The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming

Fabien Le Mentec, Thierry Gautier, Vincent Danjean

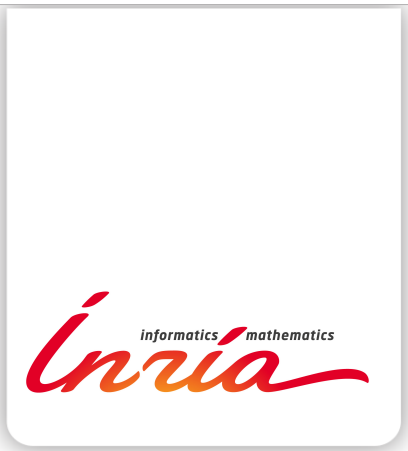
The X-KAAPI's Application Programming Interface.

Part I: Data Flow Programming

Fabien Le Mentec, Thierry Gautier, Vincent Danjean

TECHNICAL REPORT
N° 0418
November 2011
Project-Teams MOAIS

ISSN 0249-0803 ISRN INRIA/RT--0418--FR+ENG



The X-Kaapi's Application Programming Interface. Part I: Data Flow Programming

Fabien Le Mentec, Thierry Gautier, Vincent Danjean

Project-Teams MOAIS

Technical Report n° 0418 — November 2011 — 45 pages

Abstract:
In this report, we present X-Kaapi's programming model. A X-Kaapi parallel program is a C or C++ sequential program with code annotation using `#pragma` compiler directives that allow to create tasks. A specific source to source compiler translates X-Kaapi directives to runtime calls.

Key-words: parallel computing, data flow graph, scheduling, X-Kaapi

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

X-Kaapi : interface de programmation parallèle

Résumé :

Ce rapport présente le modèle de programmation X-Kaapi qui permet d'annonter un programme séquentiel écrit en C ou C++ par des directives de compilation `#pragma` afin de décrire simplement les tâches du programme. Un compilateur source à source génère un code qui permet, grâce au runtime X-Kaapi, d'extraire à l'exécution ce graphe de flot de données, y compris pour les programmes récursifs dont les tâches seront générées récursivement.

Mots-clés : programmation parallèle, graphe de flot de données, ordonnancement, X-Kaapi

Contents

1 Context

Several research projects [11, 2, 13] have investigated the use of data flow graphs as an intermediate representation of the computation of LAPACK's algebra algorithms [7]: 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 performances, three majors points have been identified as important to perform efficient tile algorithms [11, 2, 7]: 1/ fine granularity to reach high level of parallelism; 2/ asynchronous execution to prevent barriers of synchronization; 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 [6] 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 [3] or languages [14, 9] that try to schedule instructions as soon as input operands are produced.

Nevertheless, most of the recently proposed frameworks that share the data flow graph as a central representation to dynamically schedule tasks according to their dependencies, are not able to run programs with recursive tasks creations.

This report 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.

The X-Kaapi programming model comes from Athapascan [9] which, itself, was inspired by Jade [14] and Cilk [8]. Because the X-Kaapi programming model and StarSs [4] are very close, we also accept the execution of StarSs programs. Descriptions of X-Kaapi may be found in following publications [10, 15, 5, 12].

This technical report is organized as follows. Next section describes how to install X-Kaapi. Section 3 presents though simple examples how to program with X-Kaapi in C or C++. Section 4 describes the programming model and all the `#pragma` annotations and the semantics of X-Kaapi program. Several complete examples are given in section 5.

2 Software installation

X-KAAPI is both a programming model and a runtime for high performance parallelism targeting multicore and distributed architectures. It relies on the work stealing paradigm. X-KAAPI is developed in the MOAIS INRIA project by Thierry Gautier, Fabien Le Mentec, Vincent Danjean and Christophe Laferrière in the early stage of the library.

In this report, only the programming model based on code annotations with `#pragma` compiler directives is presented. The runtime library comes also with a full set of complementary programming interfaces: C, C++ and STL-like interfaces. The C++ and STL interfaces, at a higher level than the C interface, may be directly used for developing parallel programs or libraries.

Supported Platforms

X-KAAPI targets essentially SMP and NUMA platforms. The runtime should run on any system providing:

- a GNU toolchain (4.3),
- the pthread library,
- Unix based environment.

It has been extensively tested on the following operating systems:

- GNU-Linux on x86_64 architectures,
- MacOSX/Intel processor.

