Handling Massive Parallelism Efficiently: Introducing Batches of Threads

Ioannis E. Venetis
Theodore S. Papatheodorou†
Guang R. Gao

CAPSL Technical Memo 70
October 9, 2006

†High Performance Information Systems Laboratory
Department of Computer Engineering & Informatics
University of Patras, Rion 26500, Greece

Abstract

Emerging parallel architectures provide the means to efficiently handle more fine-grained and larger numbers of parallel tasks. However, software for parallel programming still does not take advantage of these new possibilities, retaining the high cost associated with managing large numbers of threads. A significant percentage of this overhead can be attributed to operations on queues. In this paper, we present a methodology to efficiently create and enqueue large numbers of threads for execution. In combination with advances in computer architecture, this reduces cost of handling parallelism and allows applications to express their inherent parallelism in a more fine-grained manner. Our methodology is based on the notion of *Batches of Threads*, which are teams of threads that are used to insert and extract more than one objects simultaneously from queues. Thus, the cost of operations on queues is amortized among all members of a batch. We define an API, present its implementation in the NthLib threading library and demonstrate how it can be used in real applications. Our experimental evaluation clearly demonstrates that handling operations on queues improves significantly. Furthermore, we show that better load-balancing and locality of memory references, due to larger numbers of thread, can automatically improve performance of applications.

Contents

List of Figures

List of Tables

1 Introduction

Since the beginning of the computing era, the computational power required to solve the most demanding problems of each specific period of time, usually exceeded the capacity of even the fastest uniprocessor system. Parallel architectures have emerged as a solution to this shortcoming. Linking more computational units together over an interconnection network, made it possible to compute solutions to problems that were previously thought impossible to solve. A wide range of parallel architectures with different characteristics has been proposed and built. However, it became obvious very soon that the cost to handle parallelism was limiting the efficiency of those architectures. Although many of them have been designed to allow an application to express and efficiently execute all of it's inherent parallelism, the cost to handle such an amount of parallelism has proven too high to allow it.

Recent advances in computer architecture, such as *Simultaneous MultiThreading (SMT)* [14], *HyperThreading* [6], which is an implementation of SMT from Intel, and multicore processors, allow efficient execution of more fine-grained parallelism, in addition to a larger number of parallel tasks. This has been achieved by better exploiting computational units of a processor and faster implementation of mutual exclusion and context-switching in hardware. On the other hand, software has not yet been adopted to fully exploit these changes. At large, parallel programming methodologies have not been influenced by the new possibilities offered from emerging parallel architectures. Although previously developed methodologies still apply, those architectures offer new grounds for unique optimizations.

In this paper we present a novel approach that allows fast creation of large numbers of threads. Our approach is based on the notion of *Batches of Threads*, which are defined as teams of threads that are handled together, with respect to operations on queues. In conjunction with the fact that modern parallel architectures are able to handle such a number of threads, allows us to significantly reduce cost of handling parallelism and allows applications to express their parallelism at a more fine-grained level. We define an API for Batches of Threads in the context of NthLib [7], a threading library that implements the Nano-Threads programming model [11], and demonstrate how the idea can be used in real applications. Moreover, our experimental evaluation clearly demonstrates the benefits of using Batches of Threads. Not only does performance of operations on queues improve, but using larger numbers of threads in an application can automatically reduce it's execution time, due to better load-balancing and locality of memory references.

The rest of this paper is organized as follows: In Section 3 a theoretical analysis of the execution time of a parallel application is presented, in order to justify the significance of operations on queues in contemporary threading libraries. In Section 4 we define Batches of Threads and the corresponding API that has been implemented in NthLib. Moreover, the most important implementation details are discussed and an example that uses the previously defined API is presented. In Section 5 we experimentally evaluate our approach. Finally, in Section 6 we conclude our paper.

2 Related Work

Despite our extensive search, we were not able to identify any published work that proposes a way to efficiently create large numbers of threads and, hence, be directly comparable to ours. However, there have been many proposals to overcome this problem. One idea that has been heavily used and extensively studied, is to create a smaller number of threads and equally distribute the work that has to be performed among them. Representative examples from this category are *Guided Self-Scheduling* [12] and *Factoring* [5]. Other ideas have been proposed and implemented, in order to reduce the amount of time required to complete basic operations in threading libraries. An important methodology in this category is recycling of used objects, in order to avoid expensive allocations of memory. Others include lazy techniques [4, 9, 13], memory aware creation of parallel tasks and self-adapting techniques for applications [1, 2, 10].

In this paper, many of these ideas are employed and used as the basis on which we have built our own framework. The latter provides the means to efficiently create large numbers of threads, which is what differentiates our work from previous approaches.

3 The Cost of Handling Parallelism

In this section, we analyze the cost of handling parallelism in contemporary threading libraries. Our goal is to identify the operations that have the highest impact on the overhead imposed by the parallelization process. The main result of the following analysis is that two operations are the most important ones:

- Operations on ready-queues, which are used to dispatch threads for execution.
- Creation of threads. Since most threading libraries exploit recycling-queues for this operation, the importance of efficiently handling queues becomes even greater.

We consider the class of threading libraries that internally use a local queue per processor to recycle objects. This class can be further divided into two categories. The first one includes the libraries that allocate a memory region during thread creation and logically divide it into a descriptor and a stack. In this case, the objects that are recycled are the allocated memory regions. Libraries that belong to the second category use different data structures to represent descriptors and stacks, which are recycled separately. Newly created threads are inserted into ready-queues, which are also local to each processor. However, each processor can access all other queues to steal objects. Furthermore, we assume a fork/join model, where the main thread creates all other threads of a parallel region, before suspending itself and joining the rest of the processors in executing the newly created threads. Let T be the time that a serial application requires to run. We will calculate the time T_P, which is the time required to run a parallel version of the application on P processors. It is assumed that the application is parallelized such that it has only one parallel region and that N threads are created ($N \geq P$). If an application has more parallel regions, the following analysis can be applied to each one of them. The time T_P can be computed as the sum of smaller time intervals, each one corresponding to a specific operation. Those intervals can be defined by splitting the parallelization process into logically distinct phases, which are as follows:

T_C: The time required to create a thread. It includes the time to find a memory region or descriptor in a recycling queue and the time to initialize it. If an object cannot be found in a recycling queue, a new one has to be allocated.

T_E: The time required to enqueue a newly created thread into a ready-queue, so that it can be dispatched and executed by a processor.

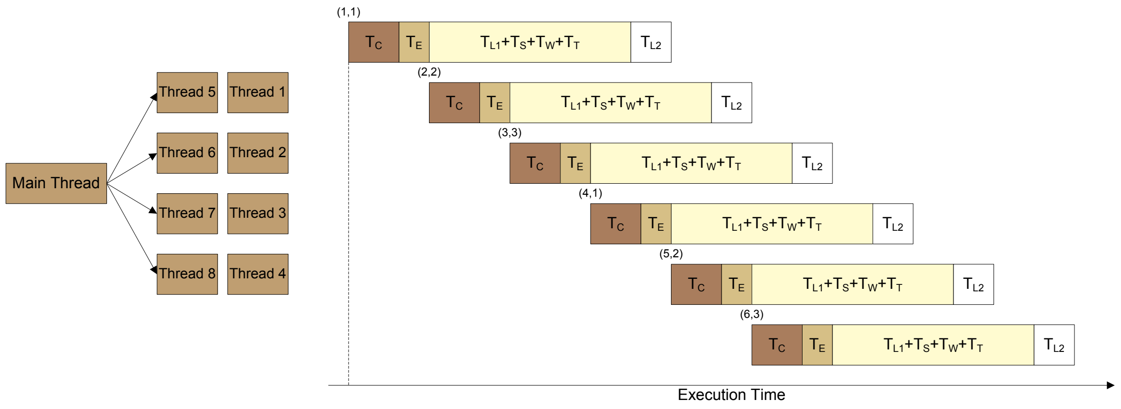

Figure 1: Creating fine-grained parallelism. The numbers in parentheses correspond to the thread and the processor that dispatches and executes it. T_C and T_E are the defining factors in this case.

