LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms

François Broquedis[1], Thierry Gautier[2], and Vincent Danjean[3]

[1] INPG
[2] INRIA
[3] UJF,
MOAIS Team, LIG, Grenoble, France
{francois.broquedis,vincent.danjean}@imag.fr,
thierry.gautier@inrialpes.fr

Abstract. To efficiently exploit high performance computing platforms, applications currently have to express more and more finer-grain parallelism. The OpenMP standard allows programmers to do so since version 3.0 and the introduction of task parallelism. Even if this evolution stands as a necessary step towards scalability over shared memory machines holding hundreds of cores, the current specification of OpenMP lacks ways of expressing dependencies between tasks, forcing programmers to make unnecessary use of synchronization degrading overall performance. This paper introduces LIBKOMP, an OpenMP runtime system based on the X-KAAPI library that outperforms popular OpenMP implementations on current task-based OpenMP benchmarks, but also provides OpenMP programmers with new ways of expressing data-flow parallelism.

Keywords: OpenMP, data-flow programming, task parallelism, runtime systems.

1 Introduction

The architecture design of high performance computing platforms keeps getting more and more complex, widening the gap between the theoretical computing power of a given architecture and the performance parallel applications can achieve on it. HPC programmers have to express massive parallelism to occupy the constantly growing number of processing units contained in a multicore chip, and finely control the way parallel flows are executed to efficiently deal with memory affinity (shared cache memory, NUMA design, etc.). This burden will not get any lighter with the recent evolution of processor design, in which architects associate a few powerful cores with numerous, more simple cores. The success of this kind of design will rely on the ability for programmers to write applications with good performance at runtime, even for small problem instances.

Several libraries and programming environments [30,15,29,9,8] were proposed to improve the productivity of programmers by encouraging them to express all the potential parallelism in an application at fine grain, while delegating to the runtime system (or the compiler) the role to extract useful parallelism for the target multicore machine. They introduce high-level parallel constructs, such as Cilk `cilk_for`, X10

B.M. Chapman et al. (Eds.): IWOMP 2012, LNCS 7312, pp. 102–115, 2012.

`for_each` or OpenMP `parallel for`, to easily describe potential parallelism in most of HPC numerical applications. In Cilk++, Intel Cilk+, Intel TBB, X-KAAPI and OpenMP (using the dynamic loop scheduler), the parallel loops generate internal tasks and rely on variations of a work stealing algorithm to deal with load balancing.

Tasks are now part of the OpenMP standard since version 3.0. To schedule task-based parallel applications, the work-stealing algorithm [6,3,16,20,26] is one of the most heavily-studied dynamic scheduler. Its biggest advantage lies in its simple predictive performance model. Several studies on OpenMP task scheduling have shown that work-stealing based algorithms seem to provide, on average, good speedup [11,27,28,1,23].

While we consider this evolution as a necessary step to exploit *manycore* computers efficiently, several studies have illustrated the limitation of the OpenMP *fork-join* task execution model [5,12,22] with respect to the data flow model, emphasizing that data flow applications are able to express more parallelism. The OpenMP ARB is already considering interesting possible extensions to the standard to deal with task/data dependencies [4,12], allowing OpenMP programmers to exploit accelerators in a unified model.

The X-KAAPI[1] library, a re-design in C of the Kaapi library [17] we develop, provides a runtime system that has proven to be efficient when it comes to scheduling data-flow parallel applications over multicore machines. X-KAAPI comes with very little task creation and scheduling overheads and implements recursive tasks in a very efficient way, making this runtime system a good candidate for both scheduling OpenMP 3.0 tasks, as the fork-join model can be seen as a particular case of the data-flow paradigm, and experiment with possible data-flow related extensions to the OpenMP standard.

This paper introduces LIBKOMP, an OpenMP runtime system based on the X-KAAPI library that performs well on current task-based OpenMP benchmarks and applications, but also emphasizes the interest of extending the OpenMP standard to express data-flow parallelism, presenting performance improvements for OpenMP benchmarks that were modified to express task/data dependencies.

The paper is organized as follows. Section 2 introduces the main assets of the X-KAAPI runtime system that the LIBKOMP library can rely on, and details the way we implement the OpenMP tasking model. Section 3 describes the evaluation of our runtime on the original BOTS benchmarks suite and modified versions of the SparseLU and NQueens kernels to benefit from the data-flow execution model while Section 4 presents some related work.

2 The LIBKOMP Runtime System

OpenMP tasks offer the application programmer new ways of expressing parallelism. This new paradigm will make OpenMP applications generate a great number of fine-grain tasks. The success of such an approach for parallelizing applications will greatly depend on the runtime system's ability to:

1. *Generate all these tasks with the smaller overhead possible:* the long term goal would be to let the runtime system decide how many tasks a parallel region should create considering both the application and the system current states.