There is no version for Windows yet.

X-KAAPI Contacts

If you wish to contact the XKaapi team, please visite the web site at:

`http://kaapi.gforge.inria.fr`

2.1 Installation

To install X-KAAPI programming environment, you need to:

- install the ROSE compiler framework,
- compile the X-KAAPI library and the X-KAAPI compiler,
- install the X-KAAPI library and the X-KAAPI compiler.

The X-KAAPI libraries and runtime are available at `http://kaapi.gforge.inria.fr` (tarball). X-KAAPI libraries and runtime are distributed under a CeCILL-C license[1]:

[1] `http://www.cecill.info/index.en.html` for english version

CeCILL-C is well suited to libraries and more generally software components. Anyone distributing an application which includes components under the CeCILL-C license must mention this fact and make any changes to the source code of those components available to the community under CECILL-C while being free to choose the licence of its application.

Moreover, the X-KAAPI compiler is build on top of the ROSE[2] framework to develop source-to-source translators. In this framework, the C and C++ frontends are based on the EDG frontend. ROSE is available under a BSD license, but the binary code for EDG must be downloaded separately.

2.1.1 ROSE and EDG frontend installation

The ROSE framework (`http://www.rosecompiler.org/`) must be installed. Please have a look at the installation guide on the ROSE web site.

In order to install ROSE, you need to have (valid for ROSE 0.9.5a):

- `wget`: to download the binary version of the EDG frontend
- `gcc, g++`: at least version 4.0.0
- `boost`: version 1.36.0 to 1.45.0
- and a lot of different utilities (GNU autotools, ...)

Detailed procedures to install ROSE and EDG frontend are described in ROSE documentation.

Once ROSE is installed, you will be able to generate `kacc`, the X-KAAPI compiler.

2.1.2 X-KAAPI library and compiler installation

Currently, X-KAAPI must be installed from sources. Other packagings (Debian or RPM packages) are under development. Please visit the X-KAAPI web site for an up-to-date information:

`http://kaapi.gforge.inria.fr`

Installing from sources

There are 2 ways to retrieve the sources:

- download a release snapshot at the following url:
  `https://gforge.inria.fr/frs/?group_id=94`.
- clone the master branch of the git repository of the project:
  `> git clone git://git.ligforge.imag.fr/git/kaapi/xkaapi.git xkaapi`

[2] `http://www.rosecompiler.org`

Configuration. The build system uses GNU Autotools. In case you cloned the project repository, you first have to bootstrap the configuration process by running the following script:

```
$> ./bootstrap
```

The *configure* file should be present. It is used to create the *Makefile* accordingly to your system configuration. Command line options can be used to modify the default behavior. A list of the configuration options is available by running:

```
$> ./configure --help
```

Below is a list of the most important ones:

- `--enable-mode=debug` or `release`
  Choose the compilation mode of the library. Defaults to release.
- `--with-hwloc`
  To compile with hwloc (`http://www.open-mpi.org/projects/hwloc`) library to take advantage of the topological information of the computer.
- `--with-perfcounter`
  Enable performance counters support.
- `--with-papi`
  Enable the PAPI library for low level performance counting. More information on PAPI can be found at http://icl.cs.utk.edu/papi/.
- `--enable-kacc`
  required to compile the `kacc` compiler. Depending on your installation of ROSE and the Boost library, you may specify their installation paths using:
  `--with-rose=<rose installation>`
  or
  `--with-boost=<boost installation>`
- `--prefix=`
  Overload the default installation path.

Example:

```
./configure --enable-mode=release --prefix=$HOME/install
./configure  --prefix=$HOME/install --enable-mode=release \
         --with-hwloc=/usr/local/hwloc --with-perfcounter \
         --with-boost=/usr/local/boost \
         --with-rose=$HOME/ROSE/install \
         --enable-kacc
```

If there are errors during the configuration process, you have to solve them before going further. If dependencies are missing on your system, logs will give you the names of the software to install.

