An Experimental Evaluation of Processor Pool-Based Scheduling for Shared-Memory NUMA Multiprocessors

Timothy B. Brecht

Department of Computer Science, York University
4700 Keele Street, North York, Ontario, CANADA M3J 1P3
URL: http://www.cs.yorku.ca/~brecht
email: brecht@cs.yorku.ca

Abstract. In this paper we describe the design, implementation and experimental evaluation of a technique for operating system schedulers called *processor pool-based scheduling* [51]. Our technique is designed to assign processes (or kernel threads) of parallel applications to processors in multiprogrammed, shared-memory NUMA multiprocessors. The results of the experiments conducted in this research demonstrate that: 1) Pool-based scheduling is an effective method for localizing application execution and reducing mean response times. 2) Although application parallelism should be considered, the optimal pool size is a function of the the system architecture. 3) The strategies of placing new applications in a pool with the largest potential for in-pool growth (*i.e.,* the pool containing the fewest jobs) and of isolating applications from each other are desirable properties of algorithms for operating system schedulers executing on NUMA architectures. The "Worst-Fit" policy we examine incorporates both of these properties.

1 Introduction

The number of bus-based shared-memory multiprocessors being manufactured and sold continues to increase at a rapid rate. In fact, the success of these systems has lead several major computer manufacturers to develop and offer a complete product line of shared-memory multiprocessors, from single bused systems containing a small number of processors to larger more scalable systems that contain tens or hundreds of processors.

The design of larger more scalable shared-memory multiprocessors has necessitated the need for a departure from single bus-based systems because of the inherent limits on the bandwidth of the bus. Recent efforts in designing shared-memory multiprocessors (for even small systems) have focused on more scalable architectures. Scalable multiprocessor architectures typically distribute memory modules throughout the system in order to optimize access times to some memory locations. This approach leads to a class of shared-memory architectures in which

A version of this paper appears in the **Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing**, Springer-Verlag, Lecture Notes in Computer Science, Vol. 1291, Geneva, Switzerland, pp. 139-165, April, 1997.

memory access latencies depend on the relative distance between the processor requesting the access and the location of the memory module being addressed. Such systems, called NUMA (Non-Uniform Memory Access) multiprocessors, are a departure from the more common single bus-based UMA (Uniform Memory Access) multiprocessors. Some examples of NUMA multiprocessors include the University of Toronto's Hector [48], MIT's Alewife [1], Stanford's DASH and FLASH [23][21], Kendall Square Research's KSR1 [9], SUN Microsystem's S3.mp [31], the Convex Exemplar SPP1000/1200 [13] and the Sequent STiNG [25].

The proliferation of more scalable, shared-memory multiprocessors presents new opportunities for the users and new challenges for the designers of such systems. Users are granted opportunities to solve much larger problems than previously possible, with applications that use more processors and more memory, as well as the opportunity to solve several problems concurrently, by simultaneously executing parallel applications. One of the challenges that these opportunities present to the operating system designer, which is the focus of this paper, is the implementation of scheduling algorithms designed to effectively utilize the processors while enabling the efficient execution of multiple applications. This multiprogramming of parallel applications is essential for the effective utilization of all of the processors because in larger systems not all applications will be capable of efficiently executing on all processors. [49].

A critical difference between processor scheduling in UMA and NUMA multiprocessors is that scheduling decisions in NUMA systems must also consider the time it takes to access different memory locations from different processors. Thus, NUMA scheduling policies must consider the latency incurred during remote communication (in some systems determined by the number of levels in the memory access hierarchy) and to the extent possible preserve the locality of data references inherent in parallel applications. Therefore, an important aspect of scheduling in shared-memory NUMA multiprocessors is application placement. That is, how should the parallel processes of an application be placed in a NUMA multiprocessor?

In this paper we experimentally evaluate a technique, called processor pool-based scheduling, specifically designed for scheduling kernel threads onto processors in NUMA multiprocessors. (We use the terms thread and process interchangeably and intend both to refer to schedulable kernel entities, in our case a kernel thread.) We examine two central issues related to processor pools. First, how should processor pools be formed? For example, what influences which processors should belong to which pool and how closely should pools match the architecture of the system? Second, how are processor pools used? That is, once we have formed the processor pools what algorithms should be used in assigning processes of parallel applications to pools?

The results of our experimental evaluation of this technique show that pools should be chosen to reflect the architecture of the systems (the clusters inherent in scalable shared-memory systems) and that the properties of co-locating processes of the same application and isolating separate parallel applications are keys to obtaining good performance. We found that the ''Worst-Fit'' policy we consider for assigning processes to pools incorporates both of these properties and that the benefits of using this approach can reduce mean job execution times significantly. Moreover, our results demonstrate that the benefits obtained from using processor pools increase as the gap between processor and memory speeds continues to widen.

The remainder of this paper is organized as follows: Section 2 describes the issues addressed by and the approach used in implementing processor pool-based scheduling. Section 3 describes the environment in which we experimentally evaluate our techniques and compare the performance of our algorithms. The applications and the workload used in our evaluation are described in Section 4. In Section 5 we discuss and experimentally evaluate a number of issues related to the formation of processor pools (that is deciding which processors belong to each pool). In Section 6 we outline and compare the performance of algorithms related to the use of processor pools (i.e., the assignment of threads of parallel applications to pools). In Section 7 we discuss related work and we conclude the paper with a summary of our results in Section 8.

2 Processor Pools

We believe that the requirements of an operating system scheduler for NUMA multiprocessors are essentially different from the requirements of processor schedulers for small-scale UMA multiprocessors. The requirements that we believe to be critical to the design and implementation of schedulers for multiprogrammed parallel application workloads executing on NUMA multiprocessors are:

- **Localization**: Parallel threads of the same application need to be placed close to each other in order to minimize overhead due to remote communication.
- **Isolation**: When possible, different applications should be placed in different portions of the system in order to reduce contention for shared resources such as buses, memory modules and interconnection networks.
- **Adaptability**: The system should be able to adapt to varied and changing demands. A scalable multiprocessor should support the execution of a single highly parallel application that is capable of utilizing all of the processors, as well as a number of applications each executing on a smaller number of processors.
- **Scalability**: A pervasive requirement of all software designed for scalable architectures is that the software also scales.

A *processor pool* is a software construct for organizing and managing a large number of processors by dividing them into groups called pools. Since the goal of localization is to place parallel threads of the same application in a manner in which the costs of memory references are minimized. This implies that the architectural clusters inherent in NUMA multiprocessors must be considered when forming pools. The locality of applications is preserved by choosing pools to match the clusters of the system and executing the parallel processes of an application within a single pool (and thus a cluster), unless there are performance advantages for it to span multiple pools. Isolation is enforced by allocating different applications to different pools, thus executing applications within separate sub-systems and keeping unnecessary traffic off of higher levels of the interconnection network. Note that it is possible for several applications to share one pool.

In very large systems (with 100 or more processors), processor pools can be grouped together to form ‘‘pools of pools’’. These ‘‘pools of pools’’ are chosen and managed in the same way as the original smaller pools except that they are constructed and managed in a hierarchical fashion. Hierarchical structuring techniques have been proposed and studied by other researchers [17][45][15]. In particular, Unrau, Stumm, and Krieger have used a technique called Hierarchical Symmetric Multiprocessing to structure operating systems for scalability [45][44]. They have demonstrated significant performance improvements in applications executing on the Hector NUMA multiprocessor whose operating system, Hurricane, is structured using this approach. We use Hector and Hurricane for our experimental platform. Hierarchical structuring is therefore not a focus of our study.