T_L: The time required by the user-level scheduler of the library to find a thread in a ready-queue and dispatch it.

T_S: The time required to start a thread on a processor, after it has been selected for execution by the user-level scheduler.

T_W: The time spent by the thread for the main computation. For the rest of this paper, we will assume that the total work that has to be performed by the application is evenly divided among all threads, hence $T_W = T/N$.

T_T: The time required until the user-level scheduler is invoked, after a thread has finished it's main computation. We will assume that this step requires a predefined number of instructions to complete, hence T_T can be considered to be a constant.

In order to simplify our analysis, we divide it into two cases. In the first case, we assume that the total work that the application has to perform has been divided among so many threads, that each one requires a very small amount of time to complete. As a result, the main thread is not able to create threads at a sufficient pace, leaving at times processors without useful work. Figure 1 illustrates an example of an application that is executing on a four processor system. The main thread is running on one of them, creating threads and enqueueing them into four ready-queues. The other three processors, which are depicted, dispatch threads and execute them. It must be clarified, that each processor executes useful work only for the time $T_{L1} + T_S + T_W + T_T$, whereas it waits for new threads during all other time intervals. However, the length of some of those intervals is known a priori and equals $T_C + T_E$, which is why we directly use this value.

Taking the first processor as an example, we observe that it waits for the first thread to be created, before dispatching and executing it. During execution of the thread, the main thread continues to create new threads and delivers them to the other two processors. However, when the first processor finishes, the main thread has not yet finished creation of the next thread that this processor will execute. As a result, this processor has to wait for T_{L2}. With T_{L1}, we denote the time that a processor requires to dispatch a thread from a ready-queue. This time is not constant and depends on the number of the available queues, which in turn is equal to the number of processors. For example, the second processor dispatches the first thread that it executes from it's own queue, whereas the second thread is dispatched from the queue of the first processor (Thread 5). Since searching all queues for a thread is performed linearly, T_{L1} depends linearly on the total number of processors. If the time required to access a specific queue is T_Q and a processor has the same probability to find a thread in any of the ready-queues, then:

$$T_{L1} = \frac{T_Q + 2 \cdot T_Q + 3 \cdot T_Q + \ldots + P \cdot T_Q}{P} = \frac{(1 + 2 + \ldots + P) \cdot T_Q}{P} = \frac{\frac{P \cdot (P+1)}{2} \cdot T_Q}{P} = \frac{P+1}{2} \cdot T_Q \quad (1)$$

It is obvious that in the case being analyzed, only $P - 1$ processors are used to execute threads throughout the execution of the application. Therefore, the time required for parallel execution equals the time that each one of those processors requires to execute $N/(P-1)$ threads. However, this time must be augmented by the time that the last processor requires, until it starts executing it's first thread, yielding:

$$T_P = \frac{N}{P-1} \cdot (T_C + T_E + T_{L1} + T_S + T_W + T_T + T_{L2}) + (P-1) \cdot (T_C + T_E) \quad (2)$$

Figure 1 reveals that the total time to execute a thread and wait for the next one on each processor, i.e., $T_{L1} + T_S + T_W + T_T + T_{L2}$, equals the time to create and insert into a ready-queue the next thread that this processor will execute, i.e. $(P-1) \cdot (T_C + T_E)$. As a result, Equation 2 can be rewritten as:

$$\begin{aligned} T_P &= \frac{N}{P-1} \cdot [T_C + T_E + (P-1) \cdot (T_C + T_E)] + (P-1) \cdot (T_C + T_E) = \\ &= \frac{N \cdot P}{P-1} \cdot (T_C + T_E) + (P-1) \cdot (T_C + T_E) = \left[\frac{N \cdot P}{P-1} + (P-1)\right] \cdot (T_C + T_E) \end{aligned} \quad (3)$$

The first important observation from our analysis can be drawn from Equation 3. For fine-grained parallelism, execution time of an application depends exclusively on the time required to create a thread and insert it into a ready-queue. Moreover, execution time depends equally on those entities.

In the second part of our analysis, we assume that the created parallelism is more coarse-grained. As a result, the main thread is able to create threads in time, keeping all processors busy. Figure 2 illustrates an example, where an application is running on a four processor system. The time required to execute each thread is depicted for the three processors that dispatch threads when the application starts. Moreover, the time required to create and insert a thread into a ready-queue is also depicted, in order to make the relation among all time intervals clear. We observe that a processor never waits for a thread to arrive, after it has dispatched it's first thread, due to the fact that the main thread has enough time to create new threads. Moreover, as execution advances, more threads are created and accumulated in the ready-queues.

We observe that the first processor has to wait for $T_C + T_E$, before it starts executing the first thread. Let us assume that this processor manages to execute k threads until the main thread creates all N threads. For the rest of our analysis, we will assume, without loss of generality, that the main thread creates and inserts the last thread into a ready-queue at the time the first processor finishes executing thread k. Thus, those processors will execute threads at the same pace for the rest of the application. As a result:

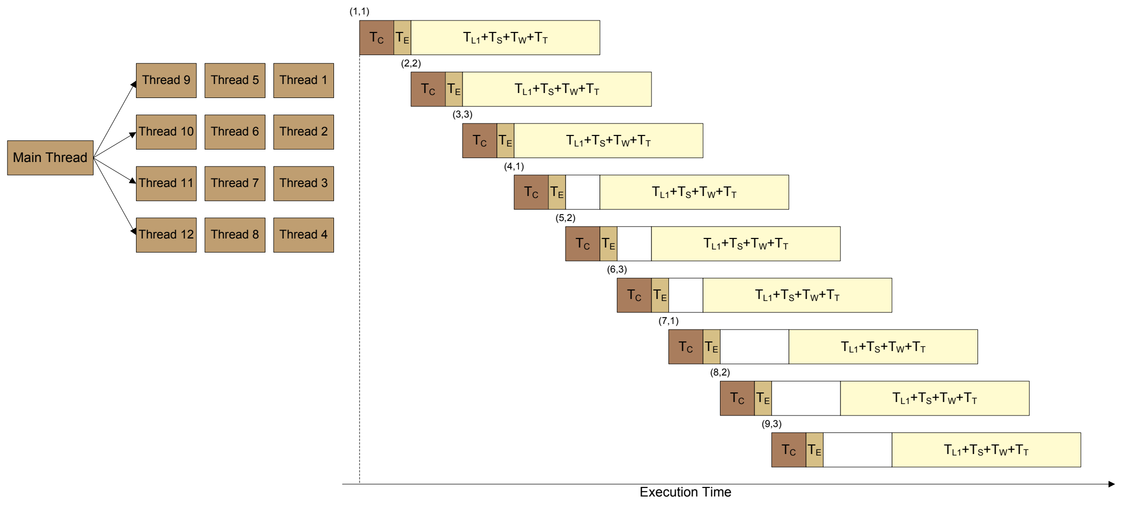

Figure 2: Creating coarse-grained parallelism. The numbers in parentheses correspond to the thread and the processor that dispatches and executes it. T_C and T_E are important in this case too.

