A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier*,
Fabien Lementec, and Bruno Raffin

MOAIS Project, INRIA- LIG
(marc.tchiboukdjian, vincent.danjean, fabien.lementec,
bruno.raffin)@imag.fr thierry.gautier@inrialpes.fr

Abstract. Reordering instructions and data layout can bring significant performance improvement for memory bounded applications. Parallelizing such applications requires a careful design of the algorithm in order to keep the locality of the sequential execution. In this paper, we aim at finding a good parallelization of memory bounded applications on multicore that preserves the advantage of a shared cache. We focus on sequential applications with iteration through a sequence of memory references. Our solution relies on a work stealing scheduler combined with a dynamic sliding window that constrains cores sharing the same cache to process data close in memory. This parallel algorithm induces the same number of cache misses as the sequential algorithm at the expense of an increased number of synchronizations. Experiments with a memory bounded application confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs.

1 Introduction

Many applications in scientific computing are memory bounded. Favoring the locality of access patterns through data and computation reordering can bring significant performance benefits. When designing parallel algorithms, one must be extra careful not to lose the locality of the sequential application, which is the key for good performance. In most last generation multicores, the last level of cache is shared among all cores of the chip. For instance the Intel Nehalem, the AMD Phenom and Opteron (only for the quadcores and hexacores) and the IBM Power7 all have a shared L_3 cache.

In this paper, we focus on one specific aspect of the parallelization of memory bounded applications: how to adapt the scheduling to take advantage of the shared caches of multicore processors. The goal is to propose a scheduling algorithm that improves performance by reducing cache misses, compared to parallel algorithms that do not take into account the shared cache amongst several cores. We propose to have cores working on independent but close (regarding the memory layout) data sets that can all fit in the shared cache. If a core needs a data that is not in its data set, there is a good chance it will find it in the data set loaded in the

* Part of this work was done while the third author was visiting the ArTeCS group of the University Complutense, Madrid, Spain.

cache by one of its neighbors, thus saving cache misses. The algorithm behaves as if each core would benefit from a full-size private cache, at the price of a few extra synchronizations required to ensure a proper collaboration between cores.

This paper focuses on algorithms that take an input sequence to produce an output sequence of results. Such algorithms encompass many of the C++ Standard Template Library (STL) functions like `for_each` or `transform`. Moreover, many parallel libraries such as Intel TBB or the GNU STL parallel mode provide parallel implementations of the STL. Thus providing shared cache aware parallelizations of these algorithms can improve performance of many applications.

We provide a cache constraint that parallel algorithms should respect to induce no more cache misses than the sequential algorithms. We present two new algorithms respecting this cache constraint and two implementations, one based on PThread and the other one based on work-stealing allowing efficient dynamic load balancing. We also implement those new algorithms with the parallel library TBB and the GNU parallel STL and compare them with our implementations.

2 Scheduling for Efficient Shared Cache Usage

2.1 Window Algorithms for Sequence Processing

We consider algorithms that take an input sequence $i_1, i_2, \ldots, i_n$ (different input elements can share some data) and a function *op* to be applied on all elements of the input producing an output sequence $o_1, o_2, \ldots, o_{n'}$. Notice that treating one element may produce a different number of elements in the output sequence. Most STL algorithms are variations over this model. The sequential algorithm processes the sequence in order from i_1 to i_n. We assume that the sequential algorithm already performs well with respect to temporal locality of data accesses. Data processed closely in the sequential execution are also close in memory. We focus on the case where all elements of the sequence can be processed in parallel.

We introduce two parallel algorithms to process such a sequence in parallel. These two algorithms are parameterized by m, the maximum distance between the threads. In the first one, denoted *static-window*, the sequence is first divided into n/m chunks of m contiguous elements. Then, each chunk is processed in parallel by the p processors sharing the same cache. Several strategies can be used to parallelize the processing of each chunk. The m elements could be statically partitioned into p groups of m/p elements, one per processor, or a work-stealing scheme can be used to dynamically balance the load. The second parallel algorithm, denoted *sliding-window*, is a relaxed version of the *static-window* algorithm. At the beginning of the algorithm, the first m elements of the sequence are ready and can be processed in any order. Each time the first element i_k not yet processed in the sequence is treated by a processor, it enables the element i_{k+m} at the end of a window of size m. These two algorithms will be compared with an algorithm denoted *no-window* that do not respect the cache constraint. All the elements of the sequence can be processed in any order. This algorithm induces more cache misses than the sequential algorithm and the window algorithms, but it requires fewer synchronizations.

