*Communicating Process Architectures – 2002*
*James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)*
*IOS Press, 2002*

# Cache-Affinity Scheduling for Fine Grain Multithreading

Kurt DEBATTISTA, Kevin VELLA and Joseph CORDINA
*Department of Computer Science and Artificial Intelligence*
*University of Malta*
*Msida MSD06, Malta, Europe*
*Email:* {`kurt,kvel,jcord`}`@cs.um.edu.mt`

**Abstract.** Cache utilisation is often very poor in multithreaded applications, due to the loss of data access locality incurred by frequent context switching. This problem is compounded on shared memory multiprocessors when dynamic load balancing is introduced and thread migration disrupts cache content. In this paper, we present a technique, which we refer to as 'batching', for reducing the negative impact of fine grain multithreading on cache performance. Prototype schedulers running on uniprocessors and shared memory multiprocessors are described, and finally experimental results which illustrate the improvements observed after applying our techniques are presented.

## 1 Introduction

Multithreaded applications, particularly those of the fine grain variety, deal a severe blow to effective cache use. Frequent context switching disrupts the operation of the locality principle, on which cache hits depend. This problem manifests itself on uniprocessors as well as on shared memory multiprocessors, but in the latter case further issues related to thread migration emerge.

With fine grain threads, descheduling points tend to occur very frequently. The expense of repopulating the processor's cache with a newly dispatched thread's footprint becomes significant when viewed in relation to the shortened thread dispatch time. To make matters worse, an individual thread is unlikely to accumulate a significant cache footprint by itself: only when threads are considered in groups can a long term cache footprint be identified. Coarse grain threads and operating system processes do not suffer from either of these shortcomings, as their dispatch times are long enough to push this cost into relative insignificance.

On multiprocessors, applications execute well only when all processors are sharing the work load and running threads are located close to the data they access. A system that promotes load balancing aims to ensure all processors service an equal workload through the use of a central or shared run queue. While load balancing is commendable, the shared run queue will frequently migrate descheduled threads onto processors on which they would have not recently executed. This will often mean that the newly dispatched thread is unable to find its data footprint in cache, eventually leading to cache misses in the order of tens to hundreds of nanoseconds. Furthermore, Anderson *et al.* [1] note how fine grain multithreading is more susceptible to thread migration due to the frequency of inter-thread communication and the large number of threads being executed concurrently.

On the other hand, cache-affinity schedulers traditionally utilise per-processor run queues. Squillante and Lazowska [2] discuss the benefits of allowing a process to develop affinity to a processor. Markatos and LeBlanc [3] also demonstrate the benefits of locality management. Subsequently, in [4] they observe that the current imbalance between ever-faster processors and slow memory technology will induce a move towards locality awareness in scheduler design. In particular, they propose techniques to improve performance of fine grain thread scheduling using 'memory conscious scheduling' [5], which groups threads that access the same data areas.

In the light of the above discussion, a conflict between load balancing and locality is evident. A scheduling algorithm that complies with the load balancing policy would attempt to migrate threads to balance work load evenly, disregarding the thread's last dispatch locale. Conversely, locality conscious scheduling would opt for threads to spend their entire execution life time on the same processor neglecting that another processor might be left idle. Ideally, scheduling algorithms should aspire to include both policies.

Through the use of novel scheduling algorithms which group fine grain threads together into coarser grain entities termed batches [6], we attempt to improve cache exploitation on uniprocessors. Furthermore, on shared memory multiprocessors, such batching algorithms maintain data locality and minimise contention for shared scheduler data structures while still maintaining a balanced workload. In batch schedulers, the scheduled entity becomes the batch. Processors obtain batches from a common batch pool and service the threads on the batch for a number of dispatches maintained by a dispatch counter. When the dispatch counter threshold is considerably larger than the batch size and the entire memory footprint of the batch fits in the cache, each thread is guaranteed to find the data it requires in the cache (assuming that the underlying kernel thread is not descheduled by the operating system, a relatively rare event). This is bound to alleviate the memory access locality problem in fine grain multithreaded applications. On multiprocessors, batching reaps further benefits by decreasing the incidence of false sharing since threads accessing data in a common cache line could be batched together. Moreover, when balancing load across processors, migrating threads in batches alleviates the contention that arises when migrating multitudes of individual threads.

## 2 Related Work

In the context of our discussion, SMP scheduling strategies can be broadly classified into two camps. On one side are the more commonly used shared run queue schedulers, which make load balancing their main priority. At the other end there are the per-processor run queue configurations, which naturally provide cache-conscious scheduling and reduce run queue contention. In this section we survey work related to thread scheduling using both policies.

