X-KAAPI

Gautier Thierry and Lementec Fabien

EPI MOAIS
INRIA Rhône-Alpes
Grenoble
France
thierry.gautier@inrialpes.fr, fabien.lementec@inrialpes.fr

ABSTRACT. Programming multicore architectures becomes more and more painful: architectures are complexes and programming models are numerous. X-KAAPI is an original parallel programming environment that allows to describe tasks with data flow dependencies as well as parallel loops. With X-KAAPI a programmer may write specialized code in dense linear algebra where finer data flow synchronization seems to provide better performance, or he could write classical OpenMP code using parallel loop. X-KAAPI compiler allows to annotate C or C++ code in order to automatically to translate directives to calls to the runtime support. This paper presents an overview of X-KAAPI programming model and it reports performances over three applications representative of three distinct classes. For each of this application, X-KAAPI has same or better performances as the reference programming environment.

RÉSUMÉ. La programmation des multicœurs dévient de plus en plus difficile : les architectures sont complexes, les modèles de programmation multiples. Cet article présente le modèle de programmation parallèle X-KAAPI qui permet, dans un cadre unifié, de décrire un ensemble de tâches avec des dépendances de flot de données ainsi que des boucles indépendantes. Avec X-KAAPI, le programmeur peut à la fois développer des programmes en algèbre linéaire dense où il a été montré l'intérêt à gérer des dépendances dû à la disponibilité des données, ou bien programmer les célèbres boucles parallèles à la OpenMP. X-KAAPI se base sur un compilateur source à source qui permet de traduire des annotations (`#pragma`) d'un programme séquentiel écrit en C ou C++ en des appels au support exécutif. Dans cet article nous présentons X-KAAPI et nous montrons ses performances sur trois applications de trois classes bien distinctes : un micro-benchmark pour mesure le coût à créer et exécuter un grand nombre de tâches ; un algorithme de factorisation de Cholesky en algbèbre linéaire ; et enfin, sur un code industriel appelé EUROPLEXUS. Pour chacune de ces classes, X-KAAPI se comporte aussi bien voir mieux que le logiciel de référence.

KEYWORDS : Parallel programming model. Multicore. Work stealing.

MOTS-CLÉS : Model de programmation parallèle. Multicœur. Vol de travail.

1. Introduction

Several research projects [11, 1, 15] have investigated the use of data flow graphs as an intermediate representation of the computation of LAPACK's algebra algorithms [6]: the main reason was that the portability of performance in LAPACK is of the responsibility of a set of basic linear algebra subprograms (BLAS), which exhibit only a low level parallelism that is not sufficient on multicores.

Without taking care of considerations about tile algorithms to reach good performances, three majors points have been identified as important to perform efficient tile algorithms [11, 1, 6]: 1/ fine granularity to reach high level of parallelism; 2/ asynchronous execution to prevent synchronization barriers; and 3/ a dynamic data driven scheduler to ensure the execution of tasks as soon as all their input data are produced.

Point 1/ was already mentioned since the first papers about Cilk [8] and formalized in the Cilk performance model [5] where the parallel time is lower bounded by the critical path: a program that cannot exploit fine grain parallelism is difficult to schedule with guaranted linear speed up, even for a reasonable number of processors. Points 2/ and 3/ are at the basis of the execution of data flow machines [2] or languages [17, 9] that try to schedule instructions as soon as input operands are produced.

Nevertheless, most of the recently proposed frameworks (StarSs [15], StarPU [3] or Quark [19]), that share the data flow graph as a central representation to dynamically schedule tasks according to their dependencies, are not able: 1/ to run programs with recursive tasks creations; 2/ to mix task and parallel loops which are important for scientific applications.

This paper presents X-KAAPI, a runtime library and a source to source compiler to program NUMA multicore machines. X-KAAPI is based on a macro data flow graph: the sequential code is annotated to identify functions to be transformed into tasks. Each function that is candidate to become a task must be annotated in order to specify the access mode (read, write, reduction, exclusive) made through its parameters to the memory. The compiler will insert task creations and the runtime will detect the dependencies and schedule the tasks onto the cores. Parallel loops are processed in the same way: the user annotates loop to be parallelized, such as in OpenMP, and the runtime creates tasks on demand in order to serve idle cores.

