HAL
open science

Adaptive Algorithms for Shared Cache on Multicore

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, Bruno Raffin

To cite this version:

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, Bruno Raffin. Adaptive Algorithms for Shared Cache on Multicore. [Research Report] RR-7256, INRIA. 2010, pp.17. inria-00473617

HAL Id: inria-00473617

https://inria.hal.science/inria-00473617v1

Submitted on 15 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Adaptive Algorithms for Shared Cache on Multicore

Marc Tchiboukdjian — Vincent Danjean — Thierry Gautier — Fabien Le Mentec — Bruno Raffin

N° 7256

Avril 2010

Distributed and High Performance Computing

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7256--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche GRENOBLE - RHÔNE-ALPES

Adaptive Algorithms for Shared Cache on Multicore

Marc Tchiboukdjian , Vincent Danjean , Thierry Gautier , Fabien Le Mentec , Bruno Raffin

Theme : Distributed and High Performance Computing
Équipe-Projet moais

Rapport de recherche n° 7256 — Avril 2010 — 14 pages

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. On one hand, parallel computation tends to create concurrent tasks that work on independent data sets to reduce communication and synchronization. On the other hand, a multicore architecture with shared cache can bring performance benefits due to high-speed communication between cores if concurrent tasks process data close in memory. 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 an adaptive parallel algorithm 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. We theoretically analyze the synchronization overhead for both static and dynamic load balancing. Experiments with a memory bounded isosurface extraction application confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs. On quad cores Nehalem processor, our algorithms are 10% to 30% faster than algorithms not optimized for shared cache thanks to a reduced number of last level cache misses.

Key-words: work stealing; cache-efficient algorithms

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Algorithms adaptatifs pour l'utilisation efficace du cache partagé des multicœurs

Résumé : Le réordonnancement des instructions et la réorganisation des données en mémoire peut apporter des important gains de performance pour les applications limitées par les accès mémoire. Paralléliser de telles applications requièrent une conception soignée de l'algorithme pour garder la localité de l'exécution séquentielle. D'une part, les applications parallèles créent des tâches concurrentes qui travaillent sur des jeux de données indépendants pour réduire les communications et les synchronisations. D'autre part, une architecture multicœur avec un cache partagé peut améliorer les performances grâce à une communication rapide entre les cœurs si les tâches concurrents travaillent sur des données proches en mémoire. Dans ce rapport, on cherche à trouver une parallélisation des applications limitées par les accès mémoire qui peut tirer partie de la présence d'un cache partagé. L'étude se focalise sur les applications séquentielles qui itèrent sur une suite de références en mémoire. Notre solution se base sur un algorithme parallèle et adaptatif avec une fenêtre glissante qui forcent les cœurs partageant le même cache à travailler sur des données proches en mémoire. Cet algorithme parallèle cause le même nombre de défauts de cache que l'algorithme séquentiel aux dépens d'une plus grand nombre de synchronisations. On analyse théoriquement le surcout due aux synchronisations à la fois pour une répartition statique et dynamique du travail. Des expériences avec une application d'extraction d'isosurfaces confirment que la collaboration des cœurs pour l'utilisation du cache partagée améliore significativement les peformances malgré le coût des synchronisations. Sur des processeurs quadri-cœurs nehalem, nos algorithms sont de 10% à 30% plus rapide que des algorithms non optimisés pour l'utilisation du cache partagé grâce à une réduction du nombre de défauts de cache.

Mots-clés : vol de travail; algorithmes efficaces en cache

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.

Most last generation multicores share a similar design for the cache hierarchy. Each core has its own private caches while the last cache level is shared between all cores. 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. Coming GPU architectures also adopt this cache design. The L_2 cache of the Intel Larrabee and the L_1 cache of NVIDIA Fermi processors are shared.

Compared to private caches, this shared cache architecture can bring performance benefits if managed adequately since it allows fast communication between cores. If some cores work on the same data, these data are not duplicated into several caches. A core can potentially use more than its fraction of the cache if necessary. But this requires to adapt the algorithms to make the cores collaborate on cache usage. Classical parallelization approaches usually favor tasks working on independent data sets to reduce communication and synchronization overhead. This results in competition rather than collaboration between cores for shared cache usage. Performance, at most equivalent to a private cache configuration, is actually impaired as the LRU replacement policy performs poorly in this context [5].