2.2 Cache Performance of Window Algorithms

The re-use distance captures the temporal locality of a program [1]. Let consider a series of memory references $(x_k)_{k\geq 0}$. When a reference x_k access an element for the first time, the re-use distance of x_k is infinite. If the element has been previously accessed, $x_{k'} = x_k$ with $k' > k$, the re-use distance of $x_{k'}$ is equal to the number of distinct elements accessed between these two references x_k and $x_{k'}$. Let h_d denote the number of memory references with a re-use distance d. The number of cache misses of a fully associative LRU cache of size C is equal to $M_{\text{seq}} = \sum_{d=C+1}^{\infty} h_d$. We can extend this definition to sequence processing algorithms: if processing i_k and $i_{k'}$ uses similar data, the re-use distance is $k' - k$.

We consider now p processors sharing the same cache that process the sequence in parallel in distant places like the *no-window* algorithm. As we assumed the sequence has good temporal locality, elements far-away in the sequence use distinct data. In this case, the re-use distance is multiplied by p as to each access of one processor corresponds $p-1$ accesses of the others to distinct elements. Thus, the number of cache misses is $M_{\text{no-win}} = \sum_{d=C+1}^{\infty} h_{d/p} \approx \sum_{d=C/p+1}^{\infty} h_d$. The *no-window* algorithm induces as many cache misses as the sequential algorithm with a cache p times smaller. We now restrain the processors to work on elements at distance less than m like in the window algorithms. Let $r(m)$ be the maximum number of distinct memory references when processing $m-1$ consecutive elements of the input sequence. In the worst case, when processing element i_k, all elements $i_{k+1}, \ldots, i_{k+m-1}$ have already been processed accessing at most $r(m)$ additional distinct elements compared to the sequential order. Thus the re-use distance is increased by at most $r(m)$. The number of cache misses is $M_{\text{window}} \leq \sum_{d=C+1}^{\infty} h_{d-r(m)} = M_{\text{seq}} + \sum_{d=C+1-r(m)}^{C} h_d$. As we assumed the sequence has good temporal locality, $r(m)$ is small compared to m and h_d is small for large d. Therefore $\sum_{d=C+1-r(m)}^{C} h_d$ is small and the window algorithms induce approximately the same number of cache misses as the sequential algorithm.

2.3 PThread Parallelization of Window Algorithms

We present here the implementation of the *no-window* and *static-window* algorithms using PThreads. The PThread implementation allows a fine grain control on synchronizations with very little overhead.

For the *no-window* algorithm, the sequence is statically divided into p groups. Each group is assigned to one thread bound to one processor and all threads synchronize at the end of the computation. For the *static-window* algorithm, the sequence is first divided into chunks of size m. Then each chunk is statically divided into p groups and all threads synchronize at the end of each chunk before starting to compute the next one. Each synchronization is implemented with a `pthread_barrier`. Threads wait at the barrier and are released when all of them have reached the barrier. Although we expect the threads in the *static-window* algorithm to spend more time waiting for other threads to finish their work, the reduction of cache misses should compensate this extra synchronization cost. The *sliding-window* algorithm has not been implemented in PThread because it

```
typedef struct {
  InputIterator        ibeg;
  InputIterator        iend;
  OutputIterator       obeg;
  size_t               osize;
} Work_t ;

void dowork(...) {
  complete_work:
    while (iend != ibeg) {
      kaapi_stealpoint(..., &splitter);
      for(i=0; i<grain; ++i, ++ibeg)
        op(ibeg, obeg, &osize);
      kaapi_preemptpoint(..., &reducer);
    }
    if ( kaapi_preempt_next_thief(...) )
      goto complete_work ;
} // no more work -> become a thief

void reducer(Work_t *victim, Work_t *thief) {
  memmove( victim->obeg, thief->obeg,
           thief->osize );
  victim->osize += thief->osize;
  victim->ibeg   = thief->ibeg;
  victim->iend   = thief->iend;
} // victim -> dowork / thief -> try to steal
```

```
void splitter( Work_t *victim, int count,
               kaapi_request_t* request ) {
  int i = 0;
  size_t size = victim->iend - victim->ibeg;
  size_t bloc = size / (1+count);
  InputIterator local_end = victim->iend;
  Work_t *thief;

  if (size < gain)
    return;
  while (count >0) {
    if (kaapi_request_ok(&request[i])) {
      thief->iend  = local_end;
      thief->ibeg  = local_end - bloc;
      thief->obeg  = intermediate_buffer;
      thief->osize = 0;
      local_end   -= bloc;
      kaapi_request_reply_ok(thief,
                             &request[i]);
      --count;
    }
    ++i;
  }
  victim->iend  = local_end;
} // victim and thieves -> dowork
```

Fig. 1. C implementation of the adaptive *no-window* algorithm using the KAAPI API.

would require a very complex code. We present in the next section a work-stealing framework allowing to easily implement all these algorithms.