Shared run queue schedulers adhere strongly to the load balancing policy since the approach is ideal for distributing threads equally amongst processors. When a processor is idle it is assigned a job from the queue, when it is preempted (or the job finishes) it selects another from the shared queue. Shared run queue's load balancing policies are attractive since it is easy to ensure that no processor is ever idle when there is work to be done. The shared run queue strategy unfortunately, neglects the principal of locality, since threads are usually migrated across processors, without exploiting cache. However, the absence of locality is not the only issue related to shared run queue schedulers, which traditionally rely on spin locks as a method of access control for shared data structures. An oft-overlooked alternative method of synchronisation is available through the considered use of lock-free structures and algorithms, which dispense with the serialisation of concurrent tasks. Lock-free data structures rely on powerful hardware atomic primitives and careful ordering of instructions to protect them from unsafe concurrent access. Valois [7] discusses lock-free techniques in detail and supplies various definitions of relevance to us. A lock-free data structure is termed non-blocking if an operation on it is guaranteed to complete in finite time. Moreover, if every such operation on the data structure is guaranteed to complete the structure is said to be wait free. Results appearing in [8] indicate that non-blocking data structures outperform their lock-based counterparts. Furthermore, when used for thread scheduling and inter-thread synchronisation, non-blocking data structures have other advantages, including stronger fault tolerance, deadlock freedom and, in priority-based schedulers, elimination of priority inversion. Unfortunately, the non-blocking algorithms usually rely on retries to recover from unexpected alterations performed concurrently by other processors. This can result in unpredictable delays and starvation under high contention. Furthermore, the use of COMPARE-AND-SWAP in most of the algorithms introduces the *ABA* problem, which necessitates the use of costly memory management techniques to avoid it. On the other hand, wait-free data structures, as discussed by Herlihy [9], do not suffer from the *ABA* problem and do not ever require retries. As a consequence, starvation is eliminated and the maximum number of instructions executed in the algorithms is fixed at compile-time. No efficient wait-free concurrent queue algorithm is available ruling out a shared run queue version of such an algorithm. We will present a per processor run queue scheduler using wait-free data structures in Section 6.

Per processor run queue schedulers befit the principle of locality. On NORMAs, Eager *et al.* [10] favour per processor run queues due to the high costs associated with migration. Bellosa [11] arrives at similar conclusions regarding NUMA multiprocessors. However, the general trend on UMA multiprocessors was to base scheduling policies on shared run queue models, to balance loads evenly [12, 13]. Anderson *et al.* [1] observe that the central queue can be a cause of great contention and find that per processor run queues perform better than shared run queues when scheduling threads at the user level. Anderson's scheduling strategy involves placing new threads on the run queue of the processor on which it was created. Processors first check their own run queue for threads to execute, if none are found they scan through other run queues. Threads that are created on one processor generally spend their entire execution time on that processor's run queue, thus maintaining cache affinity. Rarely do threads migrate to other processors. Based on the above results thread packages such as Filaments [14] and Cilk [15] adopt per processor run queues as the scheduling strategy. Cilk's per processor run queue algorithm is based on Arora, Blumofe and Plaxton [16] non-blocking double-ended queue (deque). Their non-blocking version of the work stealing algorithm permits processors to access their local deque only by pushing and popping from the bottom (effectively a local deque functions as a stack), while idle processors acquire work by popping threads off the top of the queue. Results [15] demonstrate the system to be highly efficient due to the combination of per processor run queues and non-blocking data structures.

## 3 Scheduler Design Overview

To investigate the effects of different scheduler configurations on the performance of multithreaded applications, we have implemented a suite of uniprocessor and multiprocessor user-level thread schedulers [17] under the collective name of smash. All uniprocessor and SMP smash schedulers borrow many basic ideas, such as active context switching, from CERN's uniprocessor MESH [18] as well as uniprocessor CCSP [19]. Of the various uniprocessor schedulers we have implemented, we consider two: a basic uniprocessor scheduler (uni-smash) and a batch based uniprocessor scheduler (unibatch-smash). We also consider three SMP schedulers: a traditional lock-based shared run queue SMP thread scheduler (shared-smash), a lock-based batch thread scheduler (smpbatch-smash) and our innovative wait-free SMP thread scheduler (wf-smash).

Each scheduler takes the form of a C library, and application programs are written in C and linked to the library. The schedulers provide functionality to create, execute and join (barrier synchronisation on thread termination) threads, as well as facilities for inter-thread communication and synchronisation by means of CSP [20] constructs.

Both the uniprocessor and SMP thread schedulers schedule threads entirely at the user level. A kernel thread is permanently bound to each processor, and user-level threads are scheduled atop the kernel threads. In the case of the SMP scheduler's, each processor's identity is stored in a reserved register for fast retrieval (thus avoiding the use of `getpid()`). Since the schedulers are expected to operate in a multiprogrammed environment, an idle kernel thread (that is, with no user-level threads to execute) sleeps on a kernel semaphore, rather than busy-waiting. This is the only instance in which we make use of a system call within the scheduler (excluding scheduler start-up and shut-down, of course).

uni-smash is our vanilla uniprocessor scheduler composed of a single circular queue acting as the run queue. shared-smash is based on the designs of SMP KRoC [13] and the University of Malta's SMP MESH implementation [12]. Processors acquire threads by accessing the shared run queue. This scheduling strategy ensures that no processor is idle when there are threads on the run queue. This methodology favours load balancing at the expense of minimal cache reuse. Access control to the shared run queue is taken care of by a spin lock. The shared run queue is the main cause of poor performance when scheduling fine grained applications. unibatch-smash will be described in Section 4, smpbatch-smash in Section 5 and wf-smash in Section 6.