Compilation and installation. On success, the configuration process generates a Makefile. The 2 following commands build and install the X-KAAPI runtime:

```
$> make
$> make install
```

Checking the installation. The following checks that the runtime is correctly installed on your system:

```
$> make check
```

Compilation of the examples. The following compiles the sample applications:

```
$> cd examples; make examples
```

Installation directory

The configure, make, make install, commands create in the prefix directory the following directory structure:

```
<prefix>/include
<prefix>/lib
<prefix>/bin
<prefix>/share
```

The KaCC compiler is located in `<prefix>/bin`, it needs to be set in your `$PATH` variable to avoid full path naming. All programs compiled with KaCC are linked against libraries located in `<prefix>/lib`. Executables linked with KaCC do not require to add this directory in your `LD_LIBRARY_PATH` under linux, and `DYLD_LIBRARY_PATH` under Mac OS X.

The `<prefix>/share` directory contains some documentation and examples.

3 Getting started

The source distribution of X-KAAPI contains the repository `examples`. Each subdirectory is dedicated to a problem (fibonacci, matrix product, ...). The examples that come with the compiler are located in `examples/compiler`. A presentation of the examples using the X-KAAPI library directly (with C, C++ or STL) interface is out of the scope of this report.

In the remaining, we only present the X-KAAPI programming model and we show examples using the `kacc` compiler only. The process of describing a parallel program with X-KAAPI is:

- X-KAAPI has a task based parallel programming model: the user must indicate where tasks are.
- A task is a function call without side effect: the user must annotate the code to specify what function is a task.
- The compiler inserts task creation code for each appropriate function call.
- Dependencies between tasks are automatically computed by the runtime through data flow analysis between function calls: when a function is annotated as a task, the user must specify the memory access mode for each formal parameter.

All the annotations are specified by pragma directives with the form:

```
#pragma kaapi <clause specification>
```

3.1 Hello World

The example of figure 1 presents the creation of one task that prints "Hello world". The X-KAAPI `kacc` compiler is based on pragma annotations to describe the parallelism, as for OpenMP.

First it is necessary to start a parallel region where tasks may be executed concurrently. Outside a parallel region tasks are executed only by the current thread. The construction `#pragma kaapi parallel` at line 11 starts the parallel region: it begins at the next basic block (between `{` and `}`) and it ends at the end of the basic block. Without brackets it only concerns the following statement.

The `#pragma kaapi parallel` defines a region where concurrent executions may occur. The number of threads used to execute tasks inside a parallel region may vary during the execution. All tasks created inside a parallel region are guaranteed to be completed at the end of the parallel region.

The task in the figure takes one parameter (a C string) and displays it to the screen. The clause `value(msg)` tells the compiler the effective parameter is considered as being passed by value. We will see in the following sections what it means.

3.1.1 Compilation

To compile the previous example, simply type:

```
> kacc -o hello_world hello_world.c
```

```
1  #include <stdio.h>
2
3  #pragma kaapi task value(msg)
4  void say_hello(const char* msg)
5  {
6    printf("%s\n", msg);
7  }
8
9  int main(int argc, char** argv)
10 {
11 #pragma kaapi parallel
12     say_hello("Hello World!");
13   return 0;
14 }
```

Figure 1: hello_world1.c example

Then, run it with:

```
> ./hello_world
Hello World!
```

This simple example only creates one task without real concurrency[3].

3.1.2 Two Hello World

The next example in figure 2 creates 2 independent tasks that print 2 messages.

```
1  #include <stdio.h>
2
3  #pragma kaapi task value(msg)
4  void say_hello(const char* msg)
5  {
6    printf("%s\n", msg);
7  }
8
9  int main(int argc, char** argv)
10 {
11 #pragma kaapi parallel
12   {
13     say_hello("First call: Hello World!");
14     say_hello("Second call: Hello World!");
15   }
16   return 0;
17 }
```

Figure 2: hello_world2.c example

Executing the program should produce one of the following output:

[3]In fact, there is concurrency between the task execution and the code executed between the end of the function call at line 12 and before the instruction at line 13: it is possible that the task is theft and executed by another thread than the main thread. Nevertheless, at the end of the parallel region, the task is completed.

```
> ./hello_world2
First call: Hello World!
Second call: Hello World!
```

Or:

```
> ./hello_world2
Second call: Hello World!
First call: Hello World!
```

In X-KAAPI the task creation is a non-blocking operation: the thread that creates tasks never waits for the completion of the tasks if it is not specified by the user. All tasks created into a parallel region are guaranteed to be completed at the end of the parallel region: the end of a parallel region has an implicit synchronization point that forces execution of tasks. But inside a parallel region, tasks may be executed in **any order** that respects the data flow dependencies between tasks. In the example, tasks have been declared as taking their parameter by value so they are independent.