In this paper, we focus on one specific aspect of the parallelization of memory bounded applications: how to adapt the algorithm to take advantage of the shared caches of multicore processors. The goal is to propose an algorithm that improves performance by saving 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 and spatial locality) 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 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.

Figure 1: Cache Hierarchy of two multicore processors. On the left, the AMD Opteron 875 @ 2.2Ghz has only private caches whereas on the right the four cores of the Intel Xeon Nehalem E5540 @ 2.4Ghz share a L_3 cache.

This paper focuses on the algorithms that takes 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 running on multicores.

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 implements those new algorithms with the parallel library TBB and the GNU parallel STL and compare them with our implementations on the `for_each` function. Experiments on an isosurface extraction algorithm confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs.

The paper is organized as follow. In section 2, we present the cache constraint and the associated algorithms. In section 3, we detail the implementation of these two algorithms using the work-stealing based framework Kaapi. Section 4 analyzes the overhead due to the increased number of synchronizations. Finally, we introduce the isosurface extraction application we use to benchmark our algorithms in section 5 and the experimental data in section 6 before the conclusions.

2 Scheduling for Efficient Shared Cache Usage

2.1 Review of Work-Stealing and Parallel Depth First Schedules

Work Stealing (WS) is a scheduling algorithm that is very efficient both in theory and in practice. It has been implemented in many languages and parallel libraries including Cilk [3] and TBB [7]. In WS, each processor manages its own list of tasks. When a processor becomes idle, it becomes a thief, randomly chooses another processor, the victim, and try to steal some work. For an efficient load balancing, the thief should choose a task that represents a big amount of work far in memory from the work of the victim. This reduces the number of steal operations and thus synchronization costs. Moreover, as the stolen work is far from the victim work, private caches performance is not impaired. Unfortunately, stealing such tasks may not be optimal if one takes into account the shared cache of last generation multicores.

Contrary to WS, the Parallel Depth First (PDF) schedule of [2] tries to optimize shared cache usage. This schedule is based on the sequential order of execution, which is supposed to be cache-efficient. When several tasks are available, a processor will preferably execute the earliest task in the sequential order. The authors showed that a PDF schedule induces no more cache misses than the sequential execution when the parallel execution uses a slightly bigger cache. However, computing and maintaining such a schedule is difficult in practice. Moreover, as processors work on very close data in memory, bad private cache behavior could arise due to false sharing or conflict misses.

Informally, one could think of the PDF scheduler as a WS scheduler where the thieves would choose the closest task in the victim list inducing lots of steal operations. This is not as simple as all processors should work on data close in memory. In addition to the steal close operation, another mechanism is needed to prevent processors to deviate from each other after the steal operation. The cache constraint we present in the next section serves exactly this purpose. The processing order we proposed is a trade-off between WS and PDF. Processors work on data just close enough in memory to fit in the shared cache. This way the parallel application should not make more cache misses than the sequential application. Moreover, as processors still work on data far in memory compared to the private cache sizes, private cache performance is not impaired. The number of synchronizations is better than PDF but not as good as WS. Although, as the number of cache misses is reduced, the overall performance should be improved over WS.

2.2 Cache Constraint for Sequence Processing Algorithms

In this section, we explain the cache constraint that parallel algorithms should respect to induce no more cache misses than the sequential algorithm.

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. Any improvement in the performances of the sequential algorithm will be reflected in the parallel algorithm. We focus on the case where all elements of the sequence can be processed in parallel.

The parallel algorithm is based on the sequential execution order. Informally let i_{m+1} be the first element whose processing would need to evict from the shared cache data needed to process element i_1. To keep the cache performance of the sequential algorithm, the parallel scheme will deviate from the sequential order at most for m elements. That is, element i_k can be processed only when elements i_1 to i_{k-m} have been completed. This way, data evicted when processing i_k do not affect the processing of the other elements. Moreover, as the cores work on elements close in the sequential order, they work on close data and thus can benefit of other cores cache misses. In practice, m is chosen so that data managed for m sequential input elements fit in the cache. We refer to this parallel order of processing as the cache constraint in the sequel.