## 4 Uniprocessor Batch Scheduling

It is clear that fine grain multithreading has an adverse impact on locality, and thus on the ability of cache hardware to have the necessary data in the cache when it is needed. On shared memory multiprocessors these cache miss problems are compounded by frequent process migration as well as false sharing, when processes executing on separate processors repeatedly write to memory addresses on the same cache line. On uniprocessors, we will demonstrate that performance loss through cache misses alone may be substantial, but can be avoided through specially designed scheduling policies and algorithms.

We introduce batching as a scheduling strategy which improves cache exploitation in fine grain multithreading systems in its uniprocessor incarnation unibatch-smash, and analyse its performance compared to uni-smash in Section 7. In Section 5, batching is proposed as a technique to mitigate efficiency problems associated with fine grain parallelism on shared memory multiprocessors. Uniprocessor batching, illustrated in Figure 4, consists of organising runnable threads in queues of queues. The run queue is transformed into a queue of batches of threads which we will refer to as the batch run queue. Each batch is itself organised as a queue of (possibly related) runnable threads grouped together into a coarser grain entity. A batch has two variables associated with it: a size counter `BSize` and a dispatch counter `BCount`. The batching strategy also includes a threshold size, `MAXBSIZE`, which `BSize` cannot exceed. New batches are created by old batches overflowing, and batches whose constituent threads all terminate are destroyed. Whenever a thread launches a new thread, the newly created thread is placed on the current batch. However, if the current batch has met its size limit, the thread is placed onto a temporary batch which we call the overflow batch. The purpose of the overflow batch is to store excess threads until the overflow batch's `BSize` meets the size threshold. When an overflow batch is full it is placed onto the batch run queue. When scheduling a new batch, the scheduler picks up the batch at the top of the batch run queue, making it the current batch. The dispatch time of a batch is much longer

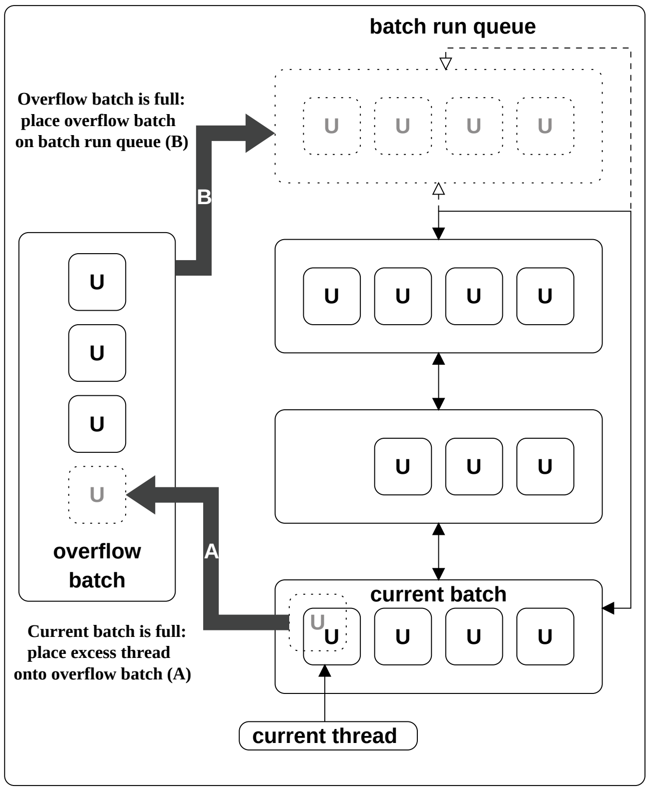

batch run queue
Overflow batch is full: place overflow batch on batch run queue (B)
B
A
overflow batch
current batch
Current batch is full: place excess thread onto overflow batch (A)
current thread
U

Figure 1: Scheduler data structures and information flow in uniprocessor batching.

than that of an individual thread, and when `BCount`, reaches a threshold, `MAXBCOUNT`, the batch is appended to the end of the batch run queue and the next batch is scheduled.

The rationale behind this strategy is that the primary benefactors of caches are threads which access the same areas of memory repeatedly over a long period of time, and the ones which are thwarted by multithreading are those which deschedule frequently over their lifetime. This is because intervening threads tend to push the relevant memory areas out of the cache. If the batch size is limited and the dispatch period for a batch is long enough, threads in that batch will get rescheduled several times within the batch's dispatch period, often finding their memory still intact in the cache. This is because the relatively smaller number of intervening threads will not have managed to wipe out the required cache content. Moreover, if communicating processes gather onto the same batch, they will both get rescheduled on that batch time after time, with high probability, and will therefore still be able to benefit from this scheme.