[1] `http://kaapi.gforge.inria.fr`

2. *Provide efficient ways of performing load balancing to reach scalability:* a task-based application can dynamically generate tasks of different types and workloads.
3. *Implement recursive tasks in an efficient way:* recursive algorithms should be parallelized using recursive tasks, as it's most of the time the most convenient way to parallelize them, and not being penalized in terms of performance.

On top of focusing on these three aspects while implementing the OpenMP 3.0 libGOMP ABI, our LIBKOMP runtime system also provides the OpenMP programmer with new ways of expressing dependencies between OpenMP tasks, thanks to specific keywords provided by a source-to-source compiler we also develop, called *KaCC* [25]. So, LIBKOMP can be used either as a run-time replacement of the libGOMP runtime for OpenMP binaries compiled with GCC, or it can be used as a classical shared library for applications compiled with *KaCC* (allowing use of its extended features).

2.1 The LIBKOMP Execution Model

In LIBKOMP, each OpenMP thread corresponds to a X-KAAPI task. The number of kernel threads used to run an OpenMP program is controlled by the internal control variable called *nthreads-var* [29]. When the application reaches a scheduling point, a kernel thread is able to suspend the current task to execute another one, and resume execution of the previous task later. LIBKOMP takes advantage of such context switches to restore previous internal control variables (ICV), OpenMP thread number, *etc.*, if required. When the execution starts, the master thread of the current process starts to execute the main task. A thread creates tasks and pushes them on its own workqueue. The workqueue is represented as a stack. The enqueue operation is very fast, typically about ten cycles on current processors. As for Cilk, a running X-KAAPI task can create child tasks. Once a task terminates its execution, the thread that was executing it picks its children first, following the FIFO order of their creation.

2.2 Parallel Regions in LIBKOMP

A parallel region creates a set of implicit initial tasks, each of them being associated with a unique OpenMP thread number, which share team-related information. Tasks are pushed into the X-KAAPI stack of the running thread in a new activation frame. Tasks are not bound to kernel threads: it is the responsibility of the X-KAAPI work stealing scheduler to dynamically decide the mapping. LIBKOMP interprets a program specification of a number of threads `num_threads` in a *parallel* directive as the creation of `num_threads` X-KAAPI tasks. Several of these tasks may be scheduled on the same kernel thread, depending on the threads workload and the scheduling decisions taken by the X-KAAPI work-stealing scheduler. At the end of a parallel region, its master thread calls a LIBKOMP function to wait for the completion of all previously created tasks in the activation frame associated with this region.

2.3 Data Access Modes for Dependent Tasks

A X-KAAPI task is a function call that should return no value except through the shared memory and the list of its effective parameters. Tasks share data if they have access to

```
1  for (k = 0; k < NB; ++k)
2  {
3  #pragma kaapi task readwrite (sli[k,k])
4    potrf (BS, sli[k,k]);
5
6    for (m = k+1; m < NB; ++m)
7    {
8      if (is_empty (sli[m,k])) continue;
9  #pragma kaapi task read (sli[k,k]) readwrite (sli[m,k])
10     trsm (BS, sli[k,k], sli[m,k]);
11   }
12
13   for (m = k+1; m < NB; ++m)
14   {
15     if (is_empty (m, k, &sli)) continue;
16 #pragma kaapi task read (sli[m,k]) readwrite (sli[m,m])
17     syrk (BS, sli[m,k], sli[m,m]);
18
19     for (n = k+1; n < m; ++n)
20     {
21       if ((is_empty (n, k, &sli) || (is_empty (m, n, &sli))) continue;
22 #pragma kaapi task read (sli[n,k], sli[m,k]) readwrite (sli[m,n])
23       gemm (BS, sli[n,k], sli[m,k], sli[m,n]);
24     }
25   }
26 }
27 #pragma kaapi sync
```

Fig. 1. Pseudo code for sparse Cholesky factorization

the same memory region. A memory region is defined as a set of addresses in the process virtual address space. With X-KAAPI, this set has the shape of a multi-dimensional array [25].

The user is responsible for indicating the mode each task uses to access the memory: the main access modes are *read*, *write*, *cumulative write* or *exclusive* [16,17,25,24]. The syntax to specify these access modes is very close to the directives proposed in StarSs meta model [5] and those defined by the OpenMP dependent tasks proposal [12].

Code of figure 1 illustrates the API provided by X-KAAPI along with the KaCC compiler [25] on a sparse Cholesky factorization used in the performance evaluation section. The matrix is composed of at most $NB \times NB$ blocks of size $BS \times BS$. The clauses *read* or *readwrite* specify access mode for variables following the structured block. True dependencies exist when a task read data produced by a previously created task. For instance, the task created at line 3 produces (read-write access) the diagonal block $[k, k]$ that will be consumed by tasks created at line 9. A variable that does not appear in any clause is passed by value.

OpenMP task model does not allow dependent tasks. The application programmer has to insert coarse grain synchronizations using the `taskwait` keyword to respect data flow dependencies, which can limit parallelism [22].