3 Work-Stealing Window Algorithms with Kaapi

In this section, we present the low level API of KAAPI [2] and detail the implementation of the windows algorithms.

3.1 Kaapi Overview

KAAPI is a programming framework for parallel computing using work-stealing. At the initialization of a KAAPI program, the middleware creates and binds one thread on each processor of the machine. All non-idle threads process work by executing a sequential algorithm (`dowork` in fig. 1). All idle threads, the thieves, send work requests to randomly selected victims. To allow other threads to steal part of its work, a non-idle thread must regularly check if it received work requests using the function `kaapi_stealpoint`. At the reception of `count` work requests, a `splitter` is called and divides the work into `count+1` well-balanced pieces, one for each of the thieves and one for the victim.

When a previously stolen thread runs out of work, it can decide to preempt its thieves with the `kaapi_preempt_next_thief` call. For each thief, the victim merges part of the work processed by the thief using the `reducer` function and takes back the remaining work. The preemption can reduce the overhead of storing elements of the output sequence in an intermediate buffer when the final place of an output element is not known in advance. To allow preemption, each thread regularly checks for preemption requests using the function `kaapi_preemptpoint`.

To amortize the calls to the KAAPI library, each thread should process several units of work between these calls. This number is called the *grain* of the algorithm.

In particular, a victim thread do not answer positively to a work request when it has less than *grain* units of work.

Compared to classical WS implementations, tasks (`Work_t`) are only created when a steal occurs which reduces the overhead of the parallel algorithm compared to the sequential one [3]. Moreover, the steal requests are treated by the victim and not by the thieves themselves. Although the victim has to stop working to process these requests, synchronization costs are reduced. Indeed, instead of using high-level synchronization functions (mutexes, etc.) or even costly atomic assembly instructions (compare and swap, etc.), the thieves and the victim can communicate by using standard memory writes followed by memory barriers, so no memory bus locking is required. Additionally, the `splitter` function knows the number `count` of thieves that are trying to steal work to the same victim. Therefore, it permits a better balance of the workload. This feature is unique to KAAPI when compared to other tools having a work-stealing scheduler.

3.2 Work-Stealing Algorithm for Standard (*no-window*) Processing

It is straightforward to implement the *no-window* algorithm using KAAPI. The work owned by a thread is described in a structure by four variables: `ibeg` and `iend` represents the range of elements to process in the input sequence, `obeg` is an iterator on the output sequence and `osize` is the number of elements written on the output. At the beginning of the computation, a unique thread possesses the whole work: `ibeg=0` and `iend=n`. Each thread processes its assigned elements in a loop. Code of Fig. 1 shows the main points of the actual implementation.

3.3 Work-Stealing Window Algorithms

The *static-window* algorithm is very similar to the *no-window* algorithm of the previous section. The first thread owning the total work has a specific status, it is the *master* of the window. Only the master thread has knowledge of the remaining work outside the m-size window. When all elements of a window have been processed, the master enables the processing of the new window by updating its input iterators `ibeg = iend` and `iend += m`. This way, when idle threads request work to the master thread, the stolen work is close in the input sequence. Moreover, all threads always work on elements at distance at most m.

The *sliding-window* algorithm is a little bit more complex. In addition to the previous iterators, the master also maintains `ilast` an iterator on the first element after the stolen work in the input sequence (see Fig. 2). When the master does not receive any work request, then `iend == ilast == ibeg+m`. When the master receives work requests, it can choose to give work on both sides of the stolen work. Distributing work in the interval `[ibeg,iend]` corresponds to the previous algorithm. The master thread can also choose to distribute work close to the end of the window, in the interval `[ilast,ibeg+m]`.

4 Experiments

We base our experiments on a common scientific visualization filter: extracting an isosurface in an unstructured mesh using the marching tetrahedra (MT) algorithm [4].