## 5 SMP Batch Scheduling

The idea of batching processes increases in relevance on shared memory multiprocessors. The coarser grain nature of batches (compared to individual threads) in the SMP case helps not only in improving locality due to cache affinity, but also in reducing false sharing and decreasing contention for shared resources. We present two algorithms that we use to migrate batches thus effectively balancing load amongst processors. The first algorithm implemented in our scheduler smpbatch-smash is described in this section, and only requires the use of a `spinlock` construct. The second, wf-smash, described in Section 6, uses powerful atomic primitives which are not available on all architectures, to construct wait-free data structures reducing the contention for shared data structures to a minimum.

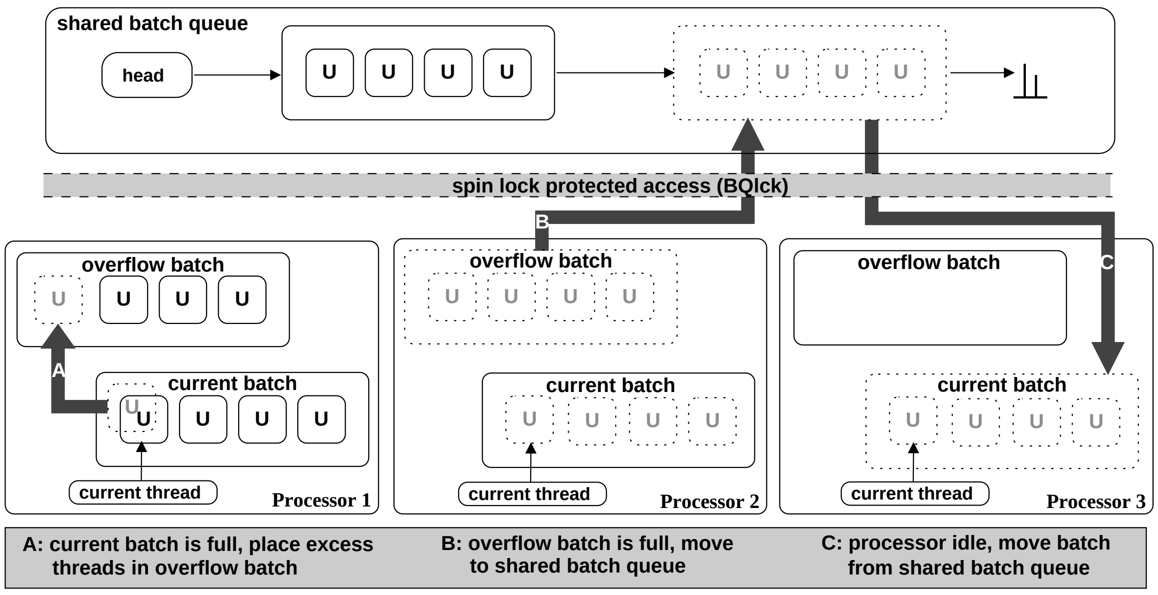

shared batch queue
head
U U U U
U U U U
spin lock protected access (BQlck)
B
C
A
overflow batch
current batch
current thread
Processor 1
Processor 2
Processor 3
A: current batch is full, place excess threads in overflow batch
B: overflow batch is full, move to shared batch queue
C: processor idle, move batch from shared batch queue

Figure 2: Scheduler data structures and information flow in shared run queue SMP batching.

## 5.1 Shared Run Queue Batch Scheduling

smpbatch-smash data structures consist of an overflow batch and current batch, which are local to each processor and a shared batch queue. A three processor configuration is illustrated in Figure 2. As with unibatch-smash each processor maintains a separate `BCount` counter, incrementing it every time a new thread from the batch is dispatched, and reinserting the batch in the batch run queue when it reaches `MAXBCOUNT`. The batch size is maintained by incrementing and decrementing `BSize` as processes join and leave the batch during its dispatch period.

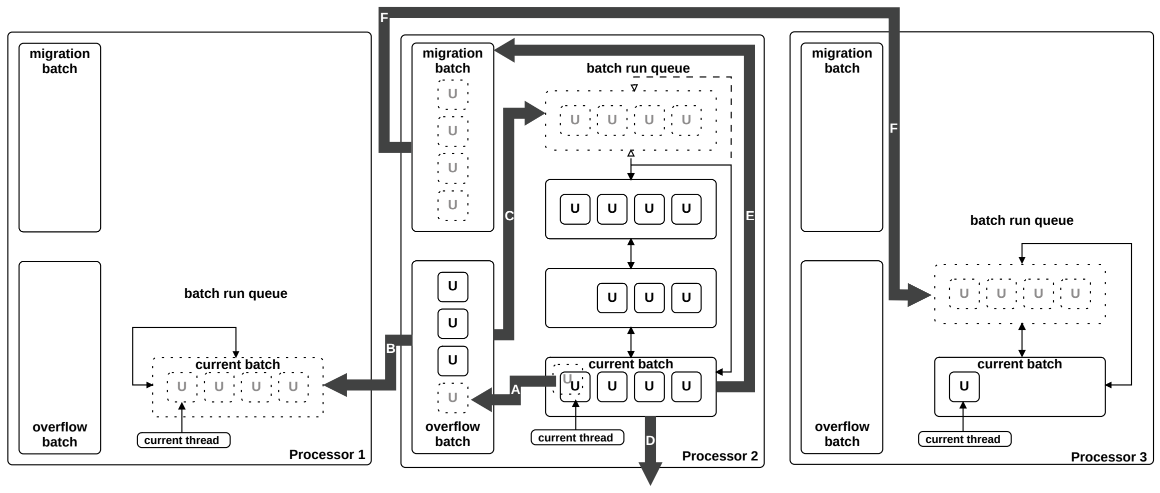
The shared batch run queue obeys the shared run queue model. Access control is maintained by means of a lock `BQlck`. Idle processors retrieve threads in the form of batches by first acquiring `BQlck` then dequeuing a batch from the shared batch queue. Similarly, processors with excess workloads enqueue overflow batches on to the shared batch queue.