Tasks with data flow dependencies have already been cited to be important in linear algebra [22] or for managing multi-CPUs multi-GPUs computations [4,12,2,21] in a unified model. It could be used to avoid unnecessary synchronizations in recursive divide and conquer programs, such as the BOTS NQueens. Indeed, the number of solutions cumulated by each task only requires one final synchronization. Due to the limitations of the OpenMP tasking model, the BOTS NQueens implementation waits for the

completion of all created child tasks at each level of the recursion. This synchronization allows to cumulate subresults but it also permits fast C stack allocation of chess board state for each child. Thanks to the cumulative access mode and the stack-based task management proposed by X-KAAPI, it is also possible to avoid these synchronization points by allocating a chess board state in the internal X-KAAPI stack, such that it is valid when a spawned task performs its computation [17].

A X-KAAPI task is a very light object. It basically holds a pointer to the main entry point function, its parameters and some flags set by the task scheduler. For each type of task, the runtime maintains a *format* object which is responsible to interpret the task: retrieve the access mode and type of each parameters, getting the implementation of the entry point of the task (CPU or GPU [21]). Such separation reduces the task size by factorizing common information.

2.4 Stack-Based Execution

At runtime, a X-KAAPI task generates a sequence of child tasks that access data in a shared memory area. Each task is pushed into the queue of the current thread. After a task finishes, its children tasks are executed with respect to the order of their creation. The local queue is managed as LIFO queue of activation frames. Each activation frame is a FIFO queue of tasks. This model implements a valid, highly efficient sequential execution order [16,17], as the runtime system only needs to compute data flow dependencies when the thread execution scheme reaches a task that has been stolen and not completed yet. The successors of the stolen task depend on its completion. So, all tasks following the first stolen task encountered, must require computation of data flow dependencies to detect whether they are ready or not. In order to keep fast stack-based execution without computation of data flow dependencies in X-KAAPI [17], a thread suspends its execution when it reaches the first stolen task in its stack and calls the work stealing scheduler to steal a new ready task.

2.5 Work Stealing and Data Flow Dependencies

Thanks to Cilk [6,15], the work stealing technique has become mainstream and is now often considered when it comes to dynamically balance the work load among processing units. The work stealing principle can be synthesized as follows. An idle thread, called a thief, initiates a steal request to a random selected victim. On reply, the thief receives one or more ready tasks.

At the startup time, only the main thread of X-KAAPI process performs tasks, all others threads are idle. This original idea in X-KAAPI follows the *work first principle* [15]; at the expense of a larger critical path, X-KAAPI moves the cost of computing ready tasks from the work performed by the victim during task's creations to the steal operations performed by thieves. Theoretical analysis of work stealing algorithms to schedule dependent tasks are studied in [16,18] and an elegant recent proof is written in [32] which considered specifics of the X-KAAPI work stealing protocol.

To compute a ready task, a thief thread iterates through the victim's queue from the last recent pushed task to the most recent one and it computes true data flow

dependencies for each task. False dependencies are resolved through variables renaming. The iteration stops on the first task found ready.

The main difference between X-KAAPI and other software [5,2,34] is that X-KAAPI computes data flow dependencies only when idle thread search for a ready task.

2.6 Discussion

If a program is highly parallel, i.e. $T_\infty \ll T_1$, then the number of steal operations per thread remains in order $O(T_\infty)$ which is low. In that case, the cost of computing data flow, perhaps multiple times if several idle threads iterate over the same queue, is negligible with comparison to systematic computation on task creation. Otherwise, if the frequency of steal operations increases, X-KAAPI tries to aggregate multiple requests to the same victim. Our protocol elects the thieves to reply to all of the victim's requests. This aggregation strategy permits the combination of k searches of ready tasks in a less costly operation to one search of k ready tasks [19]. In [32], a theoretical analysis shows it can reduce the total number of steal requests.

Nevertheless, the overhead to manage tasks and computing data flow graph could remain important. Also, X-KAAPI implements an original optimization. It is applied when the cost of computing ready tasks becomes important, especially when the victim's stack contains many tasks. The user may annotate code or the scheduler automatically detects such situation. Then, the scheduler computes, and attaches to the stack, an accelerating data structure to make faster steal operations. The structure maintains the list of ready tasks. When a task completes and activates dependent tasks, the runtime pushes them directly into list (of ready tasks). The capacity of X-KAAPI to pass from workqueue' stack representation to this accelerating data structure makes it unique. It allows to move overhead in computing ready tasks during steal operation to the computation of accelerating data structure with low cost steal operation.

2.7 Parallel Loops in LIBKOMP

The parallel loop support in GCC/OpenMP relies on three main functions to initialize the iteration space, get the next slice for local computation and a function call at the end of the loop. Static scheduling may inline some of them. LIBKOMP follows the same ABI and relies on the X-KAAPI loop support [24]. Loop support in X-KAAPI is based on adaptive algorithms [33,31] to dynamically adapt the parallelism grain (number of tasks, number of iterations per task, etc.) considering the current system state. The LIBKOMP loop port on X-KAAPI is only required to decompose the original X-KAAPI parallel loop in order to fit the parallel work share construct of the libGOMP ABI.

