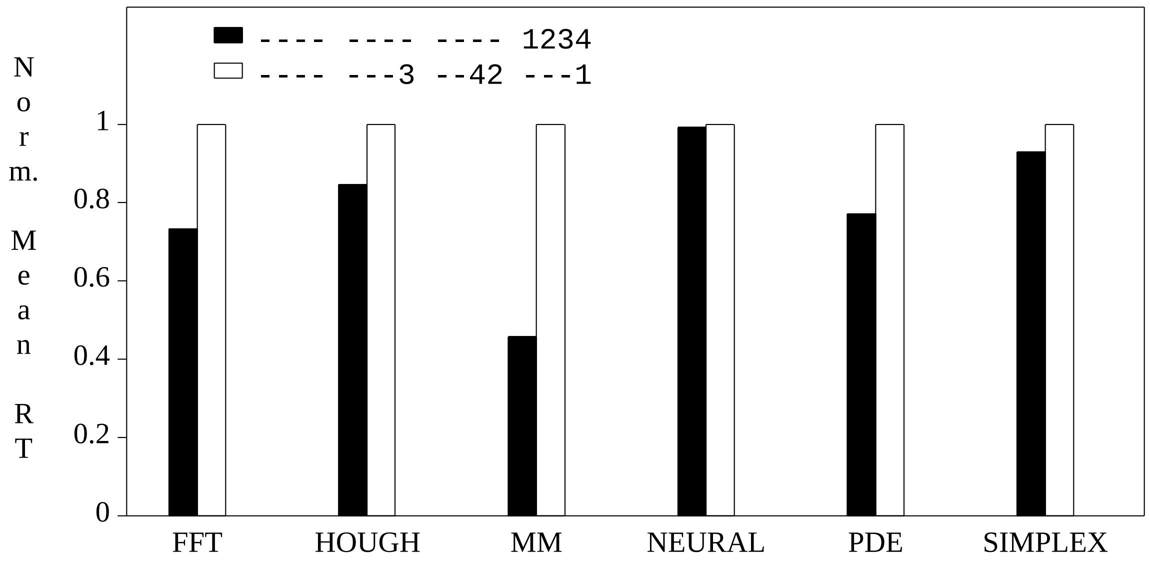


Figure 1.4: Normalized response times, localized and non-localized placements, delay = 16

The experimental results of Figure 1.4 show substantial improvements in execution times as a result of localization. The matrix multiplication application (MM) is improved by more than 50%, while the fast fourier transformation (FFT) is improved by more than 25%, the partial differential equation solver (PDE) more than 20%, and the hough transformation (HOUGH) by more than 15%. Only the neural network application is not significantly improved. The reason for this is an unusually large number of system calls (a more detailed explanation is provided in a subsequent section).

7.2. NUMAness

Figure 1.5 illustrates the affects that the NUMAness of the system has on the execution of these applications under non-localized and localized placements. The graphs show the normalized execution times of each application obtained with delay settings of 0, 4, 8, 16, 32 and 64. The delay setting is shown just below the pair of bars representing the localized and non-localized execution times.

These graphs demonstrate that as the latencies in the system increase the performance benefits achieved through localization increase for all applications except NEURAL. If the communication and memory references within an application are completely localized then the increase in latencies should have no affect on the execution times when the localized placement is used. The results show this to be true for MM and PDE. Note however, that this is not the case for FFT, HOUGH, SIMPLEX and especially NEURAL.