The shared batch queue exhibits the same load balancing properties akin to shared run queues, and it is easy to ensure that no processor is ever idle when there is a batch on the shared batch queue. Each processor's overflow batch is used in very much the same way as in the uniprocessor batching implementation, to store surplus threads when the current batch size exceeds `MAXBSIZE`, without causing frequent expensive accesses to the shared batch queue. When the overflow queue itself overflows, it is offloaded onto the shared run queue in the normal manner, acquiring `BQlck` to protect against concurrent access. Since a batch footprint is limited in this way, the cache-related improvements that were observed in our analysis of uniprocessor batching can be preserved here. This time, the improvements are compounded by the other factors peculiar to SMPs.

This activity reflects the workings of a shared run queue model, albeit at a macro level. The thread entity has been replaced by the batch, which has a much longer collective dispatch time, thus reducing the frequency of shared queue access. The expense of retrieving batches from the shared queue remains minimal, so locking duration has not been drastically increased. Accordingly, contention for shared resources should be greatly reduced. In addition, a thread will get scheduled to execute on the same processor on several successive occasions, as the time during which a batch is associated with a processor is substantially longer than the dispatch time of a single thread. There is therefore a greater chance of the thread finding a portion of its working set intact in the processor's cache on being scheduled, especially if the combined working set size of the threads in the batch does not exceed the cache size.

## 6 Wait-Free Batch Scheduling

As with smpbatch-smash, wf-smash makes use of batches as a coarser grain entity to reduce contention and enhance locality. wf-smash is particularly scalable in terms of batch size. In fact results demonstrate the algorithms work well even when migrating single threads for fine grain multithreaded applications [21]. Since wf-smash makes use of wait-free data structures contention is kept at a minimum and batching is used mainly as a vehicle for cache-coherence, even though batches reduce the frequency at which relatively high latency atomic operations (like SWAP and COMPARE AND SWAP) are performed. wf-smash's scheduling algorithms are discussed in further detail in [21].



A: current batch is full, place excess threads in overflow batch
D: current batch dispatch expires, try sender initiated migration
B: overflow batch is full, try sender-initiated migration
E: if D fails, try to place batch on migration queue
C: if B fails, move overflow batch to batch run queue
F: processor about the become idle, attempt receiver initiated migration

Figure 3: Scheduler data structures and information flow in wait-free SMP batching.

wf-smash data structures consist of a batch run queue and overflow batch for each processor which function similarly to unibatch-smash and an additional migration batch. Figure 3 shows these data structures. To understand the purpose of each structure a brief description of the scheduling mechanism is required. As with unibatch-smash the current batch is directly attached to the batch queue. In this case though it also serves to indicate the processor's state, since the current batch pointer is `NULL` if and only if a processor lies idle. As with other batch schedulers, whenever a thread launches a new thread, the newly created thread is placed on the current batch and if the current batch's `BSize` has met `MAXBCOUNT` the thread is placed onto the overflow batch. In this case when the overflow batch becomes full, an attempt is made to migrate the thread to any idle processor. If sender initiated migration fails, the overflow batch is placed onto the local batch run queue. The migration batch is used when a processor has excess workload and wants to offload a batch, but none of the other processors are currently idle. In this case the excess batch is placed onto that same processor's migration batch. A processor with a reduced workload can hence obtain more work by scanning the other processors' migration batches. To improve load balancing, when the

```
// ... launch ARRAY_SIZE/GRANULARITY threads running test_cache()

cthread threads[ARRAY_SIZE/GRANULARITY];
int array[ARRAY_SIZE];

void cache_test(cthread * ct, int i) {
    int j,k,g;
    g = GRANULARITY * i;
    for (j = 0; j < PROCESS_LENGTH; j++) {
        for (k = 0; k < GRANULARITY;k++) {
            array[g + k]++;
        }
        cthread_yield();
    }
}
```

Figure 4: The benchmark algorithm.

current's batch dispatch counter, `BCount` expires and the batch run queue consists of more than one batch, the current batch can also be migrated.