3 Performance Evaluation

This section presents our evaluation on the BOTS benchmarks suite and two versions of the Cholesky factorization to compare the performance obtained by our solution with respect to two other OpenMP implementations: the original libGOMP that comes with

version 4.6.2 of the GCC compiler and version 12.1.2 of the Intel C OpenMP compiler. LIBKOMP is based on version 1.0.2 of the X-KAAPI runtime system.

We conducted our experiments on CC-NUMA 48 cores AMD Magny Cours. There are three levels of cache memory. L1 (64 KB) and L2 (512 KB) are per core, whereas L3 (5 MB) is shared by 6 cores. This configuration provides a total of 256 GB (32 GB per NUMA node) of main memory. We will refer to this configuration as **AMD48** in the following of the paper.

3.1 Task Management Overhead

This section compares the overhead of task creation and execution with respect to the sequential computation. The experiment evaluates the time to execute the KaCC/LIBKOMP program of figure 2 for computing the 35-th Fibonacci number using the fast task creation protocol. Equivalent programs in term of task creations and synchronizations are written in Intel Cilk+, Intel TBB 4.0 and GCC/libGOMP. Sequential time is $0.091s$. Figure 2 reports times using 1, 8, 16, 32 and 48 cores from our *AMD48* configuration. On 1 core, LIBKOMP has the smallest overhead with respect to the sequential computation (slowdown of about 8). This overhead can easily be absorbed by increasing the task granularity, but at the expense of increasing the critical path, thus reducing the available parallelism [15,11]. The grain is too fine for OpenMP/libGOMP (computation was stopped on 32 and 48 cores after 5 minutes). For one core execution, libGOMP never creates tasks and makes function calls as sequential execution does.

```
void fibonacci(long* result,
    const long n)
{
  if (n<2)
    *result = n;
  else
  {
    long r1,r2;
#pragma kaapi task write(&r1)
    fibonacci( &r1, n-1 );
    fibonacci( &r2, n-2 );
#pragma kaapi sync
    *result = r1 + r2;
  }
}
```

(a) LIBKOMP benchmark using KaCC

#cores	Cilk+	TBB	LIBKOMP	libGOMP
1	1.063	2.356	0.728	2.429
(slowdown)	(x 11.7)	(x 26)	(x 8)	(x27)
8	0.127	0.293	0.094	*51.06*
16	0.065	0.146	0.047	*104.14*
32	0.035	0.072	0.024	*(no time)*
48	0.028	0.049	0.017	*(no time)*

(b) Time (second) on the AMD48 configuration for `fibonacci(35)`. Sequential time is 0.091 s.

Fig. 2. Fibonacci micro benchmark

3.2 Parallel Loops

In this section, we compare the performance obtained by both the libGOMP runtime system and LIBKOMP on a parallel version of EUROPLEXUS [14], an industrial application that computes finite element simulation of fluid-structure systems, exposing a single OpenMP parallel loop. Because work per iteration is lightly irregular, we tested both static and dynamic scheduling for libGOMP. We use the MAXPLANE

instance as input of the EUROPLEXUS application. Figure 3 reports the obtained speedup of parallel implementations with respect to the sequential version. The same cores was used in both libGOMP or LIBKOMP using the environment variable `GOMP_CPU_AFFINITY`. Overall speedups are very close, but LIBKOMP scales better for a larger number of cores (>25).