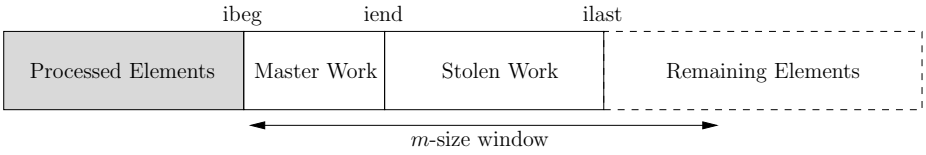

Fig. 2. Decomposition of the input sequence in the *sliding-window* algorithm.

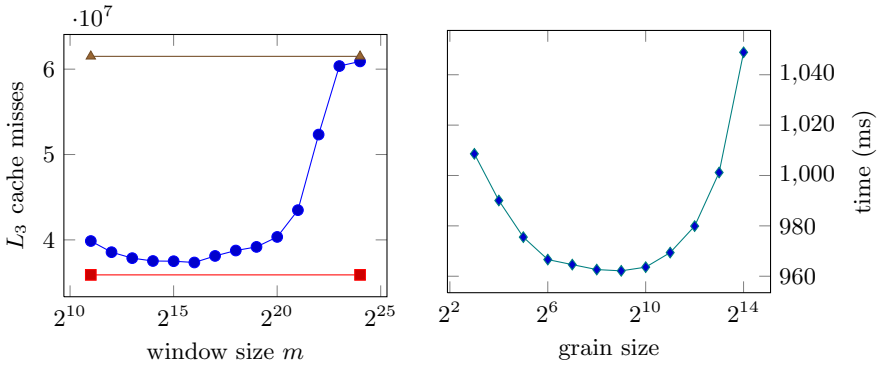

Fig. 3. (Left) Number of L_3 cache misses for the PThread implementation of the *static-window* algorithm for various window sizes compared to the sequential algorithm and the *no-window* algorithm. (Right) Parallel time for the KAAPI implementation of the *static-window* algorithm with various grain sizes. (Both) All parallel algorithms use the 4 cores of the Nehalem processor.

We first calibrate the grain for the work-stealing implementation and the window size m for the window algorithms. Then, we compare the KAAPI framework with other parallel libraries on a central part of the MT algorithm which can be written as a `for_each`. Finally we compare the *no-window*, *static-window* and *sliding-window* algorithms implementing the whole MT.

All the measures reported are averaged over 20 runs and are very stable. The numbers of cache misses are obtained with PAPI [5]. Only last level cache misses are reported as the lower level cache misses are the same for all algorithms. Two different multicores are used, a quadcore Intel Xeon Nehalem E5540 at 2.4Ghz with a shared 8MB L_3 cache and a dualcore AMD Opteron 875 at 2.2Ghz with two 1MB L_2 private caches. If the window algorithms reduce the number of cache misses on the Nehalem but not on the Opteron, one can conclude that this is due to the shared cache.

4.1 Calibrating the Window Algorithms

Fig. 3(left) shows the number of L_3 cache misses for the *static-window* algorithm compared to the sequential algorithm and the *no-window* algorithm. The *static-window* algorithm is very close to the sequential algorithm for window sizes less than 2^{20}. It does not exactly match the sequential performance due to additional `reduce` operations for managing the output sequence in parallel. With bigger windows, L_3 misses increase and tend to the *no-window* algorithm. For the remaining experiments, we set $m = 2^{19}$.

Time (ms)		Nehalem				Opteron			
Algorithms	#Cores	STL	GNU	TBB	KAAPI	STL	GNU	TBB	KAAPI
no-window	1	3,987	4,095	3,975	4,013	9,352	9,154	10,514	9,400
	4		1,158	1,106	1,069		2,514	2,680	2,431
static-window	1	3,990	4,098	3,981	4,016	9,353	9,208	10,271	9,411
	4		1,033	966	937		2,613	2,776	2,598

Table 1. Performance of the *no-window* and *static-window* algorithms on a `for_each` with various parallel libraries. GNU is the GNU parallel library. Time are in ms.

Fig. 3(right) shows the parallel time of the *static-window* algorithm with the KAAPI implementation for various grain sizes. Performance does not vary much, less than 10% on the tested grains. For small grains, the overhead of the KAAPI library becomes significant. For bigger grains, the load balancing is less efficient. For the remaining experiments, we choose a grain size of 128. We can notice that the KAAPI library allows very fine grain parallelism: processing 128 elements takes approximately 3μs on the Nehalem processor.