Batches are migrated in one of two distinct ways. Sender-initiated migration occurs when a processor has an excess batch, in which case it attempts to migrate a batch onto an idle processor. Receiver-initiated migration occurs when a processor runs out of work to do, and this idle processor attempts to migrate a batch from one of the other processors' migration batches. The key to the wait-free algorithms is that the data structures are manipulated locally before being placed onto the globally accessible shared data structure pointers. The shared data structures are accessed only by means of atomic COMPARE AND SWAP and SWAP instructions to maintain their integrity. Further details of the wait-free migration algorithms and other aspects of wf-smash are available in [21].

## 7 Results

In this section we analyse the performance of our uniprocessor batch scheduler (unibatch-smash) and the SMP batching implementations, smpbatch-smash and wf-smash, compared with the standard uniprocessor (uni-smash) and shared run queue implementation shared-smash.

Tests were performed on a quad-processor machine with 256MB RAM. The processors were PIII Xeons running at 700MHz, each equipped with 512KB of second-level cache (as well as 16KB instruction and 16KB data first-level caches). The underlying operating system was Linux (kernel version 2.2.12-20). For each of the experiments involving unibatch-smash, `MAXBSIZE` was set to 32 and `MAXBCOUNT` was set to 128.

In order to test the performance of our batching schedulers, we present a benchmark (Figure 4) which performs a series of operations on the same array data. Three variables are used, an array size (`ARRAY_SIZE`) which represents the size of the data that will be handled, a variable representing the thread's longevity (`PROCESS_LENGTH`) and a granularity variable (`GRANULARITY`). Granularity determines the number of computations between each communication and segments the area of the data to be computed by each thread accordingly. Thus, we can vary the granularity without changing the problem size. Effectively the number of threads launched would be set to `ARRAY_SIZE/GRANULARITY`.