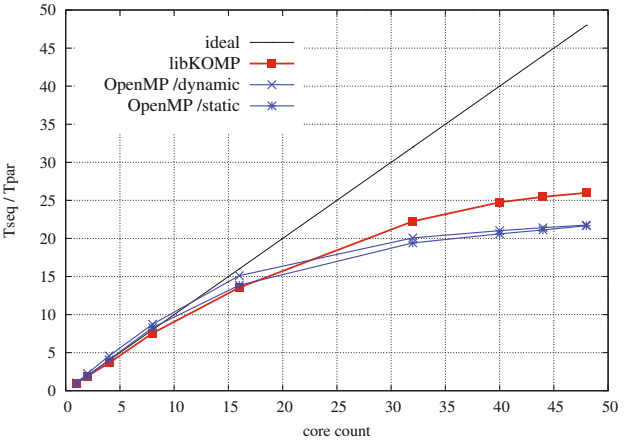


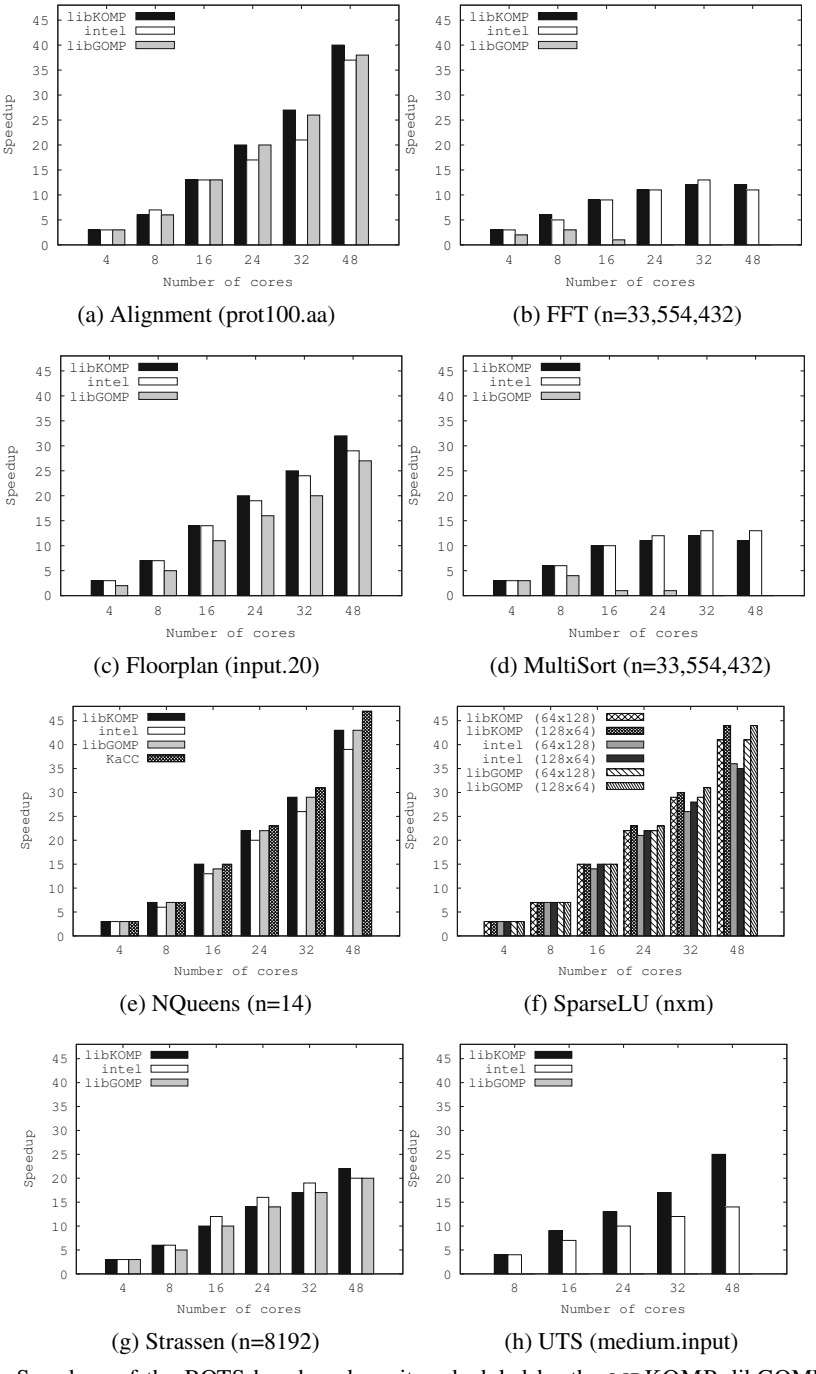
Fig. 3. EUROPLEXUS parallel loop speedup, libGOMP (static, dynamic) VS LIBKOMP (xkaapi)

3.3 Barcelona OpenMP Tasks Suite (BOTS)

The Barcelona OpenMP Tasks Suite has been introduced to test the behavior of 3.0-compatible OpenMP runtime systems regarding tasks implementation. It provides several kernels inspired from real-life OpenMP applications and projects. Each kernel, detailed in [13], comes with different implementations relying on different aspects/keywords of the OpenMP 3.0 tasks model (tied/untied tasks, controlling the cut-off using the if clause, etc.). We ran all these kernels on the AMD48 platform using a varying number of cores to experiment with each kernel's scalability, and kept the best implementation for each runtime system. Figure 4 shows the corresponding results.

Executing some of these kernels may lead to the creation of a great number of tasks. For instance, the execution of the NQueens algorithm on a 14x14 chessboard generates more than 370M of tasks. Creating such a number of tasks comes with overheads on any tested runtime systems, the worst ones being observed from libGOMP. Determining the *right* number of tasks to instanciate from an application may be really challenging. Some runtime systems like libGOMP implements a threshold heuristic that limits tasks creation when the number of tasks is greater than k times the number of threads. It has the advantage of limiting the number of tasks but may limit the parallelism of the application, as observed on the FFT benchmark performance on figure 4b in which creating all the 2M tasks expressed in the application allows both the LIBKOMP and Intel runtime systems to perform better load balancing. Some of these embarassing applications comes with implementations in which the application programmer can define the maximum depth from which new tasks will be executed sequentially, taking the number of creating tasks from 370M for NQueens to 2394 for example, thus explaining the better performance obtained by libGOMP on the these kernels.