Although the concept of a processor pool is driven by the size and NUMA characteristics of scalable multiprocessors it is not tied to any particular architecture. In actually implementing processor pools on a specific system, the NUMA characteristics of that architecture should be identified and fully exploited. In most architectures, there are clusters of processors that are good candidates for pools. For example, in a large-scale KSR1 system [9] containing a number of RING:0 subsystems connected together with a RING:1 at the higher level, pools may be formed by grouping together the processors in each of the RING:0's. In the case of the University of Toronto's Hector system [48] as well as in Stanford's DASH [23] and FLASH systems [21], pools may be formed by grouping the processors of an individual station or cluster. In the MIT Alewife multiprocessor [1], pools might be chosen to consist of the four processors forming the smallest component of the mesh interconnect or a slightly larger set of nearest neighbour processors.

Since the concept of processor pools is proposed to help simplify the placement problem, processors are grouped together so that the main location decision to be made is which pool to place the process in. The decision of which processor within the pool to use, the *in-pool* scheduling decision, is one that can be made by another level of software that handles scheduling within the pool. In our implementation we consider processors within a pool to be indistinguishable, thus simplifying the task of in-pool scheduling. Once the scheduling server determines which processor to assign to the thread, the kernel is only responsible for creating it on that processor and for the subsequent dispatching of threads. Therefore, we focus our attention on the problem of determining which of the pools to place a process in.

Processor pool-based scheduling algorithms involve making scheduling decisions based on pools rather than individual processors. In modern multiprocessors scheduling decisions must be made at each of the following points during a job's execution (the issues related to these decision points are discussed more more detail in Section 6):

1. **Arrival**: A kernel thread is created and must be assigned to a processor (pool). The essential problem is which pool to assign the first thread of a new job to when it is not know how many (if any) children that thread will create. This decision is referred to as *initial placement*.

2. **Expansion**: A job creates a new thread for parallel execution. We call this decision point *job expansion*.
3. **Contraction**: When a thread of a parallel program completes, a processor becomes available which could be assigned to execute a new thread.

Contraction can be performed by permitting the scheduler to initiate expansion by contacting one of the executing jobs. Expansion of this type requires coordination between the scheduler and the user-level thread package's run-time system. Alternatively, the scheduler could also contact and coordinate with a job to reduce the number of threads currently executing in order to reallocate processors to another job (e.g., a newly arriving job). This this type of dynamic reallocation of processors is referred to as *dynamic repartitioning*.

In fact some parallel jobs involve multiple phases of expansion and contraction and dynamic repartitioning several algorithms have been designed in order to reallocate processors fairly and/or in order to minimize mean response time [43][50][8]. This previous work has been conducted under the assumption that memory access latencies are uniform. Unfortunately, the problem of repartitioning becomes considerably more complex on clustered, NUMA architectures. We leave this important and interesting problem as a topic for future research (see [7] for a more detailed discussion of the problem). In this paper we do not consider the dynamic repartitioning of processors because of the issues related cache affinity in the assignment of user-level threads to kernel-level threads [27][28]. This ensures that the only differences in execution times are due to job scheduling decisions rather than trying to account for possible differences in execution times due to user-level thread scheduling decisions.

3 Experimental Environment

The system used to conduct the experiments presented in this paper is a prototype shared-memory NUMA multiprocessor called Hector, developed at the University of Toronto [48]. The prototype used contains a total of 16 processors grouped into four clusters, called stations. Stations are connected with a bit-parallel slotted ring and each station consists of four processor modules connected with a bus. Each processor module contains a Motorola MC81000 processor, separate instruction and data caches and 4 Mbytes of memory for a total of 64 Mbytes of globally addressable memory. Cache coherence is enforced in software by the HURRICANE operating system's memory manager at a 4 Kbyte page level of granularity, by permitting only unshared and read-shared pages to be cacheable [45][44]. Enforcing cache coherence in software simplifies the hardware design and has permitted a simple and elegant design that has relatively mild NUMA characteristics.

Although the prototype Hector system used to conduct the experiments is configured with sixteen processors, we dedicate one station (the four processors in Station 0) to the execution of system processes, including the shell scripts used to generate the workload. This ensures that differences in execution times are due solely to differences in scheduler placements of application processes.

The *NUMAness* of a shared-memory multiprocessor 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. The degree of NUMAness of a multiprocessor is affected by: 1) The differences in memory access times between each of the levels in the memory access hierarchy. 2) The amount of memory (and number of processors) that are co-located within each level. 3) The number of levels in the NUMA hierarchy.

The Hector prototype features a set of ‘‘delay switches’’ that add additional delays to off-station memory requests. Packets destined for a memory module not located on the same station are held up at the requesting processor for the number of specified cycles. The range of possible settings are: 0, 1, 2, 4, 8, 16, 32, and 64 processor cycles. The delay switches are used to emulate and gain insight into the performance of: 1) Systems with faster processors — because processor speeds continue to increase at a faster rate than memory and interconnection networks, thus increasing remote memory access 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) Systems with more processors — since increases in the number of processors will require larger and possibly more complex interconnection networks, resulting in increased remote memory access latencies.

Table 1 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. Note that the delay switches have no affect on local or on-station requests and that with the delay switches set to 16 cycles, the off-station access times for Hector are still below those of other systems that contain faster processors, more processors or mechanisms for hardware cache-coherence [6]. Off station latencies in DASH and remote node latencies on the KSR1 are 100 or more processors cycles and latencies to memory on a remote ring in the KSR1 are about a factor of six times slower [38]. For more detailed descriptions of Hector see [48][18][41].

Table 1. Memory reference times, in processor cycles, on the 16 processor Hector system

	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

In the experiments conducted in this paper we consider systems with relatively mild NUMA characteristics by using a maximum delay setting of 16. The affect that larger delays have on application performance and on the importance of application placement is explored more fully in [6]. The results of our experiments show that even for such systems the proper placement of the parallel threads of an application can significantly affect the execution time of the application.

The HURRICANE operating system also supports page migration and replication, but these features were disabled while conducting our experiments in order to ensure that difference in mean response times (and parallel application execution times) are due only to difference in scheduling algorithms. For the purposes of this study we simplify the larger problem of scheduling in a NUMA environment by not considering thread migration. Our scheduler implements a space-partitioning [43][50] of processors with strong affinity of processes to processors [42][39][19]. In fact the operating system kernel contains separate ready queues for each processor. The migration of processes is discouraged because of the cache and memory contexts that can be associated with each process (recall that besides data and instruction caches, each processor module also contains local memory).

We also assume that the parallelism of the application and the number of processors it is allocated are not known *a priori*. In our implementation, when an application wishes to create a thread and thus gain access to another processor, the library call to create a thread first contacts the scheduling server, which executes outside of the kernel's address space. The scheduling server determines if the calling application should be allocated another processor and if so, which processor it should be allocated. If the scheduler decides to allocate another processor to the application, the system call is then passed on to the kernel, which creates the thread on the processor specified by the scheduler. When a thread finishes executing or is killed by an exception, the scheduler is notified and updates its internal state. The kernel is only responsible for dispatching threads. The scheduler, therefore, sees requests for processors one at a time and assigns them to applications until all processors have been allocated, at which point requests to create additional processes fail. All of our applications and workloads have been written to execute in this fashion. We believe this to be a more realistic approach to scheduling than assuming that the number of processors required will be known when the application starts. Note that if the number of processors required by each application is fixed and know at the time of the job arrival, the problem of determining which processors to allocate is similar to a bin packing problem with multiple bins.