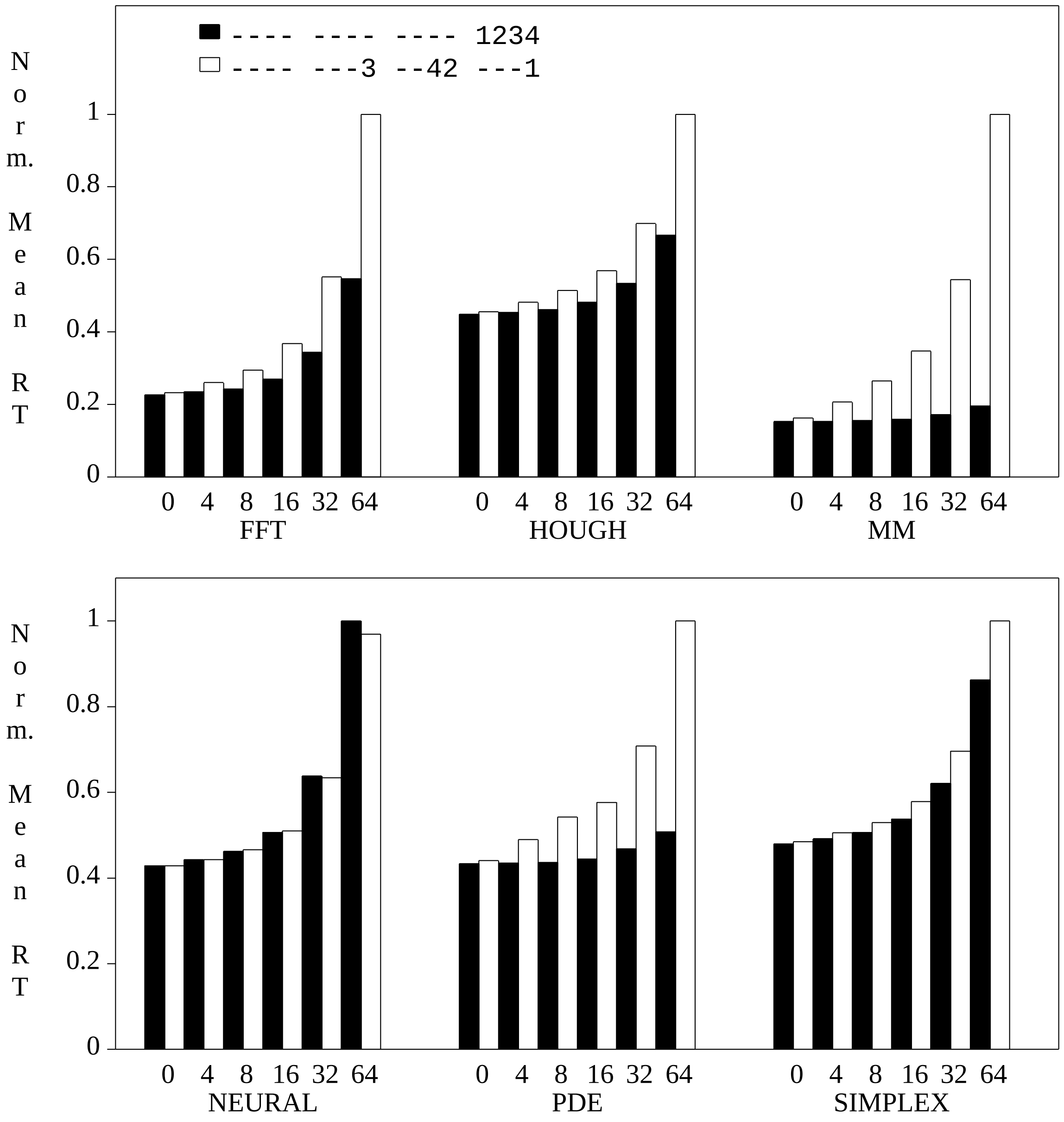


Figure 1.5: The importance of localization using different degrees of NUMAness

There are a number of reasons why, under a localized placement, some of these applications are more affected by increased latencies than others.

1) The data being accessed is not entirely localized. That is, some data is located on memory associated with processors outside of those the application is executing on. This is true, for example, of FFT. Because normally the computation is dominated by sine and cosine computations pre-computed lookup tables of sine and cosine values are created. For each of these tables, the memory allocation scheme used assigns pages to physical memory on a round-robin basis starting with processor 0 on station 0. In this case, because 64 bit double precision variables are used with a problem size of 256, two 4Kbyte pages will be allocated for each table and references to these tables may require remote memory accesses. The round-robin allocation of these tables was done by the original author in order to achieve

good performance, by reducing hot spots, when using all 16 processors. For the same reasons HOUGH also uses pre-computed lookup tables for sine and cosine values, which along with the input image, are allocated in a using a round-robin page allocation policy.

2) There are a relatively high number of system calls.

   a) Some system calls are performed by communicating, via message passing, with a server process that is executing on Station 0, thus incurring delays because of remote communication. This is reflected in increases in execution time as the delays increase because communication with Station 0 requires using the interconnection ring which means incurring the delays. This is the case with the SIMPLEX application.

   b) Some system calls are handled by a server process that is migrated to the processor of the calling process (handoff-scheduling). The server may then access system data structures, many of which have been allocated on Station 0, thus requiring off-station memory requests which increase execution times as latencies are increased. The application NEURAL performs a large number of such system calls which is the reason that the performance is not improved by using a localized placement. In fact, localization actually degrades performance in this case because the algorithm used executes synchronously, with each of the processes requiring access to shared resources at the same time. The non-localized placement decreases the degree to which the processes are synchronized and decreases the contention for shared resources, thus slightly improving the execution times.