More generally, these experiments show LIBKOMP obtains performance that is comparable to other OpenMP runtime systems (sometimes being even better!) on fork-join applications exposing a reasonable number of tasks, and outperforms libGOMP



(a) Alignment (prot100.aa)

(b) FFT (n=33,554,432)

(c) Floorplan (input.20)

(d) MultiSort (n=33,554,432)

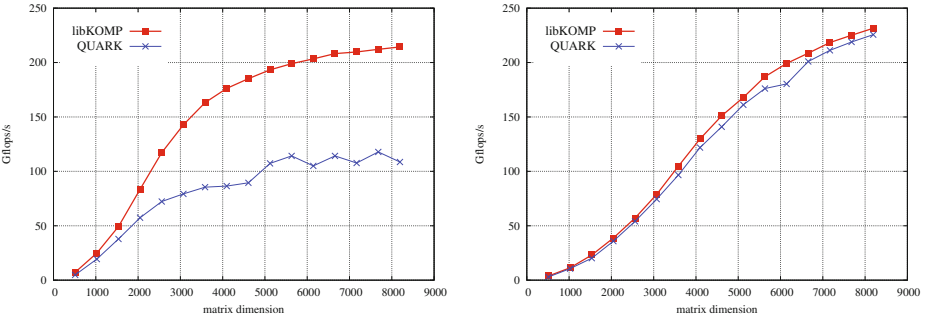
(e) NQueens (n=14)

(f) SparseLU (nxm)

(g) Strassen (n=8192)

(h) UTS (medium.input)

Fig. 4. Speedups of the BOTS benchmarks suite scheduled by the LIBKOMP, libGOMP and Intel runtime systems on a varying number of cores from the AMD48 platform with respect to the GCC-compiled sequential version



(a) Tile size of $NB = 128$ (b) Tile size of $NB = 256$

Fig. 5. Gflops on Cholesky algorithm with QUARK and LIBKOMP

and Intel on applications creating a great number of tasks, thanks to its efficient management of recursive tasks.

To conclude this section, reported experiments have demonstrated that LIBKOMP has almost the same performance as libGOMP or Intel ICC for a moderate number of tasks and a moderate number of cores. For tasks-intensive computations, such as UTS, LIBKOMP outperforms the other two OpenMP implementations. LIBKOMP is designed to schedule data-flow graphs.

Mixing tasks with declaration of memory access modes allows a finer resolution of synchronizations. It also provides valuable information on memory accesses to the runtime system. The implementation of BOTS NQueens has been modified as described at the end of section 2.4 and compiled with KaCC/LIBKOMP. Letting the runtime system deal with fine-grain sychronizations allows to significantly improve the overall performance here, as this version of NQueens reaches a speedup of 47.8 over 48 cores of the AMD48 machine (KaCC performance reported on figure 4e).

3.4 Data Flow Tasks versus Fork-Join Tasks

We evaluate the potential gain offered by data-flow tasks over OpenMP 3.0 fork-join tasks executing two different versions of the Cholesky factorization, a widely-used linear algebra algorithm.

LIBKOMP versus QUARK. The first one relies on the `PLASMA_dpotrf_Tile` algorithm coming from version 2.4.2 of the PLASMA [7] library that comes with a runtime system, in charge of scheduling PLASMA tasks, called *QUARK*. We implemented the QUARK [34] ABI for dependent tasks on top of LIBKOMP to compare our implementation with the original version of QUARK.

Figure 5 reports the performance, in GFlop/s, for different matrix sizes on the *AMD48* machine. One can observe that LIBKOMP outperforms QUARK for fine grain tasks ($NB = 128$). The main reasons are: 1/ QUARK implements a centralized list of ready tasks; 2/ Creating QUARK tasks comes with bigger overheads. We can expect this contention point to become more severe as the number of cores increases with next generation machines, affecting PLASMA performance. When the grain increases, LIBKOMP remains better but the difference decreases because of the

relatively small impact of the task management with respect to the whole computation. One can also note that increasing the grain size reduces the average parallelism and limits the speedup. For a matrix size of 3000, the performance for $NB = 128$ reaches $150GFlops$, while for $NB = 256$, it drops to about $75GFlops$.

LIBKOMP versus OMP. The second version of the Cholesky factorization we studied here is a sparse factorization (LDL^t) coming from the industrial code EUROPLEXUS [14]. We compare a data flow program on top of LIBKOMP with respect to the original EUROPLEXUS code using OpenMP 3.0 task. The two code structures are similar to Cilk and SMPSs codes presented in [22] for the dense case. Management of sparsity is done in the same way as in the BOTS SparseLU code. Figure 6 reports speedup using a matrix used by the MAXPLANE simulation in EUROPLEXUS [14]. The dimension of the matrix is 59462 with 3.59% of non zero elements. The block size with the best sequential time is $BS = 88$ and is used for parallel versions. The sequential time is $47.79s$. LIBKOMP version outperforms libGOMP version, for the same reasons as for the dense case [22]: thanks to the knowledge of data dependencies, independent tasks between outer loop iterations can be executed concurrently.