There are 3 ways for X-KAAPI to enforce an execution order:

- Tasks have declared special access modes to their parameters (read, write, exclusive), which introduce read-after-write data flow dependencies between tasks: they are the only dependencies that X-KAAPI respect at runtime. This is the subject of the next section.
- If tasks are independent, then the user may introduce a synchronization point using the `#pragma kaapi sync` directive to force all tasks created in the same function body before the synchronization point to be completed at the end of the synchronization point. In the previous example, this directive must be placed between line 13 and line 14:

```
11 #pragma kaapi parallel
12   {
13     say_hello("First call: Hello World!");
14 #pragma kaapi sync
15     say_hello("Second call: Hello World!");
16   }
```

- The user may also define a sequence of parallel regions in order to take benefit of the implicit synchronization at the end of each parallel region. The code of figure 2 then becomes:

```
11 #pragma kaapi parallel
12     say_hello("First call: Hello World!");
13 #pragma kaapi parallel
14     say_hello("Second call: Hello World!");
```

3.2 A simple writer-reader scheme

The example of figure 3 creates two tasks in a parallel region defined at line 18. The tasks are created at line 20 and 22.

The first task takes 3 parameters: the first one is the size of the buffer which is given as the second parameter. A third parameter points to the message to copy into the buffer. The respective access modes for these parameters, defined at line 3, are:

```
1  #include <stdio.h>
2
3  #pragma kaapi task write(buffer[size]) value(size) read(msg)
4  void write_msg(int size, char* buffer, const char* msg)
5  {
6    snprintf(buffer, size, "%s", msg);
7  }
8
9  #pragma kaapi task read(msg)
10 void print_msg(const char* msg)
11 {
12   printf("%s\n", msg);
13 }
14
15 int main(int argc, char** argv)
16 {
17   char buffer[32];
18   #pragma kaapi parallel
19   {
20     write_msg(32,buffer,
21       "This is a long message for the buffer.");
22     print_msg(buffer);
23   }
24
25   return 0;
26 }
```

Figure 3: simple_prodcons.c example

- **by value access mode**, for the parameter `size` defined by `'value(size)'`. This access mode is like the pass-by-value rule of a C/C++ function call: The callee copies the effective parameter[4] to the formal parameter[5]. The callee receives a copy of the value that can be modified. Modifications are not viewed by the caller.

- **write access mode** of `buffer` defined by `'write(buffer[size])'`. In this example, the function writes characters into a buffer. The clause specifies the size of the buffer (`buffer[size]`) written by the task. Modifications of the buffer during task execution are reported to the callee. With this access mode, the input value of the parameter `buffer` is unspecified.

- **read access mode** for the variable `msg` defined by `'read(msg)'`. This parameter is the C string read by the task and copied into the parameter `buffer`. The size of the string follows the C convention of null terminated string.

The second task declares (line 9) it reads its single parameter.

The same effective parameter `buffer`, declared at line 17, is passed between the two tasks at line 20 and 21. Because of the previous declaration of `write` access in the first task and `read` access in the second task, it induces a **true** data flow dependency between the tasks.

[4]The expression passed in the function call.
[5]The name of the variable in the function definition

In X-KAAPI, any parameter that has to be shared between tasks must be a pointer. In the previous example, it is a C-string, *i.e.* a pointer to a null terminated array of characters.

3.3 Recursive computation: Fibonacci

Figure 4 presents a recursive computation of the n-th Fibonacci number[6] using the classical non optimal algorithm.

```
1  #include <stdio.h>
2
3  #pragma kaapi task write(result) value(n)
4  void fibonacci(long* result, const long n)
5  {
6    if (n<2)
7      *result = n;
8    else
9    {
10     long r1,r2;
11     fibonacci( &r1, n-1 );
12     fibonacci( &r2, n-2 );
13 #pragma kaapi sync
14     *result = r1 + r2;
15   }
16 }
17
18 #pragma kaapi task read(result)
19 void print_result( const long* result )
20 {
21   printf("Fibonacci(30)=%li\n", *result);
22 }
23
24 int main()
25 {
26   long result;
27 #pragma kaapi parallel
28   {
29     fibonacci(&result, 30);
30     print_result(&result);
31   }
32   return 0;
33 }
```

Figure 4: fibonacci1.c example

The main function creates two tasks: a first one to do the computation (line 29) and a second one to print the result (line 30). Due to access modes, those two tasks have a true dependency.