2.3 Window Algorithms Respecting the Cache Constraint

We introduce two parallel algorithms to process such a sequence in parallel. 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.

When theoretically and experimentally studying these two algorithms, they 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 windows algorithms, but it requires fewer synchronizations.

2.4 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 few overhead.

For the *no-window* algorithm, the sequence is statically divided into p groups. Each group is assigned to one thread binded 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 would require a very complex code. We present in the next section a work-stealing framework allowing to easily implement all these algorithms.

3 Adaptive Window Algorithms with Kaapi

In this section, we present the low level API of Kaapi [4] 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. 2). 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 preempts its thieves with the `kaapi_preempt_next_thief` call. For each thief, the victim

```
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
```

Figure 2: C implementation of the adaptive *no-window* algorithm using the KAAPI API.

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 [11]. 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 Adaptive 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

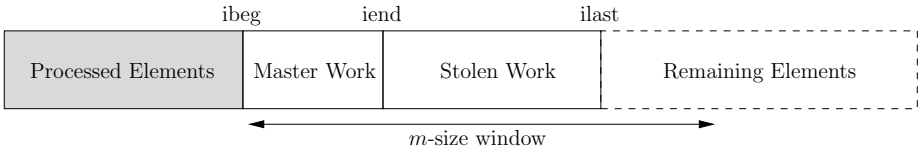

Figure 3: Decomposition of the input sequence in the *sliding-window* algorithm.

`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. 2 shows the main points of the actual implementation.

3.3 Adaptive 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 in the input sequence. This algorithm respects the cache constraint of section 2.2.

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. 3). 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]`. We implemented several variants of the `splitter`. The `local_splitter` gives in priority work in the interval `[ibeg,iend]`. It favors processing elements at the beginning to fast-forward the window thus enabling new elements to be processed. The `distant_splitter` gives in priority work in the interval `[ilast,ibeg+m]`. By distributing work at the end of the window, it should reduce the number of preemptions. The last one, `balanced_splitter` try to give well-balanced amount of work to all thieves by dividing the union of both intervals into equal size pieces. No piece of work can contains elements on both sides of the window as the resulting work would not be an interval. Thus pieces have only roughly the same size.

4 Synchronization Overhead of Window Algorithms

In this section, we theoretically analyze the number of synchronizations needed for both the PThread and Kaapi implementations of the window algorithms. We model the sequence to be processed as n independent tasks with processing times $q_1, \ldots, q_n$. The total workload W is the sum of all processing times. We denote by $q_{\min}$ and $q_{\max}$ the minimum and maximum processing times. This model is called $P||C_{\max}$ in the scheduling literature.

4.1 Number of Synchronizations of the PThread Implementation

The PThread parallelization of the *no-window* algorithm has only one global synchronization at the end of the computation when the main thread waits for the others threads to finish their work. As tasks are distributed without taking into account their processing time, one thread could end up with all the heaviest tasks. The parallel time could be at worst $q_{\max} \cdot n/p + \tau_{\text{sync}}$ where τ_{sync} is the time needed for a global synchronization. The *static-window* algorithm has n/m global synchronizations, one per window. This is n/m times more than the *no-window* algorithm. However, the parallel time of the *static-window*, $q_{\max} \cdot n/p + n/m \cdot \tau_{\text{sync}}$, is comparable with the *no-window* algorithm if τ_{sync} is small.

4.2 Number of Steal Operations of the Kaapi Implementation

A recent report [10] showed that using work stealing to schedule independent tasks (*no-window* algorithm) induces $O(p \cdot q_{\max}/q_{\min} \cdot \log_2 W)$ steal operations for a total parallel time of $W/p + O(q_{\max}/q_{\min} \cdot \log_2 W)$.

Parallelizing a chunk of m tasks with total processing time w_i with work-stealing induces $O(p \cdot q_{\max}/q_{\min} \cdot \log_2 w_i)$ steal operations. Summing on all n/m chunks, we have a total of $O(n/m \cdot p \cdot q_{\max}/q_{\min} \cdot \log_2 W/m)$ steal operations for a parallel time of $W/p + O(n/m \cdot q_{\max}/q_{\min} \cdot \log_2 W/m)$. The *static-window* has roughly n/m times more steal operations than the *no-window* algorithm. However, as the *static-window* is closer to the sequential order, it induces less cache misses and thus the processing times of the tasks q_i should be decreased. Thus it is difficult to compare these two algorithms theoretically.