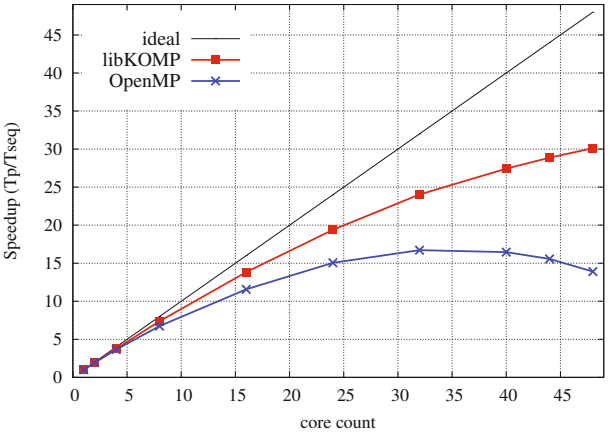


Fig. 6. Speedups on sparse Cholesky

4 Related Work

Kaapi [17] was designed in our group after our preliminary work on Athapascan [16,10]. X-KAAPI keeps definition of access modes to compute data flow dependencies between a sequence of tasks. StarSs/SMPSs [5], QUARK [34], StarPU [2] follow the same design. We can still note differences in the kind of access modes and the shape of the memory region that are defined : StarSs/SMPSs, QUARK have similar access mode but only consider unidimensional arrays. QUARK comes with an original `scratch` access mode to reuse thread-specific temporary data. StarPU [2] has a more complex way to split data and define sub-views of a data structure. X-KAAPI has a direct support for multi-dimensional arrays [25].

The data flow task model is flat in StarSs/SMPSs, QUARK and StarPU while X-KAAPI also allows recursive tasks creation. The fork-join parallel paradigm is only supported by X-KAAPI, Intel TBB [30], Cilk [6,15] and Cilk+ (Intel version of Cilk). The X-KAAPI performance for fine grain recursive applications is equivalent, and sometimes better, than Cilk+ and Intel TBB, that only allow the creation of independent tasks.

In TBB, Cilk or X-KAAPI, task creation is several order of magnitude cheaper than in StartSs/SMPSs, QUARK or StarPU. Scalability of the QUARK and StarPU runtime system is limited due to their central list scheduling. SMPSs seems to support a more distributed scheduling.

X-KAAPI has a unique model of adaptive task that allows a runtime adaptation of tasks creation when a resource turns idle. The OpenMP runtime of GCC 4.6.2, libGOMP, implements a threshold heuristic that limits tasks creation when the number of tasks is greater than k times the number of threads ($k = 64$). It has the advantage of limiting the number of tasks [11] (max-task strategy) but may limit the parallelism of the application, as observed on the FFT benchmark performance [11]. TBB, with an autopartitionner heuristic, is able to limit the number of tasks without *a priori* limiting the application parallelism.

Intel TBB, Cilk+, OpenMP and X-KAAPI support parallel loops which are not available in StarSs/SMPSs, QUARK or StarPU. Our comparison with OpenMP/GCC 4.6.2 shows that for benchmarked instances on real EUROPLEXUS code, OpenMP loop scheduling strategy is not an important feature.

From all of the tested softwares, X-KAAPI is the only runtime system that allows to mix in a unified framework data-flow tasks, fork-join tasks and parallel loops with at least equivalent performance (sometimes even better!) than specific softwares for each paradigm.

5 Conclusion

Computer architects keep designing more and more complex platforms embedding an almost constantly increasing number of processing units. To deal with these so-called manycore architectures, OpenMP had to evolve to allow the application programmer to express finer-grain parallelism. The 3.0 version of the OpenMP standard has layed the fundations of a fine-grain environment, introducing the task construct to generate fine-grain tasks, either explicitely or out of OpenMP 2.5 parallel regions. We proposed in this paper a runtime system, called LIBKOMP, that efficiently implements the OpenMP task model and is binary compatible with existing OpenMP applications built against GCC's libGOMP. LIBKOMP outperformed popular OpenMP implementations like GCC's libGOMP and ICC's KMP runtime systems on several benchmarks of the Barcelona OpenMP Tasks Suite. We also showed the interest of taking OpenMP task proposal one step further, proposing extensions to deal with data dependencies. We implemented these extensions inside a source-to-source compiler obtaining better performance using data-flow tasks compared to OpenMP 3.0 fork-join tasks. From our point of view, many of the characteristics of the LIBKOMP runtime system, and more generally the X-KAAPI runtime system LIBKOMP is based on, like adaptive loops scheduling, moldable tasks and also unified CPU/GPU programming are interesting to discuss as possible OpenMP evolutions.