The `fibonacci` task (lines 3-16) recursively creates two tasks until the input parameter `n` gets smaller than 2.

A task creation is a non blocking call, thus at line 13 a synchronization point is added (`#pragma kaapi sync`):

- this directive enforces that all previously created tasks in the same function

[6]For simplification, n was fixed to 30 in the code.

body[7] be completed before proceeding with the next instruction, at line 14, to sum the sub results.

- this directive does **not** wait for the completion of tasks created in the outter function calls.

3.3.1 Optimization

The code of figure 4 has a main drawback: it requires a synchronization at each step of the recursion. Even if, in the general case of execution, this synchronization does not wait because tasks are generally executed sequentially, it adds an extra overhead.

In this section, we present how to suppress this overhead by creating a task that will represent the *continuation* of the computation. The code is presented in figure 5. Instead of adding a synchronization point, we create, at line 18, a task `sum`, with true data flow dependencies with the previous tasks (on variables `r1` and `r2`). This `sum` task does the summation (task sum definition at line 1) of the two sub-results computed by the recursive tasks created at line 16 and 17.

```
1  #pragma kaapi task write(result) read(r1,r2)
2  void sum( long* result, const long* r1, const long* r2)
3  {
4    *result = *r1 + *r2;
5  }
6
7  #pragma kaapi task write(result) value(n)
8  void fibonacci(long* result, const long n)
9  {
10   if (n<2)
11     *result = n;
12   else
13   {
14 #pragma kaapi data alloca(r1,r2)
15     long r1,r2;
16     fibonacci( &r1, n-1 );
17     fibonacci( &r2, n-2 );
18     sum( result, &r1, &r2);
19   }
20 }
```

Figure 5: fibonacci2.c example.

The advantage of the continuation passing style is to replace an explicit coarse grain synchronization (`#pragma kaapi sync`) by an implicit data flow synchronization. It is done by creating the task `sum`.

Because the tasks created at lines 16, 17 and 18 are non blocking calls, the user must take care of the scope of the variables passed to the tasks. In the previous example, because `r1` and `r2` are allocated in the C stack of the function `fibonacci`, the user must specify at line 14 the directive `#pragma kaapi data alloca` to annotate explicitly those data: Then the compiler will allocate them so that their scope is sufficient for the execution of the task.

[7] More formally, in the same activation frame as the function body execution.

4 X-Kaapi programming model

The task model used in this work comes from the Athapascan [9] task model and semantics. As for Cilk, Intel TBB, OpenMP-3.0 or SMPSs, the semantic remains sequential. It ensures that the value returned by the read statements is the last written value according to the lexicographic order defined by the program: statements are lexicographically ordered by ';'. 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.

In the following sections, we outline the X-Kaapi programming model; Then we specify different X-Kaapi constructions and their semantic.

4.1 Task execution model

A X-Kaapi [10] 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. A X-Kaapi program begins by executing the C/C++ main entry point of the process. Then, everytime a call to an annotated task function is encountered, a task is created and the control flow proceeds its execution until it reaches a synchronization point.

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. It is of the responsibility of the programmer to annotate functions which calls become tasks, by using the `#pragma kaapi task` directive. 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.

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 uses to access to the memory: the main access modes are *read*, *write*, *reduction* or *exclusive*.

One of the main characteristics of the X-Kaapi programming model is the preservation of the sequential semantic of the program: the result of the parallel execution remains the same as the sequential execution of the program compiled without X-Kaapi.

4.2 Task

X-Kaapi is called a macro data flow language: The user is responsible for defining the granularity of the computation (task) as well as data shared between tasks. The runtime automatically schedules tasks in any order that respects

the true data flow dependencies, *i.e.* the data flow dependencies. All other dependencies are subject to be removed by renaming variables.

This section presents how to annotate a function declaration in such a way that ALL calls to this function are translated to creations of tasks. This is called *function task* annotation.

4.2.1 Function Task

In order to replace all function calls by task creations, the user must:

- Annotate the *declaration* of the selected function. The declaration is the statement that declares the signature of the function (return type, name of the function, list of the parameters with type). A declaration can be followed by the *definition* of the function, *i.e.* by the instructions that form the body of the function. This is called a *defining declaration*.
- In case of library of functions where function declarations cannot be changed, the users must re-declare the functions with the X-KAAPI corresponding annotations.