$$N \cdot (T_C + T_E) = k \cdot (T_{L1} + T_S + T_W + T_T) + (T_C + T_E) \Rightarrow k = \frac{(N-1)\cdot(T_C+T_E)}{T_{L1}+T_S+T_W+T_T} \tag{4}$$

During this time, threads are dispatched from $P-1$ processors. If we also take into consideration the time that is required until the last processor starts executing it's first thread, a total of $(P-1)\cdot k$ threads will already have been executed, until the main thread creates all the threads of the parallel region. At this point, the main thread suspends itself and allows the processor it was running on to join the rest of the processors to execute threads. As a result, there will be P processors executing the $N-(P-1)\cdot k$ remaining threads. Therefore, total execution time will be:

$$\begin{aligned} T_P &= \frac{(P-1)\cdot k}{P-1}\cdot(T_{L1}+T_S+T_W+T_T) + (P-1)\cdot(T_C+T_E)+ \\ &+ \frac{N-(P-1)\cdot k}{P}\cdot(T_{L1}+T_S+T_W+T_T) \overset{(4)}{=} \\ &= (N-1)\cdot(T_C+T_E) + (P-1)\cdot(T_C+T_E)+ \\ &+ \frac{N\cdot(T_{L1}+T_S+T_W+T_T)-(P-1)\cdot(N-1)\cdot(T_C+T_E)}{P} = \\ &= \frac{P^2-P+N-1}{P}\cdot(T_C+T_E) + \frac{N}{P}\cdot(T_{L1}+T_S+T_W+T_T) \end{aligned} \tag{5}$$

The most important conclusion drawn from Equation 3 and Equation 5, is that the execution time of a parallel application heavily depends on the time to create a thread and enqueue it into a ready-queue. In order to formulate the time T_C, we must take into consideration that during thread creation, a memory region or a descriptor has to be found in a recycling queue. However, there is also the possibility that such an object will not be found in any of those queues and that a new one must be allocated. If we assume that the probability to find an object in a queue is always q, the probability to allocate a new object will be $(1-q\cdot P)$. If the time required to access a specific queue is again equal to T_Q, the time required to allocate a new object is T_{CM} and the time required to initialize the descriptor is T_{CI}, then:

$$\begin{aligned} T_C &= q \cdot T_Q + q \cdot 2 \cdot T_Q + q \cdot 3 \cdot T_Q + \ldots + q \cdot P \cdot T_Q + (1 - q \cdot P) \cdot (P \cdot T_Q + T_{CM}) + T_{CI} = \\ &= q \cdot \frac{P \cdot (P+1)}{2} \cdot T_Q + (1 - q \cdot P) \cdot (P \cdot T_Q + T_{CM}) + T_{CI} \end{aligned} \tag{6}$$

Since the number of possible outcomes is $P + 1$, we can set $q = r/(P + 1), r \geq 1$. Substituting q in the last Equation, gives us:

$$\begin{aligned} T_C &= \frac{P \cdot r}{2} \cdot T_Q + \left(1 - \frac{P \cdot r}{P+1}\right) \cdot (P \cdot T_Q + T_{CM}) + T_{CI} = \\ &= \frac{(2-r) \cdot P^2 + (2+r) \cdot P}{2 \cdot (P+1)} \cdot T_Q + \left(1 - \frac{P \cdot r}{P+1}\right) \cdot T_{CM} + T_{CI} \end{aligned} \tag{7}$$

The total probability to find an object in any of the P recycling queues must also be less than or equal to one. Therefore, the following inequality must also hold:

$$\frac{P \cdot r}{P+1} \leq 1 \Rightarrow r \leq 1 + \frac{1}{P} \tag{8}$$

Equation 7 reveals that the time required to create a thread depends on the time spent to search for an object in the recycling queues. Our analysis makes it clear that reducing the cost to handle parallelism requires to minimize the cost of operations on queues. Moreover, this becomes more important as parallelism becomes more fine-grained and the number of processors rises. These conclusions are not in contrast to general belief, which states that minimizing creation time for threads leads to better performance for parallel programming models. However, our analysis demonstrates the important role of operations on queues. Usually, efforts to reduce thread creation time concentrated on minimizing the number of memory allocations and on minimizing the time required to initialize a descriptor, i.e., times T_{CM} and T_{CI} in Equation 7. This was also the main argument for introducing recycling queues in threading libraries. Although this was an important step, our analysis suggests that after the introduction of recycling queues, the main problem has shifted to the time required for operations on queues.

4 Defining Batches of Threads

According to the previous section, the main question is how to reduce the cost of operations on queues. An obvious thought would be to use lock-free mechanisms to insert into and extract objects from queues. However, this is not always possible. For example, if it is required to access a queue from both the head and the tail, the data structure that represents a queue must maintain two pointers. Insertion or extraction of an object implies that both pointers must be updated together atomically. Hence, the underlying hardware must provide the necessary instructions to allow this kind of operations, which is not always the case.

The observation that leads us to a more general solution, is the fact that the associated cost for operations on queues is always measured *per thread*. This observation reveals an obvious way that allows us to reduce the

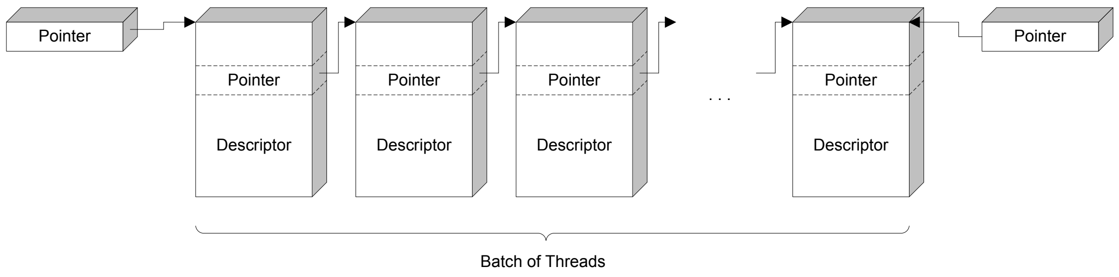

Figure 3: Representation of a Batch of Threads.

aforementioned cost. If an operation on a queue is not performed on just one thread, but on a team of threads, the cost of the operation can actually be amortized among the threads of the team. This allows us to introduce the notion of a *Batch of Threads* (BoT), which can be defined as a team of threads, that are handled as an indivisible entity with respect to operations on queues. Accordingly, the services provided by a threading library can be extended, to include creation and insertion into ready-queues of BoTs.

The above definition is very general and does not include any details about how to implement BoTs. One possibility to implement them would be to use the pointer already present in each descriptor, that is used to manage threads in queues. As can be seen in Figure 3, each member of a BoT uses this pointer to keep track of the next member, except of the last one that terminates the BoT. In order to be able to efficiently insert a BoT into a queue, it is necessary to use two pointers, that point to the first and the last member of it. Under this scheme, a BoT is actually a queue of its own, which has not yet been inserted into a predefined queue of a library, like the ready and the recycling queues. Obviously, an important parameter for a BoT is it's size, i.e., the number of threads that constitute the BoT.

Probably the easiest way to exploit BoTs is for loop-level parallelism, although their applicability is not limited to this domain. The regularity of loops in most programming languages allows easy integration of BoTs into threaded applications and allows the programmer to easily express parallelism in a natural way, as will be demonstrated in the next section.