The use of the Hector multiprocessor and the HURRICANE operating system provides us with the opportunity to examine the affects that increased memory access latencies are likely to have on our results. Since we have access to the complete source code for the HURRICANE operating system, scheduling server, run-time system and applications, we have the ability to modify the system to support our experiments. Because this system uses processors with relatively slow clock rates and contains only 16 processors, remote memory accesses latencies are quite low relative to systems with newer processors with higher clock rates and systems with larger and more complex interconnection networks (that support more processors and cache coherence). Therefore, we believe that our results and conclusions are somewhat conservative and that the importance of application placement and the decreases in execution times observed from using processor pool-based scheduling will only increase as the gap between processor speeds and memory speeds continues to increase and as the size of scalable NUMA multiprocessors continues to grow.

4 Parallel Applications

The applications comprising the workloads used in our experiments are listed in Table 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). For the single processor execution time, we use the time required to execute the parallel application on one processor because we did not have access to a serial version of each application. More detailed descriptions of each application and how their execution can be affected by poor processor placement decisions can be found in [6][7].

Table 2. Summary of the applications used

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

Although the size of the data sets may appear to be relatively 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 workload 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 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 may have to be allocated to a remote processor, resulting in remote memory references where the application programmer had not intended.

Some of the applications may appear not to execute very efficiently on four processors. This is due to the relatively small data sets used. Most of the applications were designed to be used with larger data sets on more processors (i.e., the parallelism is relatively coarse-grained). However, we believe that these applications represent a reasonable mix of efficiencies and should provide an adequate workload for the purposes of our experiments.

5 Forming Processor Pools

The question of how to choose the groupings of processors that form processor pools (i.e., how many processors should belong to each pool) is one that is potentially influenced by two main factors, the parallelism of the applications and the architecture of the system.

Issues related to the specific policies for assigning processors to pools are considered in detail Section 6. For now we assign newly arriving jobs to the pool with the largest number of available processors. Other processes of the job are placed within the same pool if possible. If there are no available processors in that pool then the pool with the largest number of available processors is chosen. This algorithm was devised using observations made while conducting experiments for a previous study [6] and is designed to isolate the execution of different jobs and to allow them "room to grow". This strategy corresponds to the "Worst-Fit" algorithm that is described in Section 6.

We now conduct a series of experiments designed to further explore the influences on the choice of processor pools. Although we do not exclude the possibility of choosing processor pools of different sizes, this work only considers pools of equal sizes. The goal in this paper is to gain insight into the forming of pools, the design of policies for their use, and the benefits of processor pool-based scheduling.

5.1 Determining Processor Pool Sizes

The first experiment is designed to determine if processor pool-based scheduling improves performance and, if it does, to examine appropriate pool sizes. This is done by varying the pool size while executing the same workload. Using 12 processors we compare the mean execution times of applications when executing under a scheduler that uses: 1 pool of 12, 2 pools of 6, 3 pools of 4, and 6 pools of 2 processors. We also consider 12 pools each containing one processor. Note that one pool of size 12 is comparable to not using processor pools and is equivalent to using a central ready queue from which idle processors grab processes. Because no grouping of pools is done to form "pools of pools", 12 pools of one processor is also equivalent to not using pools, with the exception of the overheads required to manage 12 pools. These overheads, although not significant, are present. Recall that although applications are permitted to span more than one pool and multiple jobs may execute within a single pool, our implementation of pool-based scheduling avoids these situations whenever possible.

When possible, pools are chosen to correspond to the hardware stations in Hector. Therefore, when pools of size two are used, each of the three stations used contains two pools, and when pools of size four are used, they exactly correspond to hardware stations. When two pools of size six are used, Pool 1 contains four processors from Station 1 and two from Station 2, while Pool 2 contains four processors from Station 3 and two from Station 2.

5.2 Workload and Results

The workload used for this experiment is comprised of five of the parallel application kernels FFT, HOUGH, MM, NEURAL, and PDE. The SIMPLEX application is not used in all of the experiments conducted in this paper because in order to obtain a reasonably efficient execution, the data set had to be large enough that it significantly increased the execution time of the entire workload, making multiple executions in order to obtain confidence intervals difficult.

The workload consists of a number of "streams" of parallel jobs. A stream is formed by repeatedly executing the five applications, one after another. Since each stream is implemented using a shell script, there are unpredictable but small delays between the completion of one application and the start of the next. The delays are small enough that they do not significantly affect the results. Each stream contains a different ordering of the five applications and all streams are started at the same time (recall that one station is dedicated to the execution of system and sequential processes, including the workload generator). The number of streams is adjusted to determine the multiprogramming level. The number of streams used is determined by dividing the number of processors used for the execution of parallel applications (12) by the parallelism of the applications (in this experiment 4). In the first experiment, each stream consists of 15 repetitions of the five applications for a total of 75 jobs per stream. The applications are each allocated four processors, so three streams are used and the entire workload consists of 225 jobs.

Because the system is relatively small, has very mild NUMA characteristics and because we are interested in how increases in NUMAness affect the results, we also run each set of experiments with delay settings of 0, 8, and 16 cycles. The results of these experiments can be seen in Figure 1. Each line in a graph plots the mean response time versus the pool size. Graphs are shown for each of the different applications with the graph labelled "COMBINED" representing the mean response times over all jobs. The vertical bars represent 90 percent confidence intervals.

We first note that, as expected in a small system with only mild NUMA characteristics, represented by the lines in the graphs labelled Delay=0, the mean response times are not significantly improved by using processor pools. However, as the NUMAness of the system increases, the performance improvements due to pool-based scheduling increase and are substantial when using a delay of 16 cycles. Also note that these improvements increase with the NUMAness of the system. The improvements can be seen by comparing the execution times of the applications using a pool of size of 12 (the no pool case), with those using other pool sizes. The closer the pool size is to 4 the better the performance. The exception is the NEURAL application which, as described in detail in previous work [6], suffers from an excessive number of system calls which overshadow the locality in the application.