Once defined, all calls to annotated functions are compiled to task creations. The runtime runs every task that is ready by performing the instructions of the function body.

Function task declaration.
The code of figure 6 illustrates the annotation of two functions selected to be *function tasks*:

- The `#pragma kaapi task` annotation must precede the function declaration
- The access mode of each formal parameter must be defined as well as the memory region accessed by the task.

```
1 #pragma kaapi task value(n) read(array[n]) write(result)
2 void accumulate (int n, int* array, int *result);
3
4 #pragma kaapi task reduction(+: result)
5 void compute( int* result );
```

Figure 6: Examples of two declarations of function tasks

At line 1, the parameter `n` is declared to be passed by value; the parameter `array` references a 1-dimensional array of size `n` that is to be read; and the parameter `result` is written by the task to store a result. At line 4, the function task `compute` takes one parameter (`result`) defined to be a reduction variable.

The section 4.4 presents all the different ways to declare access mode for parameters.

```
1 #pragma kaapi task value(n) read(array[n]) write(result)
2 void accumulate (int n, int* array, int *result);
3
4 void accumulate (int n, int* array, int *result)
5 { result = std::accumulate(array, array+n ); }
```

Figure 7: Declaration and definition of a function task

Function task definition.
As for functions, a function task declaration represents the signature of a function. The body of the task is the function body of the annotated function. It contains the instructions performed at runtime to execute the task.

In figure 7, line 1 and line 2 correspond to the function task declaration. Line 3 through line 4 to the function task definition.

Figure 8 illustrates the case where the declaration and the definition are given by the same statement, i.e. this is a defining declaration of the function:

```
1 #pragma kaapi task value(n) read(array[n]) write(result)
2 void accumulate (int n, int* array, int *result)
3 { result = std::accumulate(array, array+n ); }
```

Figure 8: Defining declaration of a function task

Task creation.
After the declaration of a function task, The X-KAAPI compiler replaces every call to the function by a task creation. Recursive task creations are allowed.

A major difference with the sequential call to function is that 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:

- By passing the variable to a task that declares to read it (see section 4.4).
- After a synchronization point (see section 4.5).

4.3 Parallel region

A parallel region is annotated by the directive `#pragma kaapi parallel` (see line 6 of figure 9). The directive must precede:

- A statement,
- A basic block of statements defined between braces ('`{`' and '`}`').

Every tasks created inside a parallel region may be executed concurrently: a parallel region is a dynamic scope where several threads cooperate to execute created tasks.

At the end of a parallel region, an implicit synchronization point waits for all previously created tasks.

```
1 int main(int argc, char** argv)
2 {
3   int size;
4   int* A = ...;
5   int result;
6 #pragma kaapi parallel
7   accumulate (size, A, &result);
8   return 0;
9 }
```

Figure 9: Parallel region example

Parallel regions may be embedded inside other parallel regions. Before the control flow enters a parallel region, the execution is driven by the main thread. Outside the outermost parallel region, only the main thread drives the execution.

This does not imply that synchronization becomes unnecessary in the main thread: task creation is always a non blocking instruction that may require to be synchronized with the C/C++ sequential execution flow.

4.3.1 Contextual parallel region definition

The scope of a parallel region is the next instruction or the next basic block (between '`{`' and '`}`') if the directive is placed just before. For instance in figure 9, the declaration of the parallel region at line 6 only applies to the statement at line 7 that creates a task.

The runtime selects the number of threads using in the following order:

1. X-KAAPI creates `KAAPI_CPUCOUNT` threads on the cores given by the variable `KAAPI_CPUSET`.
2. Else, if `KAAPI_CPUCOUNT` is undefined, X-KAAPI creates one thread mapped on each core defined in `KAAPI_CPUSET`.
3. Else, if `KAAPI_CPUSET` is undefined, X-KAAPI creates `KAAPI_CPUCOUNT` threads scheduled by the operating system.
4. Else, X-KAAPI creates one thread per physical core of the machine.

See appendix C for a detailed description of these environment variables.

4.3.2 Enclosed parallel region

Parallel regions may be defined inside parallel regions. At the difference of OpenMP, a parallel region is not associated with a number of threads and the enclosed parallel region does not create another set of threads: the threads are shared among the region and the number of threads is controlled by X-KAAPI environment variables `KAAPI_CPUCOUNT` and `KAAPI_CPUSET` (see above).