4.1 Defining the API for Batches of Threads

In order to demonstrate that BoTs can actually be used in threading libraries, we implemented an API in the context of NthLib. More specifically, the addition of the API for BoTs has been carried out on a newer implementation of the library, which has been optimized with respect to memory requirements [15, 16]. This allows creation of a greater number of threads, compared to the original implementation of NthLib. Our main concerns while designing the API were simplicity and ease of use. In order to achieve these goals, the API has been designed to be as similar as possible to existing and widely used APIs. Moreover, the design allows both, the original and the new API for creating threads to be used simultaneously in an application, if the programmer decides that this would benefit the application.

Carefully analyzing an example that uses the current interface of NthLib to create threads, reveals important aspects of the procedure and helps us define the interface for BoTs. Figure 4 presents the simplest method to create parallelism using NthLib. The function `nth_self()` returns a pointer to the descriptor of the running

```
/****************************************************************************/

void nth_func(long Arg1, long Arg2)
{
       /* Work performed by each thread */
}

/****************************************************************************/

void nth_main()
{
        long            i;
        struct nth_desc *nth, *nth_myself = nth_self();

        nth_depadd(nth_myself, NumOfThreads + 1);
        for (i = 0; i < NumOfThreads; i++) {
                nth = nth_create_1s(nth_func, 0, nth_myself, 2, Arg1, Arg2);
                nth_to_lrq(i % kthreads, nth);
        }
        nth_block();
}

/****************************************************************************/
```

Figure 4: Creating threads with the original API of NthLib. '`kthreads`' is the number of processors.

thread, which is stored in `nth_myself`. Using the last value, a number of dependencies is added to the current thread (`nth_depadd()`), which equals the number of threads that will be created (`NumOfThreads`). The additional dependency is added for internal use of the library. Consequently, all threads are created one by one in a loop through the function `nth_create_1s()`, which returns a pointer to the descriptor of the newly created thread. Through this value, the thread is inserted into a ready-queue (`nth_to_lrq()`). Finally, the main thread suspends itself by calling the function `nth_block()`, hence calling the user-level scheduler to select a new thread to run on the processor.

A thread is interested only on the number of dependencies it has, and not on the method used to create those threads. Hence, the function `nth_depadd()` does not need any modifications in the API for BoTs, which is also true for the function `nth_block()`. As a result, new functions must be added only to create and enqueue threads. Defining the latter ones poses no special problems. If the first and the last member of a BoT are known, insertion either in front or at the end of a queue can be performed easily. However, defining the functions to create BoTs seems to be more demanding. Creation of a thread can be divided into two stages. First, a memory region or a descriptor has to be found in a recycling queue. If that fails, a new object must be allocated. Secondly, the descriptor must be initialized. The goal in using BoTs is to minimize references to queues. Therefore, finding objects in the recycling queues should also be done using BoTs. Each time a processor locks a queue, in order to create a BoT, it should return from that queue as many objects as possible, without surpassing the requested size of the BoT. Initializing each descriptor in this step, would increase the time that each processor remains in a critical section. Therefore, it would be more efficient to initialize each descriptor of a BoT, after the latter has been created. Due to this fact, creating a BoT and initializing each descriptor of it have been defined to be separate operations in the current implementation. As a result, the API that we defined is as follows:

- `struct nth_desc *nth_batch_get_desc(struct nth_desc **last_nth, long num_of_nths)`:
  Create a BoT of size `num_of_nths`. A pointer to the first member of the BoT is returned, whereas a pointer to the last member is stored in `last_nth`.

- `void nth_batch_create(struct nth_desc *nth, void (*nth_func)(), int ndep, int nsucc, int narg, ...)`:
  Initializes the descriptor `nth`, which belongs to a previously created BoT. The thread will execute the function `nth_func()` and will depend on `ndep` threads. The parameters `nsucc` and `narg` are the number of threads that depend on the thread being created and the number of arguments to `nth_func`. Finally, the dependant threads and the actual parameters of the function are mentioned.

- `void nth_batch_create_1s(struct nth_desc *nth, void (*nth_func)(), int ndep, struct nth_desc *succ, int narg, ...)`:
  This function is the same as the previous one, with one exception. It allows only one thread to depend on the thread being created (`succ`).

- `struct nth_desc *nth_batch_get_next(struct nth_desc *nth)`:
  Returns the thread that follows `nth` in the BoT or `NULL`, if there are no other threads.

- `void nth_batch_to_lrq(int which, struct nth_desc *first_nth, struct nth_desc *last_nth)`:
  Insert the BoT with first member `first_nth` and last member `last_nth` into the local ready-queue of processor `which`.

- `void nth_batch_to_lrq_end(int which, struct nth_desc *first_nth, struct nth_desc *last_nth)`:
  Insert the BoT with first member `first_nth` and last member `last_nth` into the end of the local ready-queue of processor `which`.

Having defined the new API, we will demonstrate its use by changing accordingly the example of Figure 4. The new program is depicted in Figure 5. We assume that each BoT will have a size of `BatchSize` and that the total number of threads will be again `NumOfThreads`. Firstly, we compute the total number of BoTs that will be created (`nth_batch`) and the possible remainder (`nth_batch_remainder`). In the next step, we update the dependencies of the current thread, as in the previous example. At this point, we observe that instead of the loop that creates threads, there is a loop over all BoTs. In each iteration, a BoT of size `BatchSize` is allocated, using the function `nth_batch_get_desc()`. An important difference is that in addition to the pointer that is returned (`first_nth`), another pointer to the last member of the BoT is also updated (`last_nth`). At this point, the descriptors in the BoT have not yet been initialized. Using the temporary variable `temp_nth` and a second loop, we initialize the first descriptor (`nth_batch_create_1s()`) and move to the next descriptors in the BoT (`nth_batch_get_next()`). Finally, using the pointers to the first and last member of the BoT, we insert the latter into the local ready-queue of a processor. If there are any remaining threads, they are handled in the same way, as a BoT of a smaller size. The last step, as in the original example, is to suspend the main thread.

The main difference of this approach, compared to the first example, is that thread creation is handled in two levels, instead of one. Specifically, we handle a BoT as one entity, with respect to operations on queues, and each

```
/******************************************************************************/

void nth_main()
{
        long            i, nth_batch, nth_batch_remainder;
        struct nth_desc *first_nth, *last_nth;
        struct nth_desc *temp_nth, *nth_myself = nth_self();

        nth_batch = NumOfThreads / BatchSize;
        nth_batch_remainder = NumOfThreads - nth_batch * BatchSize;

        nth_depadd(nth_myself, NumOfThreads + 1);

        for (i = 0; i < nth_batch; i++) {
                first_nth = nth_batch_get_desc(&last_nth, BatchSize);
                temp_nth = first_nth;
                while (temp_nth != NULL) {
                         nth_batch_create_1s(temp_nth, nth_func, 0, nth_myself, 2, Arg1, Arg2);
                         temp_nth = nth_batch_get_next(temp_nth);
                }
                nth_batch_to_lrq(i % kthreads, first_nth, last_nth);
        }

        first_nth = nth_batch_get_desc(&last_nth, nth_batch_remainder);
        temp_nth = first_nth;
        while (temp_nth != NULL) {
                nth_batch_create_1s(temp_nth, nth_func, 0, nth_myself, 2, Arg1, Arg2);
                temp_nth = nth_batch_get_next(temp_nth);
        }
        nth_batch_to_lrq(0, first_nth, last_nth);
        nth_block();
}

/******************************************************************************/
```

Figure 5: Creating threads with the new API of NthLib.

descriptor of a BoT separately to initialize them. The second point that needs some attention, is that the number of threads that must be created might not be exactly divisible by the size of the BoTs. In this case, the remaining threads must be handled separately.