Acknowledgement. The authors would like to thank Fabien Le Mentec for providing results on EUROPLEXUS code. Work on EUROPLEXUS have been partially supported by CEA and by the 09-COSI-011-05 REPDYN ANR Project. This work has been partially supported by the ANR-11-BS02-013 HPAC ANR Project.

References

1. Agathos, S.N., Hadjidoukas, P.E., Dimakopoulos, V.V.: Design and implementation of openmp tasks in the ompi compiler. In: Angelidis, P., Michalas, A. (eds.) Panhellenic Conference on Informatics, pp. 265–269. IEEE (2011), http://dblp.uni-trier.de/db/conf/pci/pci2011.html#AgathosHD11
2. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Roman, J., Thibault, S., Tomov, S.: Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In: Symposium on Application Accelerators in High Performance Computing (SAAHPC), Knoxville, USA (July 2010)
3. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theor. Comp. Sys. 34(2), 115–144 (2001)
4. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02303-3_13
5. Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using smpss. Concurr. Comput.: Pract. Exper. 21, 2438–2456 (2009)
6. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing 37(1), 55–69 (1996), citeseer.nj.nec.com/article/blumofe95cilk.html
7. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. 35, 38–53 (2009)
8. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. 21, 291–312 (2007), http://dl.acm.org/citation.cfm?id=1286120.1286123
9. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 519–538 (2005)
10. Dumitrescu, B., Doreille, M., Roch, J.L., Trystram, D.: Two-dimensional block partitionings for the parallel sparse cholesky factorization. Numerical Algorithms 16, 17–38 (1997)
11. Duran, A., Corbalán, J., Ayguadé, E.: Evaluation of OpenMP Task Scheduling Strategies. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 100–110. Springer, Heidelberg (2008)
12. Duran, A., Perez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the OpenMP Tasking Model to Allow Dependent Tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
13. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE (2009)
14. Faucher, V.: Advanced Parallel Computing for Explosive Fluid-Structure Interaction. In: COMPDYN 2011, Corfu, Greece (May 2011)
15. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, pp. 212–223. ACM, New York (1998)
16. Galilée, F., Roch, J.L., Cavalheiro, G.G.H., Doreille, M.: Athapascan-1: On-line building data flow graph in a parallel language. In: Proceedings of PACT 1998, p. 88. IEEE Computer Society, Washington, DC (1998)

17. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO 2007 (2007)
18. Gautier, T., Roch, J.L., Wagner, F.: Fine grain distributed implementation of a dataflow language with provable performances. In: Workshop PAPP 2007 - Practical Aspects of High-Level Parallel Programming in (ICCS2007). IEEE, Beijing (2007)
19. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 355–364. ACM, New York (2010)
20. Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: PODC 2002: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pp. 280–289. ACM, New York (2002)
21. Hermann, E., Raffin, B., Faure, F., Gautier, T., Allard, J.: Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 235–246. Springer, Heidelberg (2010)
22. Kurzak, J., Ltaief, H., Dongarra, J., Badia, R.M.: Scheduling dense linear algebra operations on multicore processors. Concurr. Comput.: Pract. Exper. 22, 15–44 (2010)
23. LaGrone, J., Aribuki, A., Addison, C., Chapman, B.: A Runtime Implementation of OpenMP Tasks. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 165–178. Springer, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2023025.2023042
24. Le Mentec, F., Danjean, V., Gautier, T.: X-Kaapi C programming interface. Tech. Rep. RT-0417, INRIA (December 2011)
25. Le Mentec, F., Gautier, T., Danjean, V.: The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming. Tech. Rep. RT-0418, INRIA (December 2011)
26. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent work stealing. SIGPLAN Not. 44, 45–54 (2009)
27. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2011, pp. 49–56. ACM, New York (2011), http://doi.acm.org/10.1145/1988796.1988804
28. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: Openmp task scheduling strategies for multicore numa systems. International Journal of High Performance Computing Applications (2012)
29. OpenMP Architecture Review Board (1997-2008), http://www.openmp.org
30. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: IPDPS (2008)
31. Tchiboukdjian, M., Danjean, V., Gautier, T., Le Mentec, F., Raffin, B.: A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 99–107. Springer, Heidelberg (2011)
32. Tchiboukdjian, M., Gast, N., Trystram, D., Roch, J.-L., Bernard, J.: A Tighter Analysis of Work Stealing. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 291–302. Springer, Heidelberg (2010)
33. Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-Free Work-Optimal Parallel STL Algorithms. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 887–897. Springer, Heidelberg (2008), http://www.caos.uab.es/europar2008/
34. YarKhan, A., Kurzak, J., Dongarra, J.: Quark users' guide: Queueing and runtime for kernels. Tech. Rep. ICL-UT-11-02. University of Tennessee (2011)