The only guarantees are:

1. an implicit synchronization point is inserted at the end of every parallel region;
2. before any parallel region and outside the outermost parallel region, the tasks are performed only by the main thread.

4.4 Access mode of tasks

Two tasks sharing data have the same effective parameter, *i.e.* a region into the shared memory. A task should declare how it accesses regions in the shared memory. Several access modes[8] have been defined based on our original work in Athapascan [9]: **read**, **write**, **readwrite** or **exclusive** and **cumulative write** (also called **reduction**). Each of those access modes limits the kind of operation a task can perform on a memory region. For instance, a task that declares an read access mode for one of its effective parameters cannot write.

A task formal parameter that is not annotated by an access mode is considered as a compilation error. The access mode is defined for a memory region, *i.e.* a set of addresses in the virtual address space of the processus.

The access modes of formal parameters are declared with:

```
#pragma kaapi task <mode> ( <list_declaration> )
```

Where `<mode>` is given in table 1.

The table 1 resumes the different access modes:

Access mode	`<mode>` in Kaapi clause	StarSs clause
pass-by-value	`value(x)`	no such access
read access	`read(x)` or `r(x)`	`input(x)`
write access	`write(x)` or `w(x)`	`output(x)`
exclusive access	`readwrite(x)` or `rw(x)`	`inout(x)`
cumulative write	`reduction(op:x)` or `cw(op:x)`	`reduction(x)`

Table 1: Different access modes in X-KAAPI. Equivalence with StarSs.

4.4.1 Memory region

The `<list declaration>` defines a list of memory regions need by the task. The memory region is described by

- A memory address given by a variable.
- A shape that describes a multidimensional array.

In listing of figure 10, line 1 declares 3 memory regions:

1. `n`: this is the memory region that begins at the address of the variable `n` with a size of `sizeof(int)` byte long.
2. `array[n]`: this a 1-dimensional array that starts at address `array` and finishes at address `array+n`.
3. `result`: its memory region corresponds to the storage of the variable of type `int` pointed by the value of `result`.

[8] These access modes have either a direct correspondence in SMPSs/StarSs, called direction as in CORBA. Same access modes exist in Quark, the PLASMA abstract layer for task creations.

```
1 #pragma kaapi task value(n) read(array[n]) write(result)
2 void accumulate (int n, int* array, int *result);
3
4 #pragma kaapi task value(N,lda,ldb,ldc)   \
5                    write(B{ld=lda, storage=C, [N][N]}) \
6                    read(B{ld=ldb, [N][N]}, C{ld=ldc, [N][N]} )
7 void matmul(int N, double* A, int lda, \
8                    const double* B, int ldb,
9                    const double* C, int ldc);
```

Figure 10: Example of memory region declarations

At lines 4-6 of the same listing, multidimensional shapes memory regions are defined for formal parameters `A`, `B` and `C`. X-KAAPI considers by default two kinds of storage for matrices. Those are illustrated in Figure 11: in (a) the C storage (also called row major) and in (b) the fortran storage (called column major). The function task `matmul` of listing 10 is supposed to do a matrix multiplication of two matrices of dimension `NxN`. The 2 input matrices `B` and `C`, as well as the output matrix `A`, are defined to have dimension `[N][N]`, where `N` is a pass-by-value parameter. The dimension could be any C or C++ arithmetic expressions (for instance `[2][3+N]`), see grammar in appendix B.1.3.

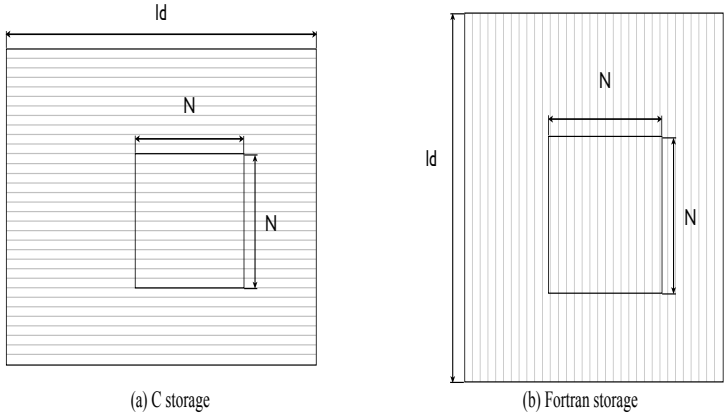


Figure 11: Two standard storage classes for matrices

1-D array.
They are two classical ways to specify a one dimensional array. The first one gives a base address and a size:

```
#pragma kaapi task value(n) read(array[n]) write(result)
void accumulate( int n, int* array, int* result );
```

The dimension of array must follow the subset of the C grammar for expression (called *additive expression* in C grammar, see appendix B). At runtime, the base address is the value of a pointer type formal parameter.