This paper is organized as follows. Next section describes how to install X-KAAPI. Section 2 overviews the X-KAAPI parallel programming model and how it schedules tasks at runtime. Section 3 reports preliminary experiments on three applications. Then, after the presentation of related works, we conclude the paper.

2. Programming Model

The X-KAAPI's task model [10] used in this work comes from the Athapascan [9] task model and semantics. As for Cilk [5], Intel's Threading Building Blocks (TBB) [16], OpenMP-3.0 or StarSs [15], task creation is a non blocking instruction: the caller creates the task and it continues to execute the program. Moreover, the semantic remains sequential [9]. The execution of a X-KAAPI program generates a sequence of tasks that access to data in a shared memory. From this sequence, the runtime is able to extract independent tasks in order to dispatch them to idle cores. This paper focus on the multicore version of X-KAAPI.

2.1. Parallel region

X-KAAPI defines the concept of a parallel region. A parallel region is a dynamic scope where several threads cooperate to execute created tasks. Even if the term is similar to OpenMP parallel region, the X-KAAPI parallel region does not restrict the number of threads that may concurrently execute tasks.

A parallel region is annotated by the directive `#pragma kaapi parallel`. The directive must precede either a statement or a basic block of statements [1]. At the end of a parallel region, an implicit synchronization point waits for all previously created tasks.

Parallel regions may be embedded inside other parallel regions. Embedded parallel regions share the same computing resources. Outside the outermost parallel region, only the main thread drives the execution. The runtime selects the number of threads using environment variables or available number of cores, see [14] for a detailed presentation.

2.2. Task

A X-KAAPI program is composed of C or C++ code and some annotations specified by a programmer to indicate what function to use to create tasks and when they are called.

2.2.1. Task definition and creation

A task is a function call: a function that should return no value except through the shared memory and the list of its effective parameters. The parallelism in X-KAAPI is explicit (task annotation), while the detection of synchronizations is implicit: the dependencies between tasks and the memory transfers are automatically managed by the runtime. It is of the responsibility of the programmer to annotate code.

A task implements a sequential computation whose granularity is fixed by the user; it is created in program statements that correspond to calls to functions annotated as tasks thanks to the `#pragma kaapi task` directive. Tasks share data if they have access to the same memory region. A memory region is defined as a set of addresses in the process virtual address space. This set has the shape of a multi-dimensional array.

The user is responsible for indicating the mode each task accesses to the memory: the main access modes are *read*, *write*, *reduction* or *exclusive*.

The code of figure 1 illustrates the annotation of a recursive algorithm to accumulate inputs of an array. Line 1 is the annotation of the function task accumulate. Every call to

```
1  #pragma kaapi task value(n) read(array[n]) reduction(+: result)
2  void accumulate (int n, int* array, int *result)
3  {
4    if (n < 2)
5        *result += array[0];
6    else {
7      size_t med = n/2;
8      accumulate( n/2, array, result);
9      accumulate( n-n/2, array+med, result);
10   }
11 }
```

Figure 1. *Examples of recursive accumulation*

this function is translated to task creation, such as calls at line 8 and 9. The parameter `n` is declared to be passed by value; the parameter `array` references a 1-dimensional array

1. In C or C++ these are statements enclosed between braces ('{' and '}').

of size `n` to be read; and the parameter `result` will be an accumulator where inputs are reduced using the '+' operator. Test, at line 4, is for the terminal case: Here a very fine grain is used. In real code a threshold halts the recursion.

Due to declaration, for each parameter, of the accesses mode, the X-KAAPI's runtime is able to analyze if two tasks have data flow dependency on a memory region.

2.2.2. Access mode of tasks and dependencies analysis

As illustrated in firgure 1, the access mode declaration defines two things. Firstly, it defines the way task accesses to the memory (*read*, *write*, *exclusive* = read or write, or *cumulative*, in case of reduction). Secondly, it describes the shape of the memory region of the parameter. Supported shapes are one dimensional array (1-D array) and two dimensional array (2-D array). The latter shape allows to describes sub matrix of a continuous memory region in a similar way the BLAS which are missing in both Quark [19] or StarSs [15, 4]. Thanks to the task's access mode, the runtime is able to compute [9, 10]., at runtime, concurrent tasks or to identify data flow dependency between tasks: *Read after Write dependency* if a task is reading a memory region that an older task writes. Also called true dependency; *Write after Write* or *Write after Read*, called a false dependency that may be suppressed by X-KAAPI using variable renaming [15].