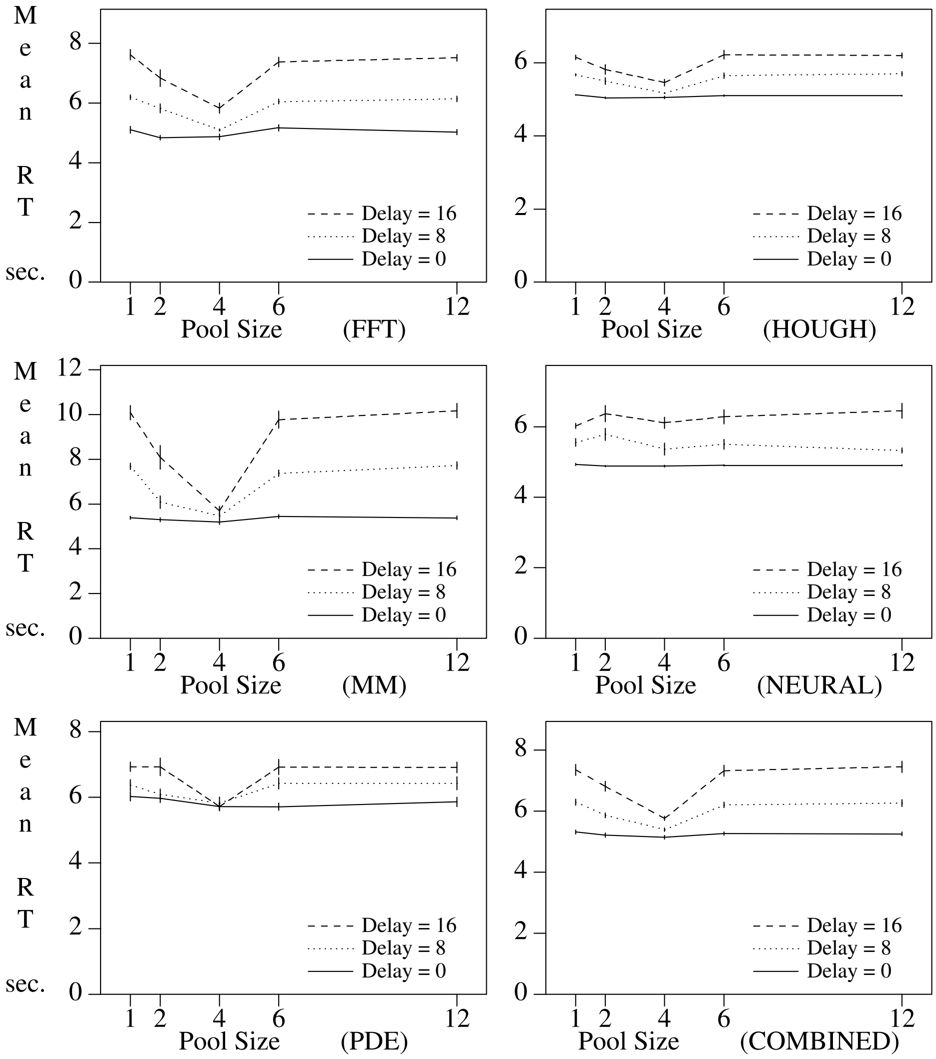

Fig. 1. Effects of NUMAness when using pool-based scheduling

Although two pools of six processors may not seem appropriate for the current workload, it is included for completeness, since it will play a central role in a future experiment. It also permits us to determine if a small enforcement of localization improves performance. The results show that even though there is a trend toward improved performance when using two pools compared with using no pools, those improvements are not large enough to be considered significant. The degree to which performance is improved varies from application to application and depends on the number and frequency of remote memory references. However, the mean response time over all jobs is improved by using pools, as shown in the graph labelled "COMBINED" in Figure 1.

The graphs in Figure 1 also show that for this set of experiments a pool size of four yields the best performance. However: 1) The parallelism of each application is four. 2) Each station in the Hector system contains four processors. Consequently, we next explore the relative influence of these two factors, application parallelism and system architecture, on the choice of pool sizes.

5.3 Application Influences on Pool Size

In order to examine the importance of application parallelism in determining an appropriate pool size, we now vary the parallelism of the applications and perform the same experiments conducted in the previous section. A delay of 16 cycles is used and the number of streams is adjusted with the parallelism of the applications (when possible keeping all processors busy). Figure 2 shows the results obtained when executing each application with two, four, and eight processes. In the case when the application parallelism is two, six streams are used, each consisting of 10 repetitions, for a total of 300 jobs. When the application parallelism is four, three streams are used, each consisting of 15 repetitions, for a total of 225 jobs. In the case when the application parallelism is eight, one stream with 25 repetitions is used for a total of 125 jobs. In this case applications are not multiprogrammed because we can not space-share two applications each using eight processors on 12 processors. The three lines plotted in each graph represent the mean response times of the applications obtained with application parallelism of two, four, and eight, versus different pool sizes. The vertical bars at each data point represent 90 percent confidence intervals.

We first observe that when eight processes are used for each application, performance is not significantly affected by the pool size. This is because the placement of eight processes within a 12-processor system does not afford as much room for localization as applications which use a smaller number of processors. Next, we observe that when applications are allocated two processors, pools of size two and four yield the best performance, again with the exception of the NEURAL application (due to excessive system calls). When the applications each require two processors, there is no significant difference in performance between using pools of size two or four because in either case each application is able to execute within one hardware station. Finally, we observe that when the application parallelism is four, the mean response time is minimized when pools of size four are used. These results might suggest that the appropriate choice of pool size might be related to the parallelism of the jobs (this is explored in the next sub-section).

An interesting outcome of the experiments shown in Figure 2 is that for some applications, most notably MM, increasing the number of processors the application uses does not necessarily improve response time. This can be seen in the case of MM by observing that the mean response time obtained using eight processors is equal to or higher than the mean response time obtained using four processors, no matter what pool size is used. These graphs demonstrate that an application's execution time can be dramatically affected by the NUMA environment and that in some cases a localized execution using fewer processors will outperform a necessarily less localized execution using more processors. Thus, there is a relationship between the allocation problem (how many processors to allocate to each application) and the

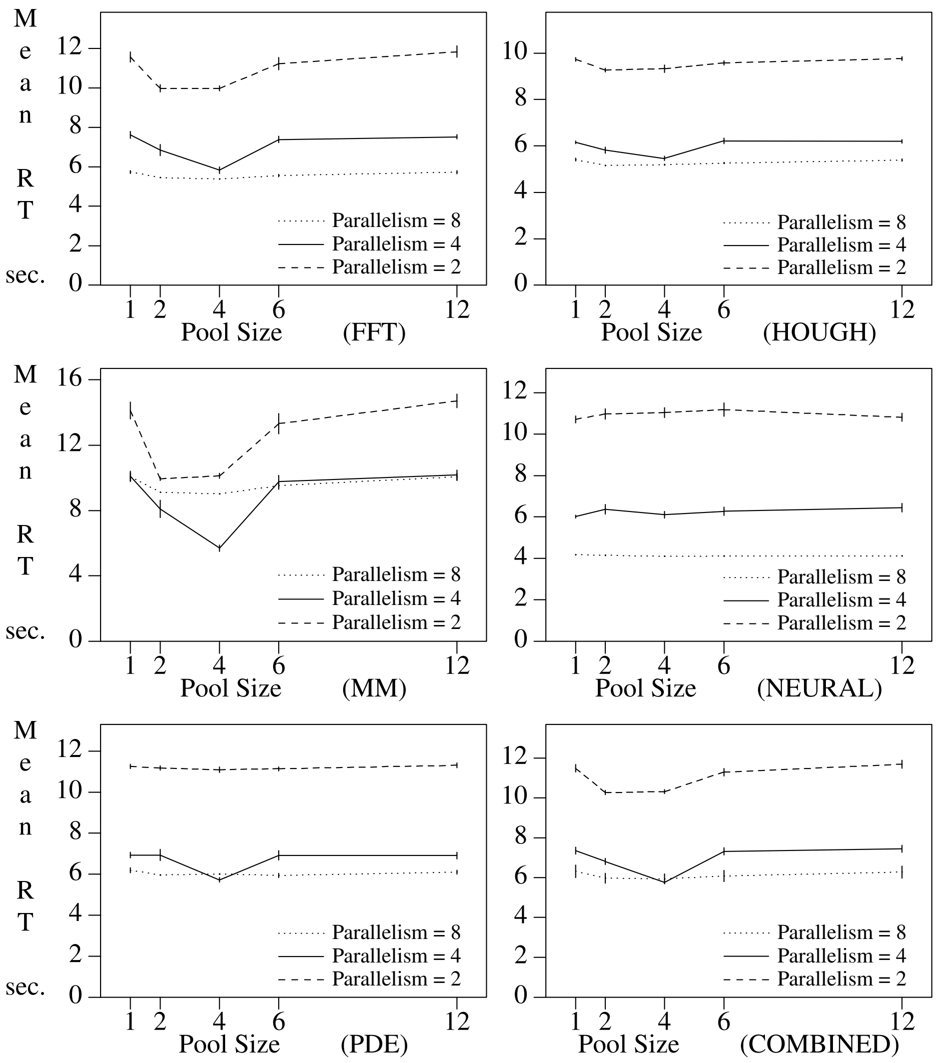

Fig. 2. Various pool sizes with application parallelism of 2, 4 and 8, delay = 16

placement problem (which processors to allocate to each application), since the number of processors to allocate to a job may depend on which processors are available. In this paper we concentrate on obtaining a first-order understanding of the issues involved in making placement decisions and in the performance benefits that can result from making good placement decisions. The relationship between these problems is discussed in more detail in [7] and is an interesting topic of future research.