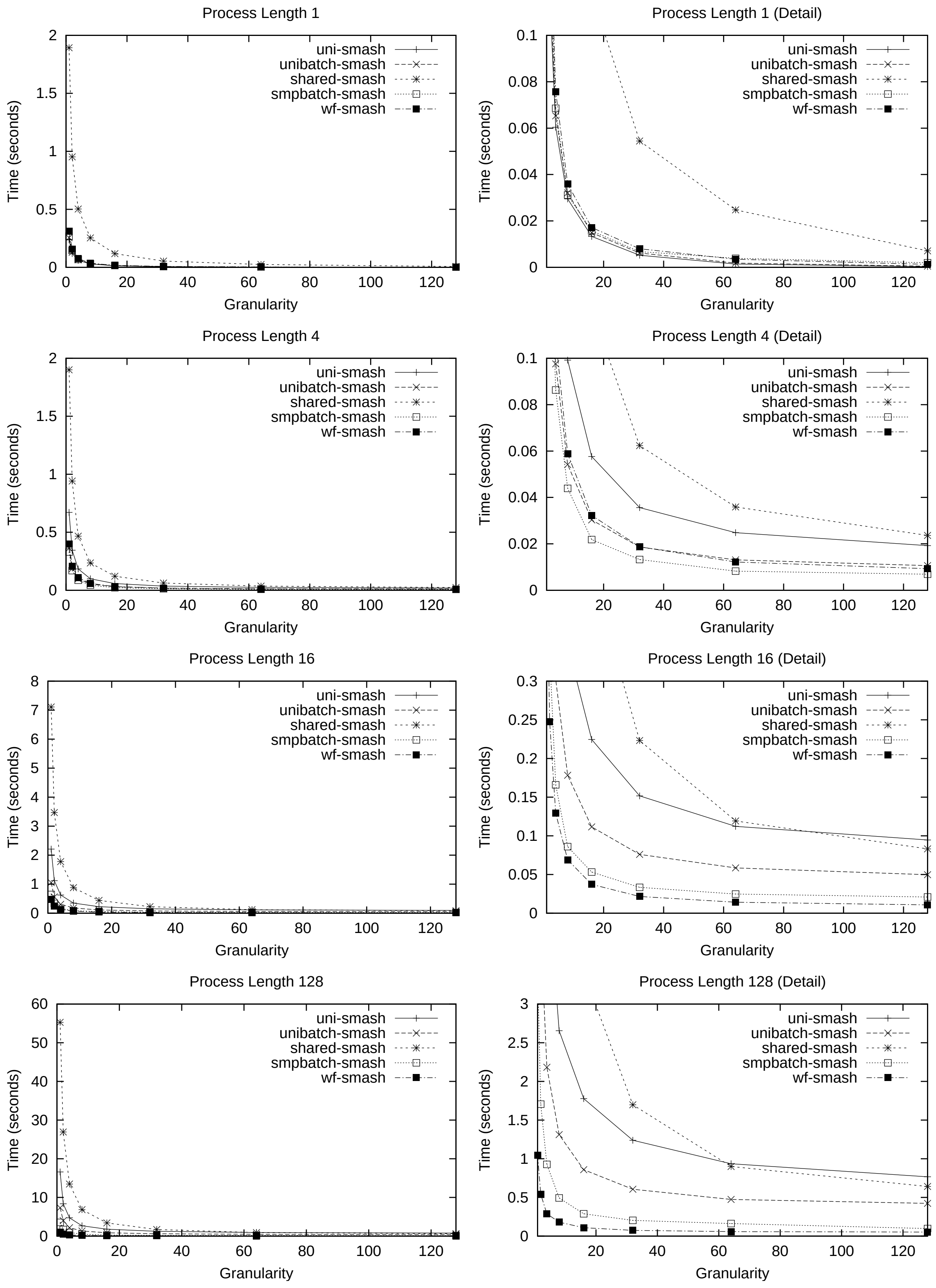


Figure 5: Benchmark results for an array size ARRAY_SIZE fixed at 262,144. SMP schedulers (shared-smash, smpbatch-smash and wf-smash utilise four processors, while the uniprocessor schedulers (uni-smash and unibatch-smash) make use of only one processor.

In our experiment, GRANULARITY was varied from $2^0$ (1) through to $2^8$ (128), to span the entire range between very fine granularity, where each increment operation is followed by a context switch, and coarser granularity. PROCESS_LENGTH was varied between $2^0$ (1) and $2^8$ (128), to experiment with both short processes that do not reuse cached data, and very long processes that repeatedly access the same data. ARRAY_SIZE was set to $2^{18}$ (262,144), representing large data segments that cannot entirely fit in cache. Results are presented in Figure 5. The different plots represent varying process lengths of 1, 4, 16 and 128, and trace granularity values of 1 to 128 against execution time. The curves indicate execution time for normal uni-smash (upper curve), unibatch-smash, shared-smash, smpbatch-smash and wf-smash (lower curve). The results for the uniprocessor schedulers (uni-smash and unibatch-smash) were taken utilising only one processor. The SMP scheduler (shared-smash, smpbatch-smash and wf-smash) results utilised all four processors available on the system.

For unibatch-smash, improvements of up to 100% have been observed for very fine granularities when compared to uni-smash. As expected, the difference in execution time tails off as GRANULARITY increases. For PROCESS_LENGTH from 16 upwards improvements can be observed even at GRANULARITY of 64. It can be seen that as process length increases, the improvements very quickly reach larger and more sustained levels, since caching is better exploited. In fact, an improvement is seen immediately when PROCESS_LENGTH is raised from 1 to 4. Further experiments, not shown in the graphs, demonstrate that improvements can even be seen at a PROCESS_LENGTH of 2. These results indicate that on uniprocessor machines, implicit context switching overheads due to cache misuse are significantly more pronounced for data sizes that do not fit in the cache.

Comparing the results of the three SMP schedulers for all PROCESS_LENGTHs it can be clearly noted that the performance of shared-smash suffers when compared to smpbatch-smash and wf-smash. For a PROCESS_LENGTH of 1, since none of the threads do not benefit from cache re-use, the improved performance of the batch schedulers compared to shared-smp is due mainly to the batching scheduler's reduced contention for shared data structures. For higher values of PROCESS_LENGTH the improvement is more pronounced since when scheduled on any of the SMP batch schedulers threads exploit the cache.

When comparing results of the SMP schedulers to the uniprocessor schedulers, we note that for a PROCESS_LENGTH of 1, no speedup is achieved by any of the SMP schedulers up to a granularity of 128. Speedup is eventually achieved at higher granularities (not shown in graph). In the case of shared-smash, the main reason for poor performance is due to contention on the run queue. In the case of the SMP batch schedulers, the initial lack of speedup is due to the use of batches. Since the threads are short lived, not all processors are serviced with batches constantly. When the maximum batch size (MAXBSIZE) is reduced, improvements would be seen at finer granularities, particularly for wf-smash. The performance of wf-smash compared to smpbatch-smash is discussed at length in [17]. For larger PROCESS_LENGTH values a marked improvement in speedup is clearly visible for both smpbatch-smash and wf-smash due partially to cache exploitation and partially to the reduction in contention for migration. Results for PROCESS_LENGTHs of 16 and 128 demonstrate that both batch schedulers achieve super scalar speedup due to the added advantage of exploiting the caches on each of the four processors. On the other hand, shared-smash's performance degrades even further when compared to the other SMP schedulers due to the high contention for the shared run queue, further accentuated by the lack of cache exploitation.

## 8 Conclusion and Future Work

We have conducted an investigation into the effectiveness of cache-conscious scheduling using batches. The experimental results obtained indicate that a significant reduction in execution times can be gained through the use of such techniques.

However, further experiments with real applications are required to gather information about performance under more realistic conditions. It should be noted that batch-based thread scheduling as presented here may be subject to problems when the pre-set batch size limit is greater than the total number of threads being executed in the application, since all threads would be serviced by a single processor. While this can be advantageous in identifying situations where parallel processing is not worth the effort, pathological cases may well occur in specific applications. Automatic modification of the batch size limit could be envisaged, whereby the batch limit is dynamically set to match the current application's needs. At the moment, threads are grouped into batches by locality and indirectly through communication, so that threads created on a common processor are placed onto the same batch. An additional grouping criterion could be based on the frequency of inter-thread communication or rely on object-affinity [22, 23]. Furthermore, the application programmer could be given the opportunity to manually specify viable thread groupings to override the automatic batching arrangements adopted by the scheduler.