2.2.3. Task synchronization

The X-KAAPI task creation is a non-blocking operation: the callee does not wait for the task completion. This implies that a variable written by a task can be read either by passing the variable to a task that declares to read it (see section 2.2.2); or after a synchronization point, using the `#pragma kaapi sync` directive: it synchronizes the control flow of the thread that creates tasks with the task executions.

2.3. Parallel loop

As in OpenMP, in X-KAAPI it is possible to annotate *parallel loop*, for which all iterations are independents. The annotation is `#pragma kaapi loop`.

The same requirements as in OpenMP exist on X-KAAPI parallel loop: the body may be evaluated concurrently for different iterations; condition and increment expressions must follow restrictions. The following example illustrates a parallel accumulation:

```
1 #pragma kaapi loop read(array[n]) reduction(+:result) nowait
2 for (int i=0; i<n; ++i)
3   result += array[i];
```

Such loop construct does not exists in data flow based software (StarSS, StarPU or Quark). In X-KAAPI, a parallel loop is an adaptive task [18], *i.e.* a task that can be split at runtime to generate parallelism, with the access mode for each memory region accessed in the body must be declared.

As for task creation, a parallel loop is a non blocking instruction. By default, the runtime adds an implicit synchronization at the end of the parallel loop. The user may explicitly specifies the clause `nowait` (see above example) to avoid this synchronization.

2.4. Execution

A X-KAAPI program begins by executing the C/C++ main entry point of the process and it initializes the X-KAAPI runtime library. When initialized, a fixed number of threads is created and the main thread continues to execute the program. Other threads are called *worker threads*. From the annotations, the X-KAAPI compiler inserts code to create tasks or adaptive task in case of parallel loop. At runtime, each created task is pushed into a

special stack of the running thread. All ready tasks (tasks for which all inputs parameters are not subject to a true dependency) in this stack may be stolen by idle threads. The runtime does not compute data flow dependencies when a task is pushed. By default, they are computed lazily by an idle thread that tries to steal ready tasks from a victim stack.

This work stealing algorithm is very close to the T.H.E protocol used in Cilk runtime [8, 5], also used in TBB [16], where lock occurs only in rare case of execution. Nevertheless, Cilk model does not have dependencies between tasks or adaptive tasks as in X-KAAPI.

3. Experimental evaluation

The multicore platform used in this section is a 48 cores AMD platform (Many-Cours) with 256GBytes of main memory. Each core frequency is 2.2Ghz.The machine is a NUMA architecture with 8 NUMA nodes, each of them composed by 6 cores. Reported times are averages over 10 runs.

3.1. Task creation time

This section is devoted to the overhead of using X-KAAPI with respect to sequential computation. The experimentation evaluates the time to execute the X-KAAPI program of figure 2 using 1, 8, 16, 32 and 48 cores. Sequential time is $0.091s$. Figure 2 reports times using state of the art softwares such as TBB [16] and OpenMP-3.0 (gcc 4.6).

```
1  #pragma kaapi task write(result) value(n)
2  void fibonacci(long* result, const long n)
3  {
4    if (n<2)
5      *result = n;
6    else
7    {
8      long r1,r2;
9      fibonacci( &r1, n-1 );
10     fibonacci( &r2, n-2 );
11 #pragma kaapi sync
12     *result = r1 + r2;
13   }
14 }
```

a. Benchmark

#cores	TBB	Kaapi	OpenMP
1	2.356	0.728	2.429
(slowdown:1)	(x 26)	(x 8)	(x27)
8	0.293	0.094	*51.06*
16	0.146	0.047	*104.14*
32	0.072	0.024	*(no time)*
48	0.049	0.017	*(no time)*

b. Time (second)

Figure 2. *Fibonacci micro benchmark. Sequential time is 0.091s.*

Benchmark codes for OpenMP or TBB are not listed but they have exactly the same number of tasks spawned and the same synchronizations. TBB has more overhead with respect to the sequential computation (slowdown of about 26) in comparison to X-KAAPI (slowdown of 8). This overhead is due to higher cost to create tasks which is the key point in this benchmark. Times of OpenMP (gcc 4.6) are very bad: the grain is to fine and OpenMP cannot speed up the computation! The relative good time of OpenMP using 1 core is due to an artifact in the libgomp implementation of gcc, where this special case is tested in order to degenerate task creation to standard function call.