7.3. System Size

Another way to view the importance of application placement is to consider possible increases in system size and the different application placements possible, given a fixed number of required processors. For example, if an application requires four processors and it is executed on a system with four processors a localization strategy is not required since any placement is localized. The potential benefits of localization increase in an eight processor system but are not as large as the benefits that can be obtained in much larger systems. That is, if the number of processors allocated to an application is fixed and different sized system are considered, the potential benefits from localization and therefore the importance of localization increase with the size of the system. This is illustrated in Figure 1.6. The different placements used correspond to considering localized versus non-localized placements in systems of 4, 8 and 12 processors. The localized placement, ‘‘−−−− −−−− −−−− 1234’’ is the same for each system size, while the placement ‘‘−−−− −−−− −−43 −−21’’ represents a non-localized placement in a system of 8 processors, because only the 8 processors of Stations 2 and 3 are considered, and the placement ‘‘−−−− −−−3 −−42 −−−1’’ represents a non-localized placement in a system of 12 processors. Therefore, each of the bars of the graphs in Figure 1.6 represent a non-localized placement in systems of size 4, 8 and 12 and should be compared with the localized placement ‘‘−−−− −−−− −−−− 1234’’ to determine the improvements possible due to localization for a system of that size.

The results of these experiments demonstrate that, for all applications except NEURAL, the benefits obtained from using a localized placement increase as the size of the system is increased, thus demonstrating the need for and increased importance of localization in larger and larger systems. Note also that the prototype system being used is relatively small and as a result the performance of each placement is also affected by the number of processors being used by the application (four). This can be seen by the small difference between the non-localized

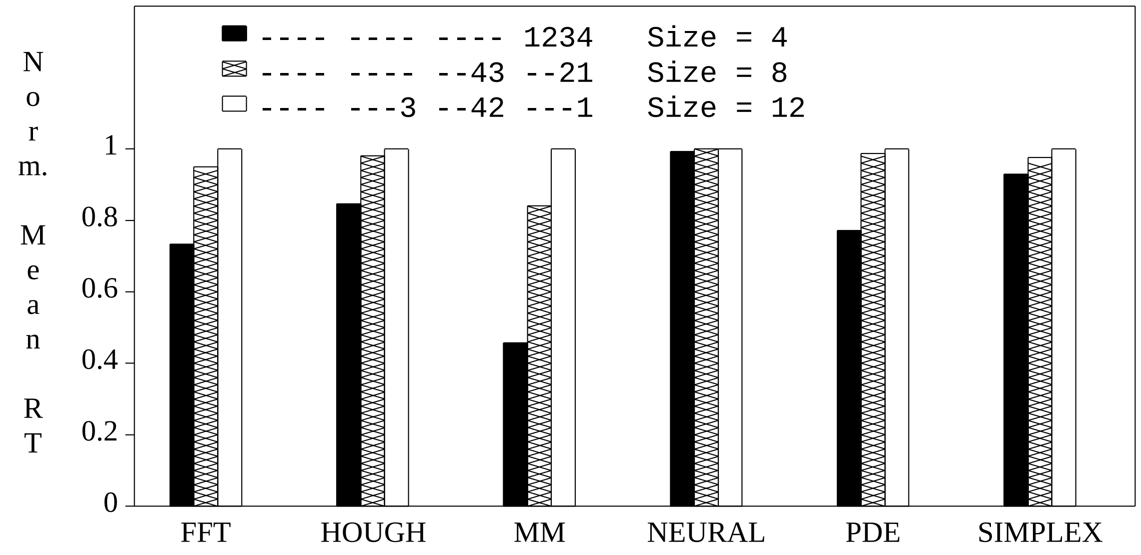


Figure 1.6: The importance of localization with varying system sizes, delay = 16

placements of four processes on two stations (8 processors) and three stations (12 processors). We expect that in larger and larger systems that the difference in performance between the non-localized placements would likely increase as the size of the systems tested were increased.

7.4. Placement of Processes within the Application

All of the applications used in these experiments consist of a parent (main) process and three children. Each application operates in a master/slave fashion, with the parent process creating the children, notifying the children of the functions they are to perform along with the sub-section of the data the functions are to be performed on, and controlling the synchronization. The child and the master processes each perform the same work on different subsets of the problem. However, because the master process is created first, it may be responsible for the initialization of some data which may cause that data to be located on the same processor as the master process. This may be as innocuous as a few variables, for example, the number of processes used and the size of the problem, but if these variables are not cached and are referenced often the cumulative cost of the remote memory accesses can affect execution times. The graph in Figure 1.7 is the result of an experiment that was conducted in order to study how the execution time of each application is affected by the location of the child processes relative to the parent. This study can be thought of as examining the following question:

- Once a localized subset of processors has been chosen for an application's execution, is the execution time affected by the location of its parallel processes within that subset of processors?