5.4 Architectural Influences on Pool Size

While the experiments shown in Figure 2 suggest that there is a relationship between pool size and application parallelism, these experiments do not fully explore the relationship between pool size and the system architecture. To determine the strength of the connection between pool size and system architecture, we conduct another experiment in which each application executes using six processors. In this experiment the HOUGH, MM and SIMPLEX applications were used. The other applications (FFT, NEURAL, and PDE) are not used because, unfortunately, they are written in such a way that executing them on six processors is not possible. In these experiments, we use two streams, each of which executes the three applications 15 times, for 45 jobs per stream and a total of 90 jobs.

The graphs in Figure 3 plot the mean response times for each of the applications versus different pool sizes. The mean response times over all of the applications is shown in the graph labelled "COMBINED". The number above each of the bars gives the percentage improvement when compared with one pool of size 12. A negative value indicates that the mean response time was increased.

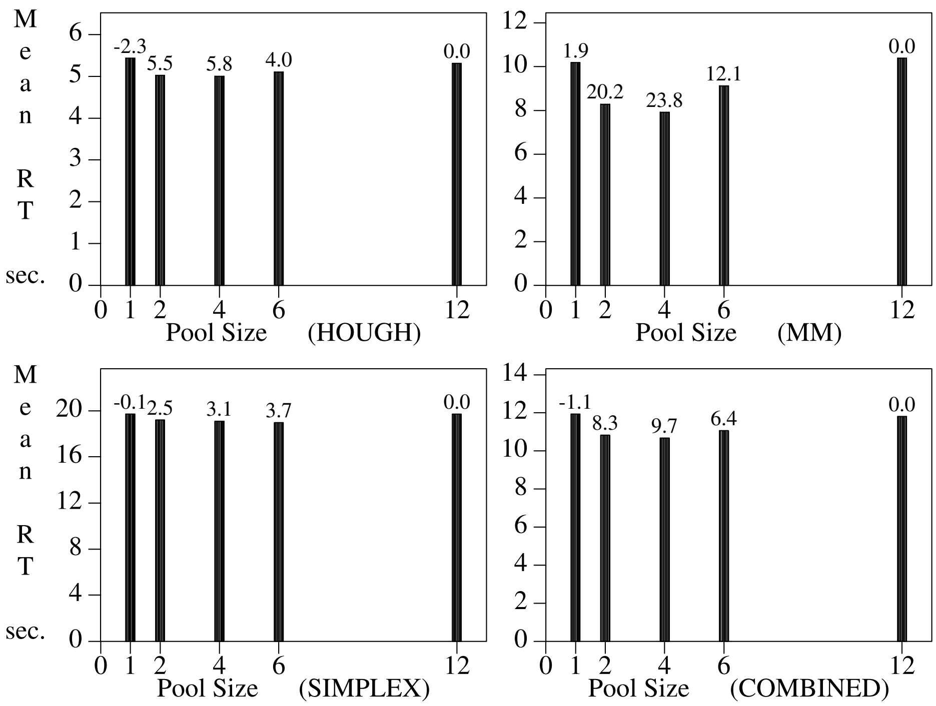

Fig. 3. Various pool sizes with application parallelism of 6, delay = 16

The main data points of interest in these experiments are the pools of size four, because this matches the size of a Hector station, and pools of size six, because this matches the application parallelism. For the HOUGH and SIMPLEX applications, although we observe slight differences in mean response times when pools of size four and six are used, the differences are not statistically significant. A pronounced difference is observed for the MM application. This is somewhat surprising since exactly the same set of processors is assigned to each application in each case. The differences in mean response times are, however, due to the placement of processes within the pools.

First, we briefly review the pool placement policy in order to understand why the placements are different. Then we explain why the resulting execution times are different. The first process of each job is placed in the pool with the largest number of available processors. Subsequent processes of that job are placed in the same pool as the first process until all of the processors in the pool are used. If more processors are required, the next pool with the most available processors is chosen. One of the goals in the design of processor pools is to form groups of processors that can be managed easily and uniformly within the pool. Therefore, we place processes randomly within pools and point out that if placement within processor pools affects performance significantly, the pools have not been chosen to appropriately reflect the architecture.

Figure 4 illustrates a scenario in which the response times of the same applications would differ using pools of size four and six. Figure 4a shows an example placement of two applications, A and B, when using pools of size four. In this case the first four processes of application A (A_1, A_2, A_3 and A_4) are placed randomly on Station 1 and the remaining processes (A_5 and A_6) are placed on Station 2. The first four processes of application B (B_1, B_2, B_3 and B_4) are placed randomly in Pool 3 on Station 3, since that is the pool with the largest number of available processors, and the remaining two processes (B_5 and B_6) are placed in Pool 2 on Station 2.

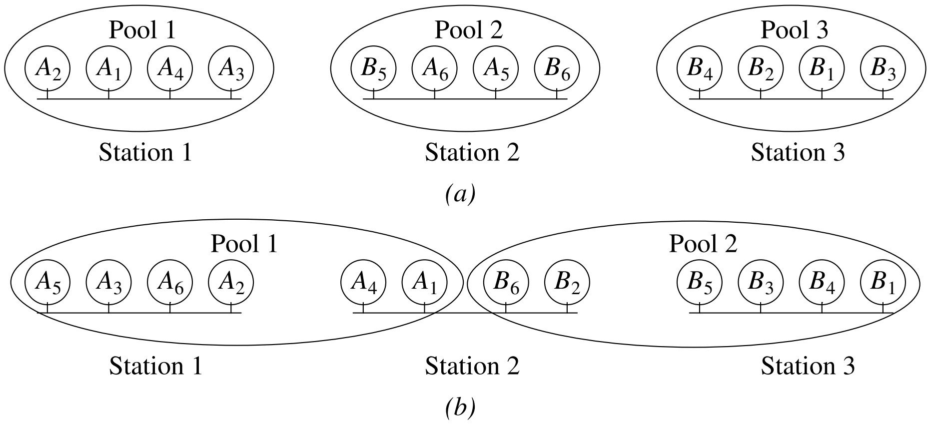

Fig. 4. Individual process placement using processor pools of size 4 and 6

Figure 4b shows an example placement when pools of size six are used. In this case each application fits entirely within a single pool. Application A is placed and executed in Pool 1 and application B is placed and executed in Pool 2. In previous work [6] we observed that if the first process of an application (the parent) is located on a station that is different from the rest of the processes of the application, the response time can be affected significantly because of the substantial memory context often associated with the first process of an application. Note that in the case when pools of size four are used, as many children as possible will be placed in the same pool as the parent. However, under closer inspection we determined in that the same is not always true when the pool size is six.

The results of this experiment, the details of which can be found in [7], show that there is a difference between using three pools of size four and two pools of size six when allocating six processors to each application. Three pools of size four yield better performance, indicating that in this case it is more important to choose pool sizes to reflect the architecture of the system than the parallelism of the applications. Also, matching pools to the architecture is likely to be relatively straightforward while, in general, a workload will consist of a number of applications with different (and possibly changing) degrees of parallelism, making it difficult to match pool sizes with application parallelism.

6 Using Processor Pools

One motivation for processor pool-based scheduling is to ease placement decisions by reducing the number and types of considerations required to make good placement decisions. This is accomplished by making placement decisions that consider pools rather than individual processors when scheduling parallel applications. An important aspect of pool-based scheduling is the strategy used for making placement decisions. We first outline the types of placement decisions that are made during the lifetime of an application and briefly point out how these decisions may influence placement strategies before examining actual strategies.