3.2. Cholesky factorization

The Cholesky factorization is an important algorithm in dense linear algebra. In this section, we report performances of the block version of PLASMA 2.4.2 [1]. On multicore architecture, PLASMA relies on a runtime, called Quark [19], to manage the tasks with data flow dependencies. Quark provides a subset of functionalities offered by X-KAAPI. Thus, we have ported Quark on top of X-KAAPI to produce a binary compatible Quark library, which is linked with PLASMA algorithms for X-KAAPI experimentations.

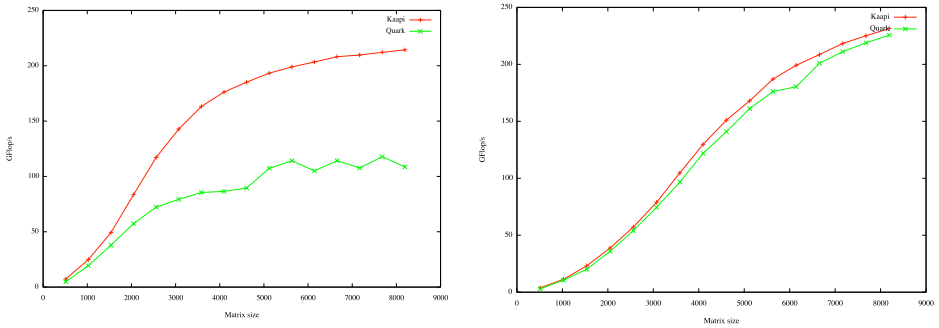

Figure 3. *Gflops on Cholesky algorithm with Quark and Kaapi.*

Figure 3 reports the performances (GFlop/s) with respect to the matrix size on 48 cores. The size of the tile is $NB = 128$ on the left, and $NB = 256$ on the right. Quark implements a centralized list of ready tasks, with some heuristics to avoid accesses to the global list. For a relative fine grain tasks ($NB = 128$) and due to a contention of the accesses to global list, the Kaapi outperforms the performances of Quark. Quark will probably limit the performance of PLASMA for the next generation of multi-core with hundreds of cores. When the grain increases, X-KAAPI remains better but the difference decreases because of the relative small impact of the task's management with respect to the whole computation.

3.3. Comparison with OpenMP on the industrial code EUROPLEXUS

EUROPLEXUS [7] is a computer code being jointly developed since 1999 by CEA (CEN Saclay, DMT) and EC (JRC Ispra, IPSC) under a collaboration contract.

The code analyses 1-D, 2-D or 3-D domains composed of solids (continua, shells or beams) and fluids. Fluid-structure interaction is also taken into account. The program uses an explicit algorithm (central-difference) for the discretization in time and therefore it is best adapted to rapid dynamic phenomena (fast transent dynamics) such as explosions, impacts, crashes etc. The spatial discretization is mainly based on the Finite Element or Finite Volume method.

The parallelism in EUROPLEXUS is based on parallel loops. The original multicore version is based on OpenMP using dynamic schedule and it was extended to use X-KAAPI parallel loop.

Figure 4 reports speedup of two parallel implementations. The OpenMP version is based on the default static scheduling. The same cores was used in both X-KAAPI or OpenMP. Globally, the two speedups are very close. X-KAAPI has a better speedup for a larger number of cores (>25).