Intuitively this will depend on the symmetry, communication, and remote memory access patterns of the application.

The experiment performed considers two non-localized placements, one in which none of the child processes are placed in the same station as the parent ‘‘−−−− −−−3 −−42 −−−1’’ and one in which one of the child processes is placed in the same station as the parent ‘‘−−−− −−−3 −−41 −−−2’’. We see in Figure 1.7 that the performance of each application is affected by the placement of the child processes relative to the parent since exactly the same

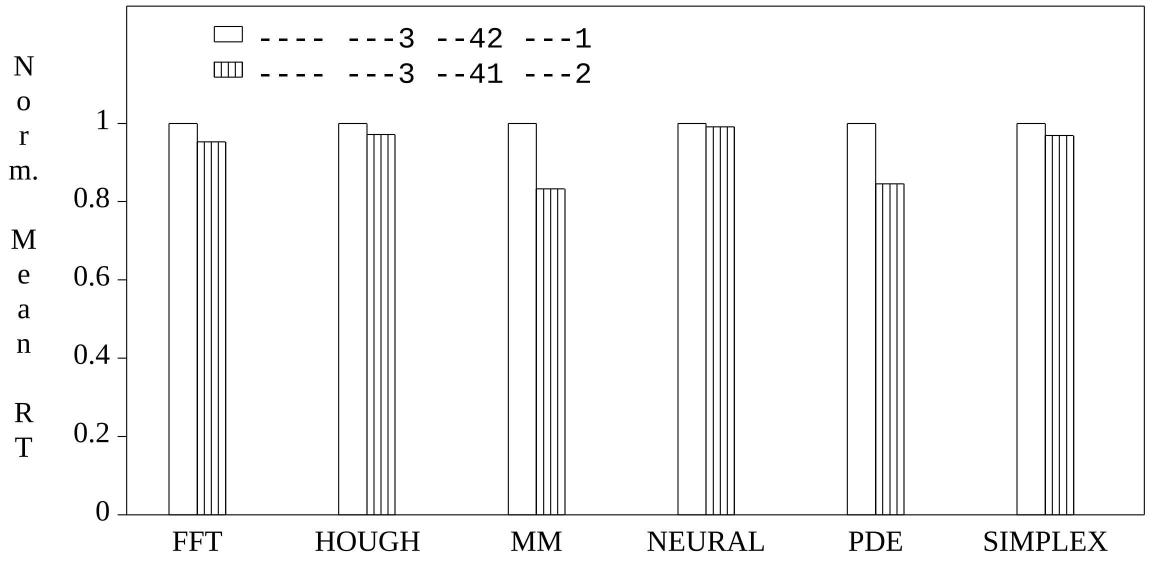


Figure 1.7: Importance of the placement of children relative to the parent, delay = 16

subset of processors is used in each case and the only difference is that the location of processes 1 (the master) and 2 (the first child) have been switched. A delay setting of 16 cycles was used and the results show that in two cases, MM and PDE the execution times differ by 15%. These results are significant enough to indicate that the location of parallel processes of an application within a subset of localized processors is important for some applications and that the child processes should be placed as close to the main process as possible.

8. Conclusions and Future Work

In this paper we have demonstrated that, in large-scale, NUMA multiprocessors, preserving the locality of parallel applications by placing processes close to each other in order to minimize the costs of accessing shared-data is essential to achieving good performance. In particular the experiments conducted in this paper have shown:

- As expected, in small-scale mildly NUMA multiprocessors, placement decisions have only a minor affect on the execution time of parallel applications.

- Application placement that considers the architectural grouping of processor and memory modules inherent in NUMA multiprocessors is essential and improves performance significantly.

- The importance of placement decisions increases with the size and NUMAness of the system and will continue to increase as the gap between processor speeds and memory access times (including interconnection schemes) continues to widen.

- Placement of the children relative to the parent (main) process affects application performance significantly. Specifically, frequently referenced data is often located on or near the processor that the parent is placed on. Thus, placing children as close as possible to the parent process reduces execution time.