The investigations presented here fit into the wider context of a general purpose server-side parallel processing system composed of a cluster of shared memory multiprocessors with high speed user-level CSP communication over Gigabit Ethernet between nodes in the cluster, as well as gigabit speed user-level TCP/IP connectivity to the outside world [24]. Many of the constituent components have already been developed or are at an advanced stage of development.

## References

[1] Anderson, T., Bershad, B., Lazowska, E. and Levy, H. *The Performance Implications of Thread Managment Alternatives for Shared-Memory Multiprocessors.* In *IEEE Transactions on Computers*, 38(12):1631-1644, December 1989.

[2] Squillante M. and Lazowska E. *Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems*, 4(2):131-143, February 1993.

[3] Markatos, E. and LeBlanc, T. *Load Balancing vs. Locality Management in Shared-Memory Multiprocessors.* Technical Report 399, Computer Science Department, University of Rochester, October 1991.

[4] Markatos E. and LeBlanc T. *Shared Memory Multiprocessor Trends and the Implications for Parallel Program Performance.* Technical Report 420, Computer Science Department, University of Rochester, May, 1992.

[5] Markatos E. and LeBlanc T. *Memory Conscious Scheduling in Shared-Memory Multiprocessors.* Technical Report, Computer Science Department, University of Rochester, December, 1991.

[6] Vella, K. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations.* Ph.D. Thesis, University of Kent at Canterbury, December 1998.

[7] Valois, J. *Lock-Free Data Structures.* Ph.D. Thesis, Renesselaer Polytechnic Institute, New York, 1995.

[8] Michael, M. and Scott, M. *Non-Blocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. Journal of Parallel and Distributed Computing*, 1998.

[9] Herlihy M. *Wait-Free Synchronization.* ACM Transactions on Programming Languages and Systems, 11(1):124-149, January 1991.

[10] Eager, D., Lazowska, E. and Zahorjan, J. *The Limited Performance Benefits of Migrating Active Processes for Load Sharing.* In *Proceedings of ACM SIGMETRICS*, May 1988.

[11] Bellosa, F. *Memory conscious scheduling and processor allocation on NUMA architectures.* Technical Report TR-14-06-95, Computer Science Department, University of Erlangen-Nurnberg, May 1995.

[12] Cordina, J. *Fast Multi-Threading on Shared Memory Multiprocessors.* B.Sc. I.T. Final Year Project Report, Department of Computer Science and Artificial Intelligence, University of Malta, May 2000.

[13] Vella, K. and Welch, P. *CSP/occam on Shared Memory Multiprocessor Workstations.* In *Proceedings of WoTUG 22: Architectures, Languages and Techniques, Volume 57 of Concurrent Systems Engineering*, IOS Press, April 1998.

[14] Engler D., Andrews R. and Lowenthal D. *Filaments: Efficient Support for Fine-Grain Parallelism.* Technical Report TR 93-13, Department of Computer Science, The University of Arizona, February, 1994.

[15] Randall, K. *Cilk: Efficient Multithreaded Computing.* Ph.D. Thesis, Massachusettes Institute of Technology, June 1998.

[16] Arora, N., Blumofe, R. and Plaxton, G. *Thread Scheduling for Multiprogrammed Multiprocessors.* ACM Symposium on Parallel Algorithms and Architectures, pages 119-129, 1998.

[17] Debattista, K. *High Performance Thread Scheduling on Shared Memory Multiprocessors.* M.Sc. Dissertation, University of Malta. February 2001.

[18] Boosten, M., Dobinson, R.W. and van der Stok, P.D.V. *Fine-Grain Parallel Processing on Commodity Platforms.* Volume 57 of *Concurrent Systems Engineering*, pages 263-275. IOS Press, April 1999.

[19] Moores, J. *CCSP - A portable CSP-based run-time system supporting C and occam.* Volume 57 of *Concurrent Systems Engineering Series*, pages 147-168, IOS Press, April 1999.

[20] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[21] Debattista, K. and Vella, K. *High Performance Wait-Free Thread Scheduling on Shared Memory Multiprocessors.* In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV. June 2002.

[22] Chandra, R., Gupta, A. and Hennessy, J. *Data locality and load balancing in COOL.* In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May, 1993.

[23] Fowler R. and Kontotnanassis L. *Improving Processor and Cache Locality in Fine-Grained Parallel Computations using Object-Affinity Scheduling and Continuation Passing.* Technical Report TR411, Department of Computer Science, University of Rochester, 1992.

[24] Cordina, J. *High Performance TCP/IP for Multi-Threaded Servers.* M.Sc. Dissertation, University of Malta. March 2002.