4.2 Comparison of Parallel Libraries on `for_each`

Table 1 compares KAAPI with the GNU parallel library (from gcc 4.3) (denoted GNU) and Intel TBB (v2.1) on a `for_each` used to implement a central sub-part of the MT algorithm. The GNU parallel library uses the best scheduler (parallel balanced). TBB uses the auto partitioner with a grain size of 128. TBB is faster than GNU on Nehalem and it is the other way around on Opteron. KAAPI shows the best performance on both processors. This can be explained by the cost of the synchronization primitives used: POSIX locks for GNU, compare and swap for TBB and atomic writes followed by memory barriers for KAAPI.

4.3 Performance of the Window Algorithms

We now compare the performance of the window algorithms. Table 1 shows that the *static-window* algorithm improves over the *no-window* algorithm for all libraries on the Nehalem processor. However, on the Opteron with only private caches, performances are in favor of the *no-window* algorithm. This was expected as the Opteron has only private caches and the *no-window* algorithm has less synchronizations. We can conclude that the difference observed on Nehalem is indeed due to the shared cache.

Fig. 4(left) presents speedup of all algorithms and ratio of cache misses compared to the sequential algorithm. The *no-window* versions induces 50% more cache misses whereas the window versions only 13% more. The window versions are all faster compared to the *no-window* versions. Work stealing implementations with KAAPI improves over the static partitioning of the PThread implementations. The *sliding-window* shows the best performance. Fig. 4(right) focus on the comparison of the *sliding-window* and *static-window* algorithms. Due to additional parallelism, the number of steal operations are greatly reduced in the *sliding-window* algorithm (up to 2.5 time less) leading to a 5% additional gain.

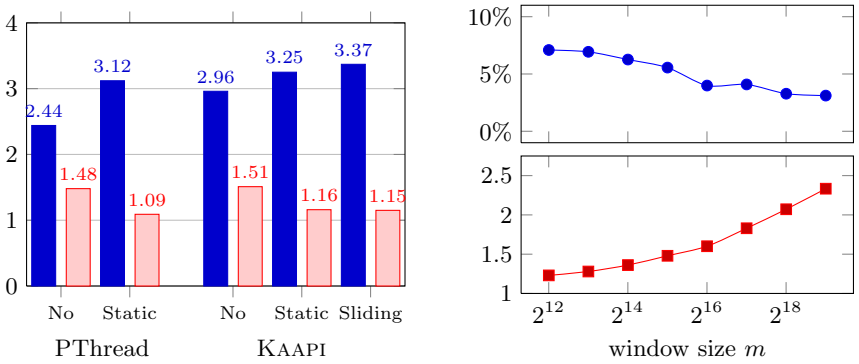

Fig. 4. (Left) Speedup and ratio of increased cache misses over the sequential algorithm for the *no-window*, *static-window* and *sliding-window* algorithms with PThread and KAAPI implementations. (Right) Speedup and ratio of saved steal operations for the *sliding-window* algorithm over the *static-window* algorithm with the KAAPI implementation. (Both) All algorithms run on the 4 cores of the Nehalem.

5 Conclusions

Previous experimental approaches have shown the interest of efficient cache sharing usage, on a recent benchmark [6] and on data mining applications [7].

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [8]. However, none of these techniques take into account the shared cache of multicore processors. Optimization of sequential locality for mesh applications has been studied through mesh layout optimization [9].

The algorithms for parallel sequence processing proposed in this paper focus on exploiting the shared cache of last generation multicores. Experiments confirm that these techniques increase performances by 10% to 30%.

References

1. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: Proc. of ICS. (2003)
2. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO. (2007)
3. Traoré, D., Roch, J.L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel stl algorithms. In: Euro-Par. (2008)
4. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed. Kitware Inc. (2004)
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications **14** (2000)
6. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: PPoPP. (2010)
7. Jaleel, A., Mattina, M., Jacob, B.: Last level cache (LLC) performance of data mining workloads on a CMP. In: HPCA. (2006)
8. Zhang, H., Newman, T.S., Zhang, X.: Case study of multithreaded in-core isosurface extraction algorithms. In: EGPGV. (2004)
9. Tchiboukdjian, M., Danjean, V., Raffin, B.: Binary mesh partitioning for cache-efficient visualization. TVCG **16**(5) (sept.-oct. 2010) 815 –828