Besides continuing to study the hypothesis that localization will reduce contention in a multiprogrammed environment, we are currently conducting an experimental evaluation of a technique, called processor pool-based scheduling, designed to automatically ensure that the locality of an application is preserved by the scheduler [1]. Preliminary simulation studies show that this technique does preserve locality and improve execution times of parallel applications [18].

9. Acknowledgments

I am indebted to the Hector and HURRICANE groups for the countless hours spent implementing, debugging and tuning the system hardware and software, most notably: Ron White, Michael Stumm, Ron Unrau, Orran Krieger, Ben Gamsa, and Jonathan Hanna. I wish to thank Songnian Zhou, Ken Sevcik, and the other members of the scheduling discussion group for many discussions related to scheduling in multiprocessors and Songnian Zhou for his helpful comments concerning the presentation of this paper. Thanks also to Karim Harzallah, Karen Reid, Brian Carlson, and the referees for providing valuable comments. James Pang, Deepinder Gill, Thomas Wong and Ron Unrau contributed the parallel applications. I especially wish to thank Ron White for the design and implementation of the delay switches in Hector.

References

[1] T. B. Brecht, **Processor Scheduling Techniques for Large-Scale Shared-Memory NUMA Multiprocessors**, Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, (in preparation), 1993.

[2] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie, ‘‘Overview of the KSR1 Computer System’’, Kendall Square Research, Boston, Technical Report KSR-TR-9202001, February, 1992.

[3] T. H. Dunigan, ‘‘Kendall Square Multiprocessor: Early Experiences and Performance’’, Engineering and Mathematics Division, Oak Ridge National Laboratory, ORNL/TM-12065, March, 1992.

[4] B. Gamsa, **Region-Oriented Main Memory Management in Shared-Memory NUMA Multiprocessors**, M.Sc. Thesis, University of Toronto, Toronto, Ontario, September, 1992.

[5] A. Gupta, A. Tucker, and S. Urushibara, ‘‘The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications’’, *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 120-132, San Diego, CA, May, 1991.

[6] D. Lenoski, J. Laudon, T. Joe, D. Nakahari, L. Stevens, A. Gupta, and J. Hennessy, ‘‘The DASH Prototype: Implementation and Performance’’, *The Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 92-103, May, 1992.

[7] E. P. Markatos, **Scheduling for Locality in Shared-Memory Multiprocessors**, Ph.D. Thesis, Department of Computer Science, University of Rochester, Rochester, New York, May, 1993.

[8] E. P. Markatos and T. J. LeBlanc, ‘‘Load Balancing vs. Locality Management in Shared-Memory Multiprocessors’’, *1992 International Conference on Parallel Processing*, pp. 258-267, August, 1992.

[9] E. P. Markatos and T. J. LeBlanc, ‘‘Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors’’, *Proceedings of Supercomputing ’92*, pp. 104-113, Minneapolis, MN, November, 1992.

[10] M. S. Squillante, **Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation**, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, Technical Report 90-10-04, October, 1990.

[11] M. S. Squillante and E. D. Lazowska, ‘‘Using Processor Cache Affinity Information in Shared-Memory Multiprocessor Scheduling’’, *IEEE Transactions on Parallel and Distributed Systems*, *Vol*. 4, *No*. 2, pp. 131-143, February, 1993.

[12] M. Stumm, Z. Vranesic, R. White, R. Unrau, and K. Farkas, ‘‘Experiences with the Hector Multiprocessor’’, *Proceedings of the International Parallel Processing Symposium Parallel Processing Fair*, pp. 9-16, April, 1993.

[13] D. Thiebaut and H. S. Stone, ‘‘Footprints in the Cache’’, *ACM Transactions on Computer Systems*, *Vol*. 5, *No*. 4, pp. 305-329, November, 1987.

[14] R. Unrau, **Scalable Memory Management through Hierarchical Symmetric Multiprocessing**, Ph.D. Thesis, University of Toronto, Toronto, Ontario, January, 1993.

[15] R. Unrau, M. Stumm, and O. Krieger, ‘‘Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design’’, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 285-303, Seattle, WA, April, 1992.

[16] R. Vaswani and J. Zahorjan, ‘‘The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors’’, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 26-40, Pacific Grove, CA, October 1991.

[17] Z. Vranesic, M. Stumm, D. Lewis, and R. White, ‘‘Hector: A Hierarchically Structured Shared-Memory Multiprocessor’’, *IEEE Computer*, *Vol*. 24, *No*. 1, pp. 72-79, January, 1991.

[18] S. Zhou and T. B. Brecht, ‘‘Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors’’, *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 133-142, San Diego, CA, May, 1991.