As the *sliding-window* exposes a little more parallelism than the *static-window* algorithm, the number of steal operations should be slightly decreased. Indeed, the depth (or critical path T_∞) of both window algorithms is $\lceil n/m \rceil - 1$ and the number of steal operations is proportional to the depth of the computation [1].

5 Marching Tetrahedra for Isosurface Extraction

Isosurface extraction is one on the most classical filters of scientific visualization. It provides a way to understand the structure of a scalar field in a three dimensional mesh by visualizing surfaces of same scalar value. The marching tetrahedrons (MT) is an efficient algorithm for isosurface extraction [8]. For one

cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a linear approximation of the isosurface going through this cell. Applied on all mesh cells sequentially, it leads to a cost linear in the number of cells.

We now look at cache misses induced by MT. The mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). Points are accessed by following a reference from the cell array, e.g. reading coordinates of a point. As cells close in the cell array often use common points or points with close indices, processing cells in the same order as the sequential algorithm induces fewer cache misses when accessing the point array due to an improved temporal locality.

When implementing the window algorithms, the window size m should be chosen such that a sub-part of m cells of the mesh fits in the shared cache. Each point is coded on four doubles and each tetrahedron with four references (64bit integers) to points. On average, meshes have six times more tetrahedrons than points. So, for an 8MB cache, we approximately have $m = 225,000$. The same reasoning could apply to other mesh processing applications.

6 Experiments

We present experiments using the MT algorithm for isosurface extraction. 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.

6.1 Calibrating the Window Algorithms

Fig. 4(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 between 2^{15} and 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 small window sizes, cache performance is poor due to bad private cache performance. For the remaining experiments, we set $m = 2^{19}$.

Fig. 4(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.

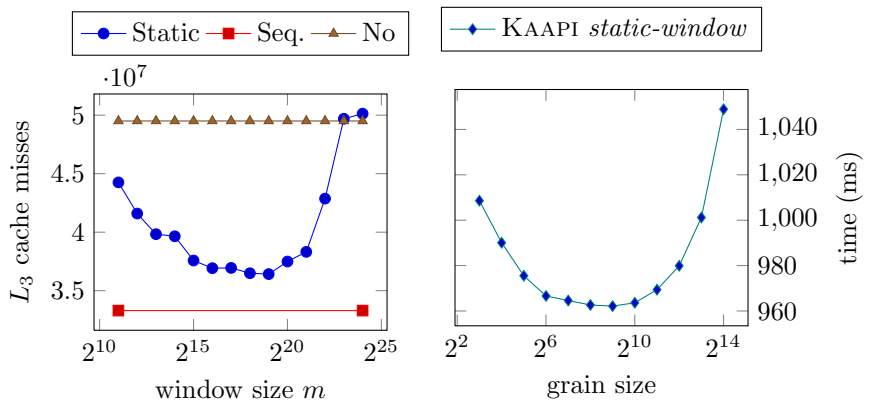

Figure 4: (Left) Number of L_3 cache misses for the PThread implementation of the *static-window* algorithm —●— for various window sizes compared to the sequential algorithm —■— and the *no-window* —▲— algorithm. (Right) Parallel time for the KAAPI implementation of the *static-window* algorithm —♦— with various grain sizes. (Both) All parallel algorithms uses the 4 cores of the Nehalem processor.

6.2 Comparison of Parallel Libraries on `for_each`

Table 1 compares KAAPI with the GNU parallel library (GNU) and Intel TBB 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.

6.3 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. 5(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 10% 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* (with the best splitter: `balanced_splitter`) shows the best performance.

Time (ms)		Nehalem			
Algorithms	#Cores	STL	GNU	TBB	KAAPI
no-window	1	3,987	4,095	3,975	4,013
	4		1,158	1,106	1,069
static-window	1	3,990	4,098	3,981	4,016
	4		1,033	966	937

Time (ms)		Opteron			
Algorithms	#Cores	STL	GNU	TBB	KAAPI
no-window	1	9,352	9,154	10,514	9,400
	4		2,514	2,680	2,431
static-window	1	9,353	9,208	10,271	9,411
	4		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.