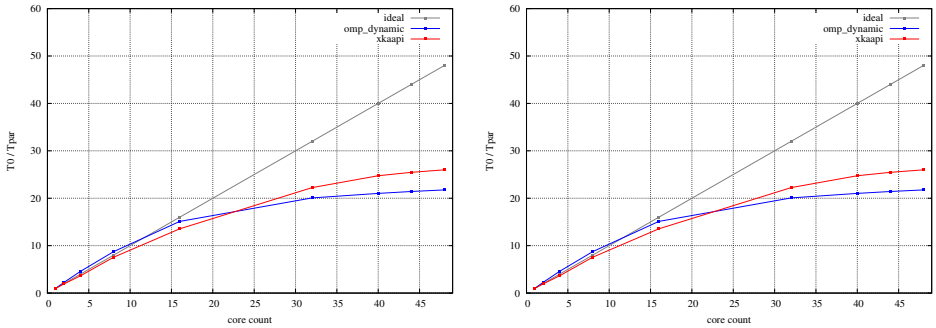

Figure 4. *Speedups for the two internal parallel loops in EUROPLEXUS.*

4. Related works

The X-KAAPI data flow programming model comes from Athapascan [9] which, itself, was inspired by Jade [17]. The X-KAAPI programming model and StarSs [15, 4] are very close. The main differences are: 1/ StarSs does not allows recursive task creations; 2/ with StarSs it is not possible to define memory region of a sub-matrix of a matrix [14]; 3/ StarSs does not provide support for parallel loop [18]. Quark [19] and StarPU [3] share the same limitations of StarSs, except the possibility to define sub-matrix memory region in StarPU.

Cilk [8] allows recursive fork-join parallel construction. Intel TBB [16] is inspired from Cilk. With the autopartitionner, TBB adapts the task creations in parallel loop to the thread inactivities. OpenMP has recently add the concept on task (OpenMP version 3.0, 2008) but the public gcc-4.6 implementation is far away of the performance of Cilk, TBB or X-KAAPI. Nevertheless, none of these softwares (Cilk, TBB or OpenMP) allows to describe finer data flow synchronizations which are important for high performance computing in linear algebra [13].

5. Conclusions

In this paper, we overview the parallel programming model of X-KAAPI. Based on code annotation coupled with a runtime support, X-KAAPI has important original features: 1/ the ability to describe dependency on non contiguous memory region (required for linear algebra subroutines) ; 2/ X-KAAPI allows task to create (recursive) tasks with data flow dependencies; 3/ X-KAAPI mix data flow dependencies between tasks and parallel loop.

Section 3 report times on three different classes of applications: X-KAAPI has similar or best performances with respect to all other tested softwares. And none of the tested software can be used directly for all the three applications (missing data flow in TBB or OpenMP), missing recursive task creation for StarSs, StarPU and Quark, missing parallel loop construct in StarSS, Quark and StarPU.

Ongoing work is to make our compiler infrastructure more robust and to integrate our past multi-CPUs multi-GPUs support [12]. Future work will deals with integrating sparse Cholesky factorization in the EUROPLEXUS code.

References

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, P. Ltaief, H.and Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180, 2009.

[2] Arvind and D E Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, 1986.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, August 2009. Springer.

[4] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using smpss. *Concurr. Comput. : Pract. Exper.*, 21:2438–2456, December 2009.

[5] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27:202–229, February 1998.

[6] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[7] Europlexus. CEA (CEN Saclay, DMT) and EC (JRC Ispra, IPSC), http://europlexus.jrc.ec.europa.eu/.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[9] F. Galilée, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 88–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO'07*, 2007.

[11] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[12] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In *EUROPAR 2010*, Ischia Naples, Italy, aug 2010.

[13] J. Kurzak, H. Ltaief, J. Dongarra, and R. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, Vol. 22, no. 1:pp. 15–44, 2010.

[14] F. Le Mentec, T. Gautier, and V. Danjean. The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming. Rapport Technique RT-0418, INRIA, December 2011.

[15] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *CLUSTER*, pages 142–151. IEEE, 2008.

[16] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[17] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20:483–545, May 1998.

[18] D. Traore, J.-L. Roch, N. Maillard, T. Gautier, and J. Bernard. Deque-free work-optimal parallel STL algorithms. In *EUROPAR 2008*, Las Palmas, Spain, aug 2008. Springer-Verlag.

[19] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee, 2011.