A last remark about the newly defined API, is the fact that it can be used together with the previous approach, due to the fact that both create and enqueue threads using the same ready and recycling queues. A possible scenario, where this could be useful, would be an application that needs to create a small number of threads per processor in some parallel regions, whereas a larger number of threads in the remaining regions. In the first case, the original API could be used, whereas in the second case the new one.

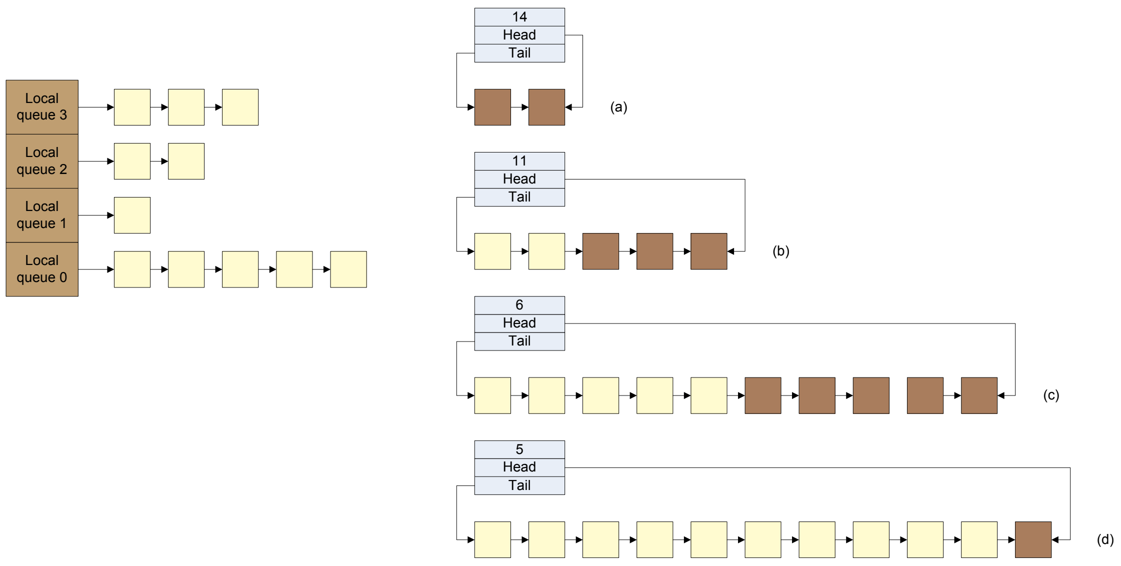

Figure 6: Finding descriptors in recycling queues to create a Batch of Threads.

4.2 Implementation Details

In order to better comprehend the concept of BoTs and their potential, we will briefly describe some implementation details. Of all the functions that were defined in the previous section, the most important are `nth_batch_get_desc()` and the functions that insert a BoT into a queue. All other functions are quite simple. The functions `nth_batch_create()` and `nth_batch_create_1s()` only initialize the fields of the descriptor. Some of them are initialized according to the parameters of those functions, whereas others get default values. Their only differences, with respect to the corresponding functions of the original API (`nth_create()` and `nth_create_1s()`), are that they receive as a parameter the descriptor that must be initialized and that the pointer to the next descriptor in a queue is not initialized, due to the fact that it is already used to manage the descriptor in a BoT. The function `nth_batch_get_next()` is also very simple, since it only returns the value of the previously mentioned pointer.

Using Figure 6 as an example, we will describe how `nth_batch_get_desc()` creates a BoT. We assume that an application is executed on four processors and that the requested size of a BoT is 16. Moreover, we assume that the descriptors in the recycling queues are as depicted. If the above function has been called from thread 2, the search for descriptors will start from local queue 2. After acquiring the lock of the queue, the function will extract as many descriptors as possible, without surpassing the requested size of the BoT. In our example, it will take two descriptors and update two pointers to the first and last member of the BoT (Case (a)). Another variable will be updated to reflect how many descriptors are still needed. The search will continue on local queue 3, where three more descriptors will be added to the BoT and all variables will be updated accordingly (Case (b)). If this queue had at least 14 descriptors, the BoT would be complete, but since this is not the case, local queue 0 is examined. Five more descriptors will be added to the BoT (Case (c)) and finally local queue 1 will be accessed. One more descriptor will be added to the BoT and all variables will be updated. At this point, all queues have been checked, however the BoT is still not complete. In this case, the remaining descriptors are allocated from memory.

An important conclusion from the description above, is the fact that creating a BoT requires a maximum number of accesses to queues, which is equal to the number of processors. If the 16 descriptors of the above example would have been allocated separately, a minimum of 16 accesses to queues would be necessary, due to the fact that a descriptor might not be available in a queue. Moreover, the number of descriptors that must be allocated in the above scenario, remains the same either with the new or the original API.

Although the functions that insert a BoT into a queue are quite important, their implementation is simple. After acquiring the lock of the specified queue, those functions insert the BoT either in the front or at the end of the queue. This procedure is simplified by the fact that a BoT is actually represented using two pointers, one to the first and one to the last member. Since all predefined queues in NthLib maintain the same information, insertion of a BoT poses no special problems.

5 Experimental Evaluation

In order to evaluate our approach, we implemented the proposed API for BoTs in the context of NthLib. The version of NthLib that has been used, is the one that implements a *Direct Stack Reuse* scheme, which allows the library to drastically reduce memory requirements to represent parallelism, without sacrificing performance. We refer the reader to [15, 16] for more details about the specific implementation.

Our experiments were run on two hardware platforms. The first one is a 4-processor, HyperThreading enabled system, running Linux 2.6.8. The second one is SMTSIM [14], a simulator that implements an Alpha processor with 8 execution contexts (EUs). More detailed characteristics for both systems are summarized in Table 1. The compiler used is gcc 4.0.2 for both platforms, at the highest optimization level (-O3).

Our evaluation is focused towards proving the main points that have been discussed so far in the paper. We believe that our experiments clearly show that:

1. The overhead of handling queues is significantly reduced using BoTs, which in turn affects positively the time required for other basic operations in threading libraries.

	Intel processor based system	SMTSIM
Processors	4 Intel Xeon MP HTs, 2 GHz, 2 execution contexts/processor	1 Alpha based, 8 execution contexts
L1 Data Cache	8KB shared, 4-way assoc.	32KB, 2-way assoc., 10-cycle miss latency
L1 Inst. Cache	12KB shared execution trace	32KB, 2-way assoc., 10-cycle miss latency
L2 Cache	512KB shared, unified, 8-way assoc.	256KB, 2-way assoc., 15-cycle miss latency
L3 Cache	1MB shared, unified, 8-way assoc.	2MB, 2-way assoc., 125-cycle miss latency
D-TLB	64 entries	128 entries
I-TLB	2x64 entries	48 entries
DRAM	2GB	Depends on host system

Table 1: Hardware configuration of the experimentation platform.

2. Real-world cases where usage of BoTs is beneficial exist. Additionally, converting those applications to use BoTs is straight-forward.

3. Using larger numbers of threads can lead to better performance in several cases, where load-balancing and locality of memory references are achieved automatically in an application.

The first benchmark we used, which we will refer to as *Empty*, follows the fork/join model. The master thread creates one million empty nano-threads, whereas the slave processors dispatch and execute them. The master thread blocks after it has created all threads, hence calling the user-level scheduler and joining the other processors to execute threads. This benchmark is appropriate for estimating the pure run-time overhead of thread management in NthLib. In the original version of the benchmark, which we will refer to as *Natural*, all threads are created one-by-one. Additionally, we implemented a version, which we will refer to as *Batch*, that creates threads using BoTs with a size of 8.