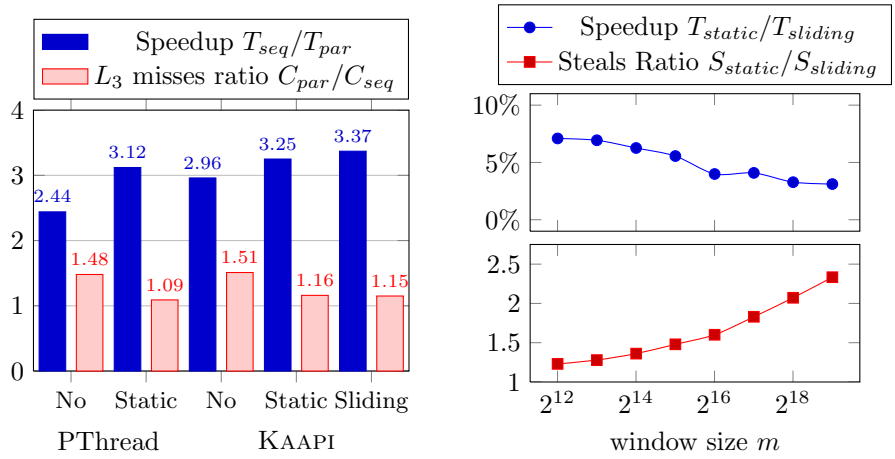

Figure 5: (Left) Speedup ■ and ratio of increased cache misses ▭ over the sequential algorithm for the *no-window*, *static-window* and *sliding-window* algorithms with PThread and KAAPI implementations. (Right) Speedup —●— and ratio of saved steal operations —■— for the *sliding-window* algorithm over the *static-window* algorithm with the KAAPI implementation. (Both) All algorithms runs on the 4 cores of the Nehalem processor.

Fig. 5(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 for bigger windows) leading to an additional gain around 5%.

7 Related works

Previous experimental approaches have shown the interest of efficient cache sharing usage, on a recent benchmark in [12] and on data mining applications in [6]. In this paper, we go beyond those specific approaches by providing general algorithms for independent tasks parallelism which respect the sequential locality.

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [13]. However, none of these techniques take into account the number of cache misses and the shared cache of multicore processors. Optimization of sequential locality for mesh applications have been studied through mesh layout optimization in [9].

8 Conclusions and Future Work

This paper focuses on exploiting the shared cache of last generation multicores. We presented new algorithms to parallelize STL-like sequence processing. We theoretically analyzed their synchronization overhead for both static and dynamic load balancing. Experiments on several parallel libraries confirm that these techniques increase performance from 10% to 30% thanks to a reduced number of last level cache misses.

Future work includes the extension of the window algorithms to general tasks with dependencies for scientific computing applications. We also plan on automatically tuning the different parameters used in this paper.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theor. Comp. Sys.*, 34(2):115–144, 2001.

[2] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[4] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In Stephen Watt, editor, *PASCO 2007*, pages 15–23. Waterloo University, Ontario, Canada, July 2007.

[5] A. Hassidim. Cache replacement policies for multicore processors. In *ICS*, 2010.

[6] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. In *HPCA*.

[7] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IPDPS*, 2008.

[8] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed.* Kitware Inc., 2004.

[9] M. Tchiboukdjian, V. Danjean, and B. Raffin. Binary mesh partitioning for cache-efficient visualization. *Transactions on Visualization and Computer Graphics*, (PrePrints), 2010.

[10] M. Tchiboukdjian, D. Trystram, J.-L. Roch, and J. Bernard. List scheduling: The price of distribution. Technical report, INRIA, 2010.

[11] D. Traoré, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel stl algorithms. In *Euro-Par*, 2008.

[12] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP*, 2010.

[13] H. Zhang, T. S. Newman, and X. Zhang. Case study of multithreaded in-core isosurface extraction algorithms. In *EGPGV*, 2004.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399