- **Initial Placement**: Before an application begins execution it must be assigned to a processor. The decision of where to place the first process of an application is an important one that can influence not only the placement of the remaining processes of the application but also the placement of other applications.
- **Expansion**: Once a parallel application begins execution, it will, at some point, create and execute a number of processes. We call this creation of new processes *expansion*. How to properly place these processes is a key consideration in preserving the locality of an application. As a result, it is essential to consider where the existing processes of the application are located. As noted in the previous section, the first (parent) process of an application may contain significant cache and memory context thus making it desirable to place as many of the child processes of that application as close as possible to the parent.
- **Repartitioning with Pools**: A change in the number of processors allocated to each application may require a dynamic repartitioning of the processors [43][24][50][29]. An important and difficult problem is how to repartition the processors while maintaining the locality of the executing applications.

The topic of repartitioning with pools is discussed in more detail in [7] and is an interesting topic for further research. In this paper we examine the first two decision points more carefully, present algorithms for making these decisions and, when possible, evaluate their performance. We begin by examining the problem of application expansion.

6.1 Expansion

Processor pool-based scheduling strategies for supporting application expansion are relatively straightforward. The desirable properties of an expansion policy are:

1. Place new processes as close to existing processes of the application as possible. This is accomplished by placing new processes in pools that are already occupied by the application. In so doing, processes are placed close to the shared data being accessed.
2. If there are no available processors in the pools already occupied by the application, choose new pools so there is as much room for future expansion as possible and interference with other applications is minimized.

Property one above is quite easy to satisfy by keeping track of where the job is already executing and assigning new processes only to pools that are already occupied by that job (using the pool containing the fewest processes). Since property two has similar requirements to the problem of initial placement, this phase of expansion can use the same algorithms as those used for initial placement. All of our experiments use the same strategy for this phase of expansion as that used for initial placement.

6.2 Initial Placement

The main considerations for making an initial placement decision (for the first process of an application) are:

1. Give the new application as much room as possible for the future creation of processes. That is, provide as much room for expansion as possible.
2. Try to isolate the execution of each application to the extent possible. That is, try to reduce the possibility of interfering with the execution of other applications by placing each application in its own portion of the system.

The problem of placing applications into pools has similarities to the problem of allocating memory in non-paged systems. An especially notable similarity is the desire to avoid fragmentation, since fragmenting processes of an application across different pools will hurt localization. Because of these similarities, we briefly consider a number of possible strategies for initial placement adapted from well known placement policies for non-paged memory systems [37].

- **First-Fit**: Pools are listed in a predetermined order by simply numbering each pool. The first process of an application is then placed in the first pool with an available processor.
- **Best-Fit**: The first process of an application is placed in a pool with the smallest, non-zero number of available processors.

- **Worst-Fit**: The first process of an application is placed in a pool with the largest number of available processors.

Of these techniques the Best-Fit and the First-Fit policies do not isolate applications from each other and may not provide room for the expansion of applications within a pool. For example, if three applications arrive in an empty system, all three may be initially placed within the same pool, thus leaving little room for the localized placement of subsequently created parallel processes of each application (recall that the number of processors an application will use is not known *a priori*). However, the Worst-Fit policy would place each of these three applications into different pools, thus permitting each to execute in their own portion of the system.

A comparison of the performance of the First-Fit and Worst-Fit policies is shown in Figure 5. A workload of three streams of applications of parallelism four is used, with each of the five applications FFT, HOUGH, MM, NEURAL and PDE being repeated in different orders within each stream. Each stream consists of 15 repetitions of the five applications for a total of 75 jobs per stream and a grand total of 225 jobs. Pools of size four are chosen to correspond to the hardware stations and a delay of 16 cycles is used to emulate systems that have stronger NUMA characteristics than our small, mildly NUMA prototype. The normalized mean response times of each of the five applications and the overall mean response time (the bars labelled COMBINED) are shown. The mean response times obtained using the Worst-Fit policy are normalized with respect to the mean response times obtained using the First-Fit policy.

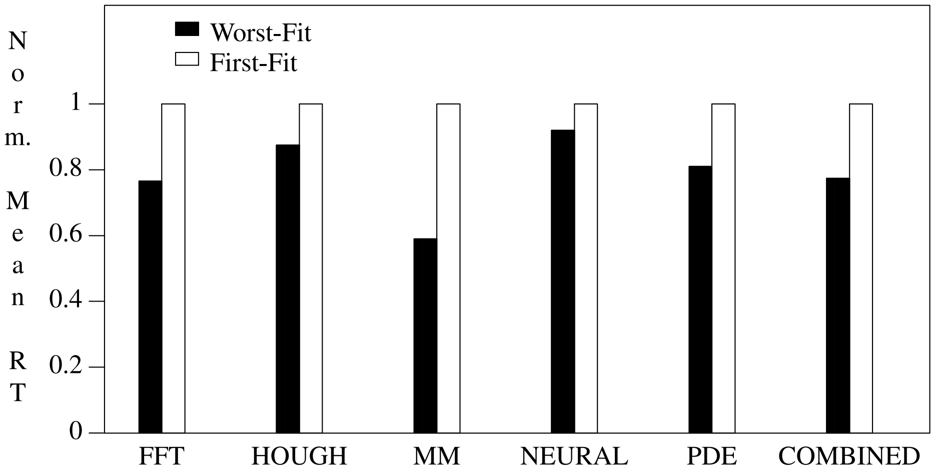

Fig. 5. Comparing Worst-Fit with First-Fit placement strategies, pool size = 4, delay = 16

As expected the Worst-Fit policy performs significantly better than the First-Fit policy and in fact it reduces the mean response times of three of the five applications by 20% or more. By examining the execution traces obtained when using the First-Fit policy (shown in Figure 6), we observe that the different applications are not always placed within one pool (and therefore one station). Figure 6 shows a number of snapshots of the allocation of processes of an application to pools and thus to stations. The numbers in parentheses to the left of each column represent, and are used to refer to, specific snapshots taken over regular time intervals. Threads within an application are labelled with the same letter of the alphabet and an unallocated processor is represented with a dash. Processors within a pool (and thus station) are grouped together by by leaving a space between pools. For example, line (1) shows that all processors are unallocated and line (16) shows that four processes of the same application (represented by “u”) are allocated to the first four processors (the first pool corresponding to Station 1), the next four processors (the second pool corresponding to Station 2) are idle and the last four processors (the third pool corresponding to Station 3) are all allocated to the same application (represented by “t”). From the trace in Figure 6, we can see a period of execution where each of the applications is executing within a separate station, in lines (12) through (17). Each application is therefore localized and isolated from the others. Lines (22) and (23) show an example of how all three applications can each have one process executing in the same station (“a”, “b” and “z” each have a processes on Station 2). These snapshots and the results of the previous experiment demonstrate that although placements using the First-Fit policy are not always bad placements, the mean response time is significantly improved by using the Worst-Fit policy.

```
     Stn1 Stn2 Stn3          Stn1 Stn2 Stn3          Stn1 Stn2 Stn3
 (1) ---- ---- ----     (10) mooo mnmm nnno     (19) xwxw yyyy wxxw
 (2) ---- ---- f---     (11) -ooo p--- ---o     (20) xwxw yyyy wxxw
 (3) ghhh ghgg ffff     (12) ---- pppp q---     (21) xwxw ---- wx--
 (4) ghhh ghgg ffff     (13) rrrr pppp qqqq     (22) abab zbab zzaz
 (5) ghhh ghgg iiii     (14) rrrr ssss q---     (23) abab zbab zzaz
 (6) k--- jjjj iiii     (15) uuuu ssss tttt     (24) abab -bab cca-
 (7) kkkk jjjj l---     (16) uuuu ---- tttt     (25) dede cede ccdc
 (8) kkk- ---- llll     (17) uuuu vvvv tttt     (26) dede cede ccdc
 (9) m--- mnmm nnno     (18) xwxw vvvv wxxw
```

Fig. 6. Sample execution trace, over time, using First-Fit initial placement policy

Other possible initial placement strategies are numerous. For example, the first process might be placed only in pools that are empty, and new applications would wait for a pool to become empty before being permitted to begin execution. Another method is a Worst-Fit policy based on the number of applications executing in a pool rather than the number of processes executing in the pool. That is, a count of the number of jobs executing within each pool is maintained and rather than assigning new jobs to the pool containing the fewest processes, they would be assigned to the pool containing the fewest jobs. This policy may be more suited to isolating applications and providing room for expansion under certain types of workloads. We believe that this last approach is likely be an improvement over our existing Worst-Fit policy. However, both algorithms behaved similarly under our current workloads.