Figure 7 summarizes the results for this benchmark on both platforms. Execution times are given in seconds for the Intel based system and in millions of simulated clock cycles for SMTSIM. For the latter, the horizontal axis represents the number of EUs used. For the Intel based system, the numbers of physical processors and EUs used on each one of them are mentioned. For example, (4, 1) means that 1 EU was used on each one of the 4 physical processors. A special case is the one denoted with (4, 1/2), where 2 EUs were used on 2 physical processors and 1 EU on the other 2 physical processors. With the exception of two EUs on one physical processor, creating threads using BoTs is from 5,06% (case (1,1)) up to 43,33% (case (2,2)) faster on the Intel platform. For SMTSIM, the range is between 1,70% (1 EU) up to 69,11% (4 EUs).

In order to better understand these large differences, we include Figures 10 up to 13, where the time required for basic operations of NthLib is presented. To obtain these results, we run the same benchmarks as above and used the Time Stamp Counter on both hardware platforms, to measure such small time intervals. All results presented are per thread, meaning that the measured times for *Batch* have been divided by the size of each BoT. With respect to the Intel platform, creation time of a thread has not changed significantly, when BoTs are used. This can be attributed to the large number of descriptors that have to be allocated during execution. The time to start a thread after it has been selected to run, also did not change significantly, since the steps required to do so are almost identical in both cases. However, the time to enqueue a thread into a ready-queue has dropped significantly, from about 180 to about 12 cycles per thread. Finally, the time required to find the next thread that will be executed on a processor, also did not change significantly. The exception occurs when all physical processors and EUs are used. At this point, the contention on the queues starts to show in the *Natural* variation, whereas the usage of BoTs contributes in keeping contention low. With respect to SMTSIM, we observe that the time to create a thread is worse, if up to two EUs are used. Again, this can be explained by the fact that many descriptors have to be allocated during execution. As a result, when members of the corresponding data structures have to be accessed, they are usually not found in the cache hierarchy. If, however, the number of EUs rises, the time required drops significantly, as reuse of descriptors improves. Since SMTSIM does not measure the time required to serve a system call, the behaviour in this case is consistent with our theoretical approach, where time spent for memory allocation is considered to be low. As with the Intel platform, time to enqueue a thread again improves significantly. We also notice that the time required to find the next thread to be executed, behaves as in the case of the Intel platform, although the contention on the queues shows up much earlier in the *Natural* variation.

As our approach is especially suitable for loop-level parallelism, we chose to use in our evaluation the C version of two of the Livermore Kernels [3, 8], specifically Loop 6 and Loop 21. The Livermore Kernels are excerpts from actual production codes, used at the Lawrence Livermore National Laboratory. Hence, they can be used to evaluate the performance of our approach in real applications. Loop 6 is a general linear recurrence equation. Due to data dependencies, the original code of the loop has to be executed serially. Therefore, we parallelized a modified version of the loop, as proposed in [3]. The main characteristics of the modified loop are that it is unbalanced and that it requires fine-grained synchronization. We implemented three variations of the loop. The first one (*Equal*) follows a classical parallelization strategy. A number of threads, equal to the number of processors used, are created and the outer-loop iteration space is divided equally among them. The second variation (*Natural*) creates one thread for each point of the iteration space of the outer-loop. Threads are created one-by-one. The last variation (*Batch*) is similar to *Natural*. However, threads are created using BoTs of size 8. In order to obtain measurable execution times for the loop, we changed the problem size to $N = 7500$, which results to an execution time of about 1 second for the *Natural* variation on the Intel based system. Figure 8 summarizes the results for this loop on both platforms. For the Intel based system, the overhead of synchronization on the queues and in the application are obvious for the *Natural* variation. However, using BoTs to create the threads significantly alleviates the queueing subsystem of the library. As a result, the performance becomes comparable to the *Equal* variation. For SMTSIM, we observe the same behaviour for the *Natural* variation, as on the Intel based system. However, the *Batch* variation yields not only comparable, but better results than the *Equal* variation on this system. SMTSIM implements a very efficient locking mechanism, based on the notion of a *lockbox* [14]. As a result, it eliminates a large percentage of the overhead associated with handling queues in the library, compared to the Intel based system, but also the synchronization required in the application. This, in turn, makes the imbalance present in the application a much more important factor. Using the *Equal* variation, all threads have not the same amount of work to complete. As a result, execution time depends largely on the slowest thread. In the *Batch* version, however, the large number of threads allows the application to self-tune it's execution and automatically achieve a much better load-balance among processors. Although it is possible to implement the *Equal* variation by taking into account load-imbalance, the code is much larger and more difficult to understand. On the other hand, using BoTs has the same effect and is much simpler to program.

The second kernel that we decided to employ for our evaluation purposes is a very interesting one, as it demonstrates how data distribution among threads can greatly affect performance. Loop 21 is a $N \times 25$ by 25×25 matrix-matrix multiply. We implemented the same three variations as in the previous loop, parallelizing the inner-most and largest loop of the application. Furthermore, we changed the problem size to $N = 200001$, so as to obtain a 1 second execution time of the *Natural* variation on the Intel based system. Figure 9 summarizes the results for this loop on both platforms. Obviously, the difference between the *Equal* and the other two variations is impressive. The reason behind this large difference is the exploitation of the cache. For the *Equal* variation, the parallelized, inner-most loop has to work on more rows of the large array that is being multiplied. As a result, the cache hierarchy is not exploited in the best possible way. However, when the remaining two variations are used, one thread is actually created for each row that is multiplied. As a result, the cache is exploited almost perfectly for each thread in this case. Although it is possible to rearrange the loops of the kernel, so that the *Equal* variation requires about the same amount of time to complete, this requires careful analysis from the programmer. However, using more threads in this case automatically provides a better mapping between the data that each thread has to access and the cache hierarchy of the system. Again, this provides the programmer with a more natural way to express parallelism and obtain good performance.

6 Conclusions

In this paper, we presented a methodology to efficiently create large numbers of threads, using BoTs. We defined an API in the context of NthLib and discussed implementation details. Our evaluation shows that time to handle parallelism has significantly improved. Moreover, exploiting large numbers of threads has proven to be beneficial in several cases, either due to better load balancing or data distribution among threads. This proves that our approach is viable and justifies our effort towards this direction.

Our current work focuses on better exploiting our improvements presented in this paper. Specifically, it has become clear that applications are not taking full advantage of our newly implemented mechanisms, due to excessive memory allocations for descriptors. This is triggered by the fact that overhead for dispatching and executing a thread in NthLib are still quite high, compared to the time that is required to insert a thread into a ready-queue. This leaves the main thread without enough descriptors in the recycling queues. In order to overcome this inefficiency, we are currently trying to exploit BoTs internally in the library. This approach can work together with the one presented in this paper. Our initial experiments show that this can improve even more the time required to create and execute a thread and yields even better results for our applications.

Acknowledgments

We would like to thank Christos D. Antonopoulos and Dimitrios D. Nikolopoulos, for providing access to the systems that where used throughout the evaluation process. We would also like to thank Jeffery A. Brown, who patiently answered many questions about SMTSIM.