7 Related Work

The notion of grouping processors to enhance scalability has also been proposed by other researchers [4][17][16][2][15]. Feitelson and Rudolph's distributed hierarchical technique is designed to also gang-schedule and load balance multiple applications in large multiprocessor systems [17][16][15]. Their evaluation of this technique does not take into account NUMA multiprocessors. They do point out that this technique could be used in NUMA systems. However, they do not describe how to map their tree structured distributed hierarchy onto NUMA architectures, although in a symmetric tree structured architecture the mapping is direct and should preserve locality. One advantage offered by processor pools is that they are explicitly designed to preserve the locality of parallel applications in a fashion that is not tied to a particular architecture. Furthermore, they are also designed to isolate the execution of multiple applications from one another. The combination of these two properties is intended to reduce the cost of remote references to shared data and to reduce the likelihood of contention for the interconnection network. Other work has also used the concept of clustering for different purposes. For example, Chapin *et al.* [12] use their notion of clusters (called cells) to prevent faults that occur in one cell from propagating to other cells, thus containing or localizing hardware and software faults.

Recent work has recognized that applications can build considerable cache context, or footprints [42] and that it may be more efficient to execute a process or thread on a processor that already contains relevant data in the processor's cache. Much of this work is concerned with the design and evaluation of techniques that attempt to track where processes or threads may have established cache context and to use this information to try reuse this context [39][46][19][27][28][40][26][3][35]. Our work in this paper 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 memory accesses by co-locating processes with processor caches that contain the data being accessed, our work investigates the importance of scheduling techniques for reducing the cost of required memory accesses (i.e., those references that are not cached).

Another area of work concerned with reducing remote memory access latencies concentrates on virtual memory management techniques for migrating and/or replicating pages of virtual memory. The goal of this research is to place the data being frequently referenced close to the processor or processors requesting the data [20][14][22][5][11][47]. Again, we view our work as complementary to these techniques, since it is our goal to locate the kernel threads of an application as close to each other as possible. A localized placement of processes of an application and the isolation of different applications from each other by placing them in different portions (clusters) of the system will reduce, but not eliminate, the need for migration and replication. More importantly, it will reduce the costs of migration and replication operations because of the already close proximity of processes to the data being accessed and because contention for shared resources, such as the interconnection network, will be reduced.

Several scheduling studies have recognized that that the execution time of parallel applications are affected not only by how many processors they are allocated but also by how much memory they are allocated and require. New techniques for determining how many processors to allocate are considering the memory requirements of such applications [34][30][36][33]. The techniques we present and evaluate in this paper are not concerned with the problem of how many processors to allocate but rather which processors to allocate to a job. Although we've previously discussed the relationship between the problems of allocation (how many processor to allocate) and placement (which processor to allocate) [7], the work in this paper concentrates on first gaining an understanding of the issues related to the placement problem before concerning ourselves with the interplay between the allocation and placement problems.

Chandra *et al.* [10] add cache-affinity and cluster-affinity to a UNIX scheduler by modifying the traditional priority mechanisms. Using a sixteen processor DASH system they found that while a sequential workload benefited significantly from the improved locality, this approach did not improve execution times when compared with the baseline UNIX scheduler for parallel workloads. They also compare the use of gang scheduling [32], processor sets [4], and process control [43] scheduling policies for executing parallel workloads. While they exploit cluster level locality in their implementations of each of these policies, they do not fully explore the strategies used in exploiting locality for parallel workloads nor how effective these strategies are at localization. In this paper we focus on developing guidelines and algorithms designed specifically to enforce localized placements and on evaluating the benefits of such algorithms.

In previous work Zhou and Brecht [51] present the initial concept of processor pools and conduct a simulation study which demonstrates the potential benefits obtained from using processor pools for scheduling in NUMA multiprocessors. Since then [6] we have implemented and executed a number of parallel applications on a sixteen node multiprocessor to demonstrate the significant decreases in execution times that can be obtained by considering the architecture of NUMA systems when making application placement decisions. Motivated by both of these previous studies the work in this paper undertakes an operating system level implementation and an experimental performance evaluation of processor pool-based scheduling. This work differs from the simulation study in that it focuses on the relationships between the choice of processor pool sizes and architectural clusters and pool sizes and the parallelism of the jobs being executed. While the concept of processor pools has not changed significantly from the previous paper, the algorithms, system and workload assumptions are different in several key ways:

- In the simulation model, arriving jobs request a predetermined number of threads and the scheduler permits the creation of possibly fewer threads (proportional to the number requested). In this paper the number of threads desired by a job is not known *a priori* (as is the case in most multiprogrammed multiprocessors). Also, we limit the number of threads in the system to be equal to the number of processors. This avoids unnecessary overheads due to context switching and improves processor affinity.

- The simulation model used a single ready queue per processor pool and scheduled threads within each pool in a round-robin fashion. Our implementation uses one ready queue per processor, thus providing strong processor affinity and eliminating contention for a shared pool-based queue.
- We've eliminated the ''tunable parameters'' present in the algorithms used in the simulation and concentrated on algorithmic decisions that are relatively easy to implement. For example, the degree of pool-spanning which is a control on the extent to which threads of the same job are permitted to be assigned to different pools is not present in the implementation. Instead the number of pools a job is permitted to span is tempered only by the parallelism of the job and the number of available processors.
- Obviously the system model and most components of the workload model used in this paper are more realistic than those used in the simulation.

8 Summary

In this paper we have proposed algorithms for scheduling in NUMA multiprocessors based on the concept of processor pools. A processor pool is a software construct for organizing and managing processors by dividing them into groups called pools. The main reasons for using processor pools are to preserve the locality of an application's execution and to isolate the execution of multiple applications from each other. The locality of applications is preserved by executing them within a pool when possible, but permitting them to span pools if it is beneficial to their execution. Isolation is enforced by executing multiple applications in separate pools (to the extent possible). This reduces execution times by reducing the cost of remote memory accesses. We also expect that processor pools reduce contention for the interconnection network, although we were not able to observe this on our small-scale, mildly NUMA multiprocessor. (Reducing the distance required to obtain remote memory references should reduce the use of the interconnection network.) It is expected that the scalability of the system will also be enhanced because processors within a pool can be treated equally.

We have conducted a series of experiments that explore desirable attributes of processor pool-based scheduling. In particular, we have found:

- Pool-based scheduling is an effective method for localizing application execution and reducing mean response time.
- Optimal pool size is a function of the parallelism of the applications and the system architecture. However, we believe that it is more important to choose pools to reflect the architectural clusters in the system than the parallelism of the applications, especially since the parallelism of an application may not be known and may change during execution.
- The strategies of placing new applications in a pool with the largest potential for in-pool growth (*i.e.,* the pool containing the fewest jobs) and of isolating applications from each other seem to be desirable properties of algorithms for using pools. The Worst-Fit policy incorporates both of these properties.