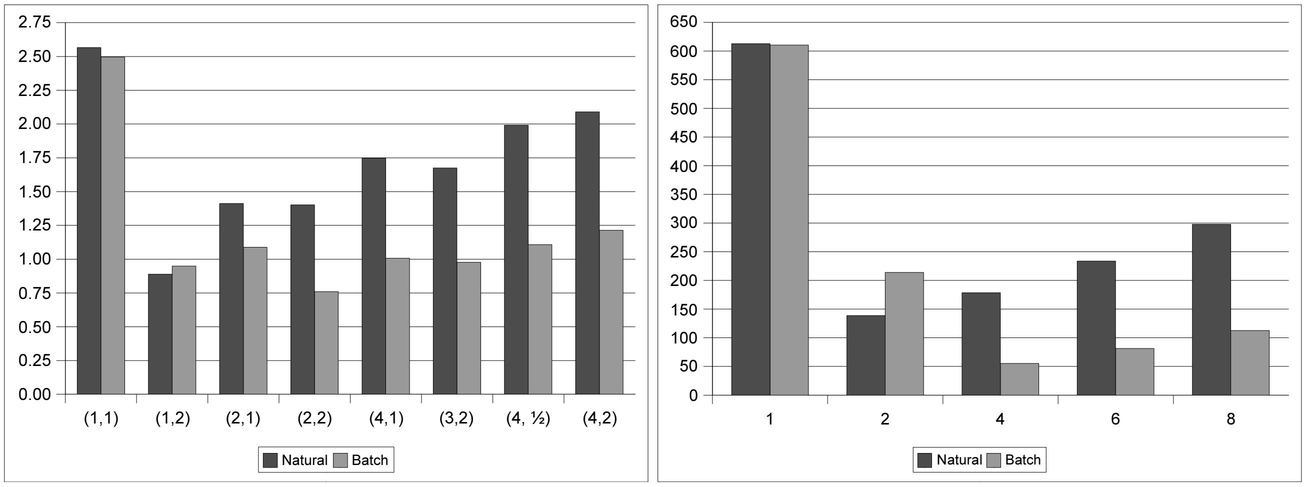

Figure 7: Execution time for the *Empty* benchmark on the Intel and SMTSIM platforms.

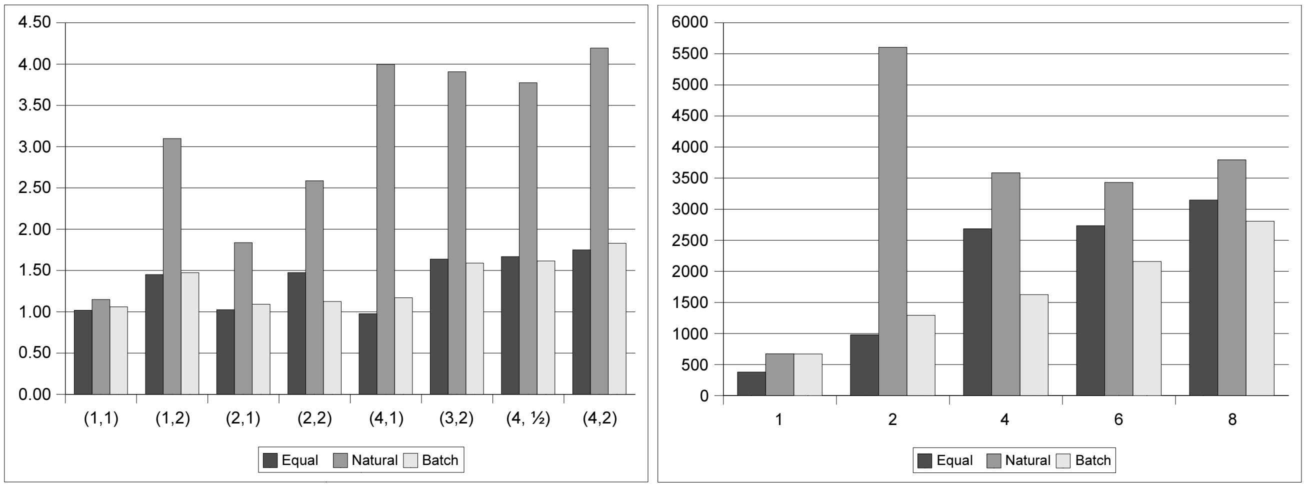

Figure 8: Execution time for the *Loop 6* benchmark on the Intel and SMTSIM platforms.

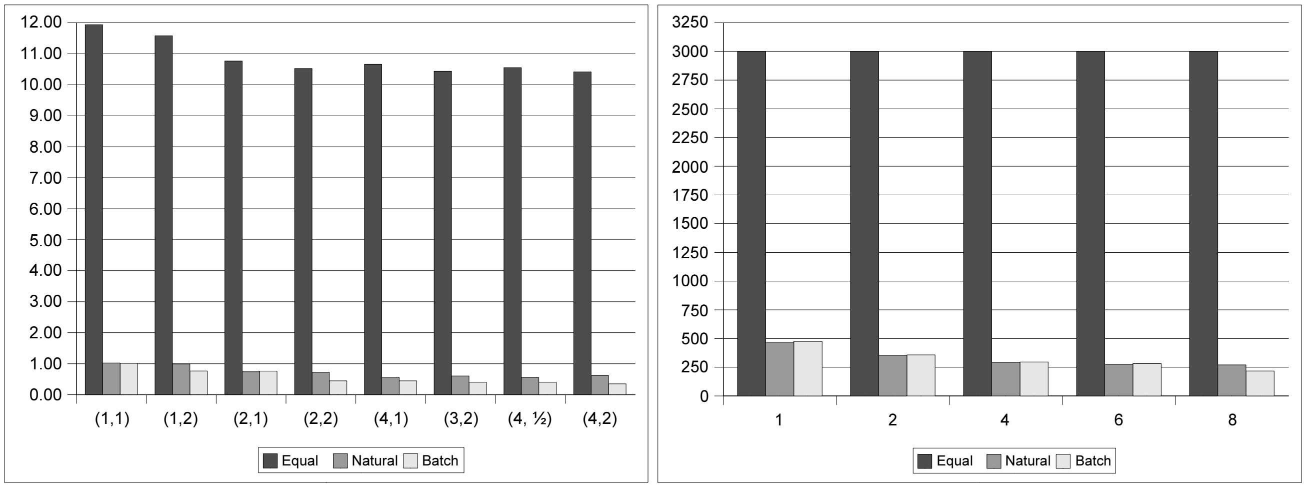

Figure 9: Execution time for the *Loop 21* benchmark on the Intel and SMTSIM platforms.

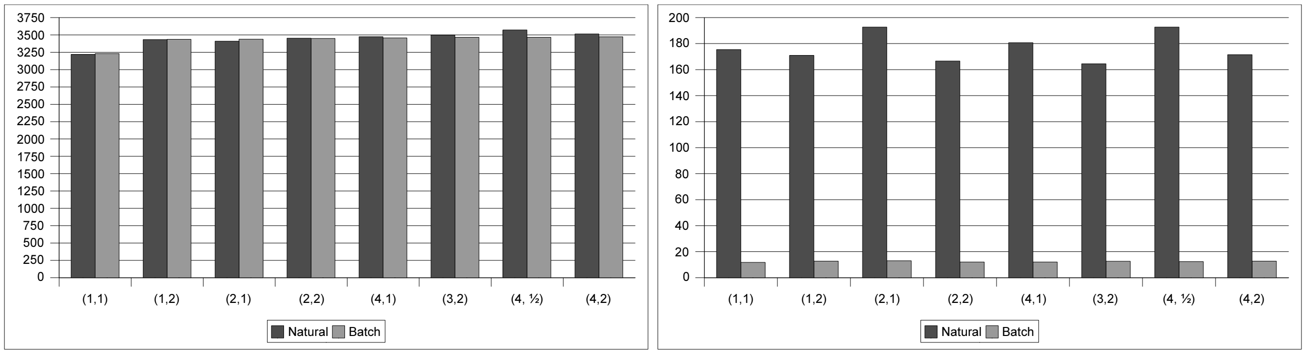

Figure 10: Time required to create and enqueue a thread on the Intel platform.