An observation made in [6] that is also apparent when analyzing the experiments conducted in this work is that the proper placement of processes of an application is critical and localized placements are essential for the efficient execution of parallel applications. As well, the importance of placement decisions and the improvements resulting from proper decisions increase as the size of NUMA multiprocessors increase and as the gap between processor and remote memory access speeds continues to widen.

9 Acknowledgments

This work was conducted while at the University of Toronto. I would like to thank the members of the Hector and HURRICANE projects there for their dedication and hard work in 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 also thank Songnian Zhou, Ken Sevcik, and the other members of the scheduling discussion group for many discussions related to scheduling in multiprocessors. James Pang, Deepinder Gill, Thomas Wong and Ron Unrau contributed some of the parallel applications. I am also grateful to the Natural Sciences and Engineering Research Council for the support they provided during the course of this work.

10 References

[1] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum, ‘‘The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor’’, **Scalable Shared Memory Multiprocessors**, ed. M. Dubois and S. S. Thakkar, Kluwer Academic Publishers, Norwell, Massachusetts, pp. 239-261, 1991.

[2] I. Ahmad and A. Ghafoor, ‘‘Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems’’, *IEEE Transactions on Software Engineering*, *Vol*. 17, *No*. 10, pp. 987-1004, October, 1991.

[3] F. Bellosa, ‘‘Locality-Information-Based Scheduling in Shared-Memory Multiprocessors’’, **Job Scheduling Strategies for Parallel Processing**, ed. D. G. Feitelson and L. Rudolph, *Vol*. 1162, Springer-Verlag, Lecture Notes in Computer Science, pp. 271-289, April, 1996.

[4] D. L. Black, ‘‘Scheduling Support for Concurrency and Parallelism in the Mach Operating System’’, *IEEE Computer*, pp. 35-43, May, 1990.

[5] W. Bolosky, M. Scott, R. Fitzgerald, and A. Cox, ‘‘NUMA Policies and their Relationship to Memory Architecture’’, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 212-221, April, 1991.

[6] T. Brecht, ‘‘On the Importance of Parallel Application Placement in NUMA Multiprocessors’’, *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, pp. 1-18, September, 1993.

[7] T. Brecht, ‘‘Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors’’, Ph.D. Thesis, University of Toronto, Toronto, Ontario, Technical Report CSRI-303, June, 1994.

[8] T. Brecht and K. Guha, ‘‘Using Parallel Program Characteristics in Dynamic Processor Allocation Policies’’, *Performance Evaluation*, *Vol*. 27 & 28, pp. 519-539, October, 1996.

[9] 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.

[10] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, ‘‘Scheduling and Page Migration for Multiprocessor Compute Servers’’, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, pp. 12-24, October, 1994.

[11] J. Chapin, S. Herrod, M. Rosenblum, and A. Gupta, ‘‘Memory System Performance of UNIX on CC-NUMA Multiprocessors’’, *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Ottawa, ON, May, 1995.

[12] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, ‘‘Hive: Fault Containment for Shared-Memory Multiprocessors’’, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 12-25, December, 1995.

[13] Convex, **Convex: Exemplar SPP1000/1200 Architecture**, Convex Press, 1995.

[14] A. Cox and R. Fowler, ‘‘The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum’’, *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 32-43, December, 1989.

[15] D. Feitelson and L. Rudolph, ‘‘Evaluation of Design Choices for Gang Scheduling using Distributed Hierarchical Control’’, *Journal of Parallel and Distributed Computing*, *Vol*. 35, *No*. 1, pp. 18-34, May, 1996.

[16] D. G. Feitelson and L. Rudolph, ‘‘Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control’’, *1990 International Conference on Parallel Processing*, pp. I1-I8, 1990.

[17] D. G. Feitelson and L. Rudolph, ‘‘Distributed Hierarchical Control for Parallel Processing’’, *IEEE Computer*, pp. 65-77, May, 1990.

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

[19] 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*, San Diego, CA, pp. 120-132, May, 1991.

[20] M. Holliday, ''Reference History, Page Size, and Migration Daemons in Local/Remote Architectures'', *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 104-112, April, 1989.

[21] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, ''The Stanford FLASH Multiprocessor'', *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 302-313, April, 1994.

[22] R. LaRowe Jr., C. Ellis, and L. Kaplan, ''The Robustness of NUMA Memory Management'', *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, pp. 137-151, October, 1991.

[23] 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.

[24] S. T. Leutenegger and M. K. Vernon, ''The Performance of Multiprogrammed Multiprocessor Scheduling Policies'', *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, pp. 226-236, May, 1990.

[25] T. Lovett and R. Clapp, ''STiNG: A CC-NUMA Computer System for the Commercial Marketplace'', *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 308-317, May, 1996.

[26] 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.

[27] 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.

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

[29] C. McCann, R. Vaswani, and J. Zahorjan, ''A Dynamic Processor Allocation Policy for Multiprogrammed, Shared Memory Multiprocessors'', *ACM Transactions on Computer Systems*, *Vol*. 11, *No*. 2, pp. 146-178, May, 1993.

[30] C. McCann and J. Zahorjan, ''Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers'', *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, Ottawa, ON, pp. 208-219, May, 1995.

[31] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin, ''The S3.mp Scalable Shared Memory Multiprocessor'', *Proceedings of the International Conference on Parallel Processing*, 1995.

[32] J. K. Ousterhout, ''Scheduling Techniques for Concurrent Systems'', *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 22-30, October, 1982.

[33] E. Parsons and K. Sevcik, ''Coordinated Allocation of Memory and Processors in Multiprocessors'', *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, pp. 57-67, May, 1996.

[34] V. Peris, M. Squillante, and V. Naik, ''Analysis of the Impact of Memory in Distributed Parallel Processing Systems'', *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, pp. 5-18, May, 1994.

[35] J. Philbin, J. Edler, O. Anshus, C. Douglas, and K. Li, ''Thread Scheduling for Cache Locality'', *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp. 60-71, October, 1996.

[36] S. Setia, ''The Interaction Between Memory Allocations and Adaptive Partitioning in Message-Passing Multiprocessors'', **Job Scheduling Strategies for Parallel Processing**, ed. D. G. Feitelson and L. Rudolph, *Vol*. 949, Springer-Verlag, Lecture Notes in Computer Science, pp. 146-164, April, 1995.

[37] A. Silberschatz and P. Galvin, **Operating System Concepts**, Addison-Wesley, Reading, Massachusetts, 1994.

[38] J. P. Singh, T. Joe, A. Gupta, and J. Hennessy, ''An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford Dash Multiprocessors'', *Proceedings of Supercomputing '93*, Portland, OR, pp. 214-225, November, 1993.

[39] 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.

[40] 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.

[41] 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.

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

[43] A. Tucker and A. Gupta, ''Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors'', *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 159-166, December, 1989.

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

[45] 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*, Seattle, WA, pp. 285-303, April, 1992.

[46] 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*, Pacific Grove, CA, pp. 26-40, October, 1991.

[47] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, ‘‘Operating System Support for Improving Data Locality on CC-NUMA Compute Servers’’, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, pp. 279-289, October, 1996.

[48] 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.

[49] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, ‘‘The SPLASH-2 Programs: Characterization and Methodological Considerations’’, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, 1995.

[50] J. Zahorjan and C. McCann, ‘‘Processor Scheduling in Shared Memory Multiprocessors’’, *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, pp. 214-225, May, 1990.

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