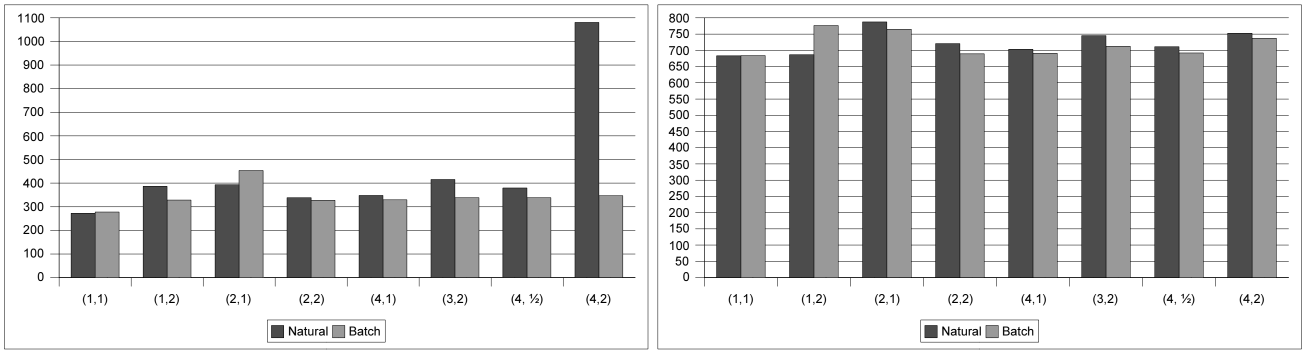

Figure 11: Time required to lookup and start a thread on the Intel platform.

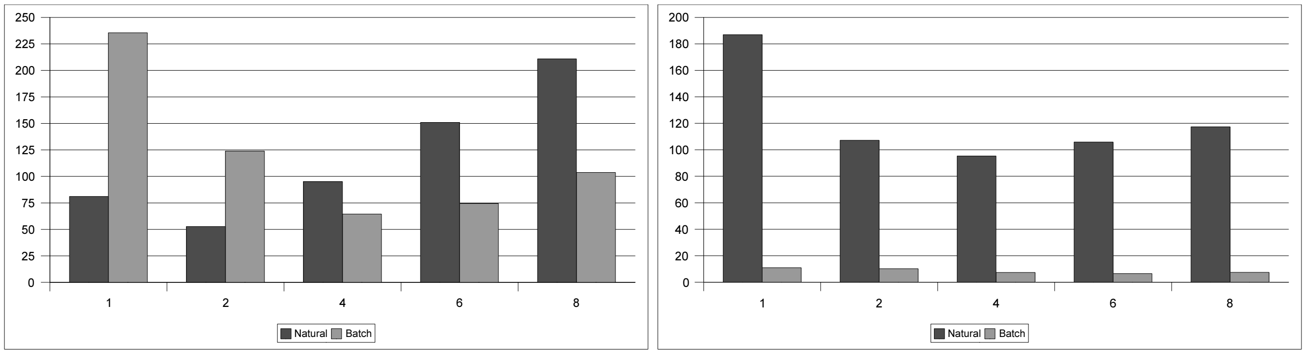

Figure 12: Time required to create and enqueue a thread on SMTSIM.

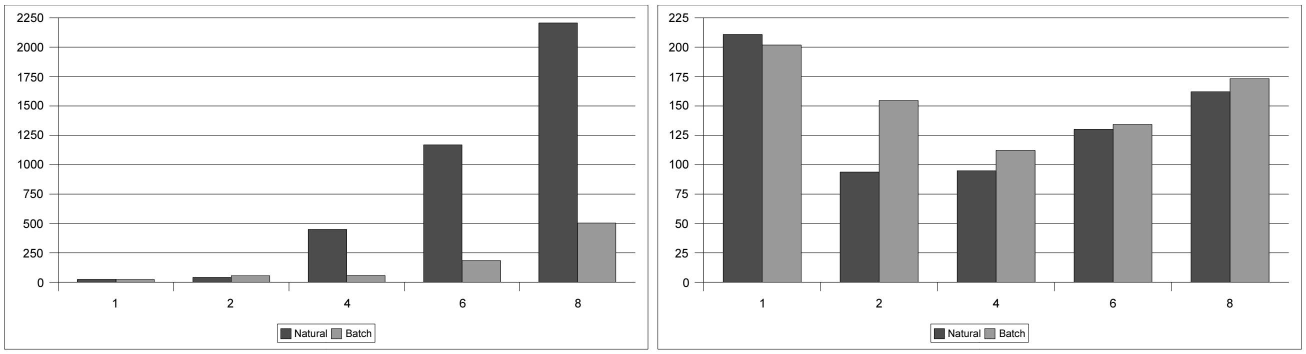

Figure 13: Time required to lookup and start a thread on SMTSIM.

References

[1] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proceedings of the 32nd International Conference on Parallel Processing*, pages 547–554, Kaohsiung, Taiwan, October 2003.

[2] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Proceedings of the 5th Workshop on Massively Parallel Processing*, Denver, Colorado, April 2005.

[3] J. Feo. An Analysis Of The Computational And Parallel Complexity Of The Livermore Loops. UCRL-95708, Lawrence Livermore National Laboratory, 1986.

[4] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, Volume 37, Issue 1:5–20, August 1996.

[5] S. Hummel, E. Schonberg, and L. Flynn. Factoring: a Practical and Robust Method for Scheduling Parallel Loops. In *Proceedings of Supercomputing 1991*, pages 610–632, Albuquerque, USA, 1991.

[6] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, Volume 6, Issue 1:4–15, February 2002.

[7] X. Martorell, J. Labarta, N. Navarro, and E. Ayguade. A Library Implementation of the Nano-Threads Programming Model. In *Proceedings of the 2nd International EuroPar Conference*, pages 644–649, Lyon, France, August 1996.

[8] F. McMahon. The Livermore FORTRAN Kernels Test of the Numerical Performance Range. *Performance Evaluation of Supercomputers, Elsevier Science B.V., North Holland, Amsterdam*, Volume 4:143–186, 1988.

[9] E. Mohr, D. A. Kranz, and Jr. R. H. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, Volume 2, Issue 3:264–280, July 1991.

[10] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé. Is Data Distribution Necessary in OpenMP? In *Proceedings of Supercomputing'2000: High Performance Computing and Networking Conference*, Dallas, TX, November 2000.

[11] C. Polychronopoulos, N. Bitar, and S. Kleiman. Nanothreads: A User-Level Threads Architecture. Technical Report 1297, CSRD, University of Illinois at Urbana-Champaign, 1993.

[12] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1485–1495, December 1987.

[13] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/MP : Integrating Futures into Calling Standards. Technical Report TR 99-01, University of Tokyo, 1999.

[14] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, S. Margherita Ligure, Italy, 1995.

[15] I. E. Venetis and T. S. Papatheodorou. A Time and Memory Efficient Implementation of the Nano-Threads Programming Model. Technical Report HPCLAB-TR-210106, High Performance Information Systems Laboratory, January 2006.

[16] I. E. Venetis and T. S. Papatheodorou. Tying Memory Management to Parallel Programming Models. In *Proceedings of the 2006 European Conference on Parallel Computing (EuroPar 2006)*, pages 666–675, Dresden, Germany, August 2006. Springer Verlag, LNCS Vol. 4128.