The second form is to specify an interval between two addresses (as used in most of the STL algorithms). The syntax for an interval is `[a:b]`, where `a` and `b` are identifiers of formal parameters.

```
#pragma kaapi task value(n) read( [begin:end] ) write(result)
void accumulate( int* begin, int* end, int* result );
```

In the STL, the right bound is not part of the interval and it is called the 'past the last' value of the iterator. The size of the interval is `b-a`. In C, `a` and `b` must be a pointer to any type; in C++ they can also be random access iterators.

2-D array.
2-D arrays appear de facto in linear algebra subroutines. To split data structures for parallel sub-computations, it is necessary to handle subsets of 2-D arrays.

X-KAAPI follows the design of BLAS or LAPACK libraries: the matrix is assumed to be allocated into a contiguous array and the user must specify the *leading dimension* (*ld* in figure 11), which is the size[9] to pass from one row to the next one with the C storage (from one column to the next column with Fortran convention).

For instance:

```
#pragma kaapi task mode( array[n][m] )
#pragma kaapi task mode( array{ ld=identifier, [n][m] } )
#pragma kaapi task mode( array{ storage=C, \
                                ld=identifier, \
                                [n][m] } )
#pragma kaapi task mode( array{ storage=Fortran, \
                                ld=identifier, \
                                [n][m] } )
```

The syntax of the clause for 2-D array is composed by the specification of three attributes:

- The *leading dimension*. If the leading dimension is not equal to the column dimension (for C storage convention, or row dimension for Fortran convention), it must be specified for 2-dimensional array using the attribute `ld=<identifier>`. The identifier must be the identifier of a formal parameter.
- The storage attribute: C or Fortran (or F) keywords are used to specify the storage of the matrix. `storage=C` for row major C storage. And `storage=Fortran` for Fortran column major. If not specified, the default value is the `C`.
- The dimension must follow the subset of the C grammar for expression.

There is no order to specify attributes. All those rules are defined in the grammar in the appendix B.

[9] The size is expressed with respect to the *element type*.

4.4.2 Pass-by-value rule

Any function parameter passed by value must have a value access mode. In this case, the effective parameter is copied at the task creation time and any modification remains local to the function.

Any C or C++ type can be defined to be passed by value. The syntax is the following:

```
#pragma kaapi task value( <list declaration> )
```

The rule `<list declaration>` is given in the appendix B and corresponds to a list of memory regions separated by '`,`'.

4.4.3 Passing rule by reference

Any memory region description in X-KAAPI must be defined by a C/C++ pointer type formal parameter. At runtime, the pointer, the declaration of the memory region and the access mode (read, write, ...) made by the task allow to detect dependencies.

The syntax of the clause is:

```
#pragma kaapi task <mode> ( <list declaration> )
#pragma kaapi task <mode_cw> ( operator: <list declaration> )
```

The previous example accumulate defines 3 arguments. 2 of them are passed by reference: `array` is an array of size `n` (known at runtime); `result` is pointer to one integer.

The access mode name `mode` of the clause is given by table 1 page 20:

write: the task can write to any location of the declared memory region. Any read instruction is guarantied to return the value produced by the task. A read instruction to a location not written by the task returns an undefined value.

read: the task that reads a location in the memory region declared `read` accesses the value produced by the previous task in the lexicographic order of tasks in the program, that required a write or read-write or cumulative access. If no such task existed, then the value is the value given during initialization.

The task cannot write location of the memory region.

readwrite: the task can read or write to any location of the memory region. As for `read` access mode, a read instruction returns the value previously written by the task or the value produced by a previous task or at initialization.

reduction: this is the subject of the next section.

4.4.4 Cumulative access mode

In X-KAAPI, it is possible to reduce variable with associative operator. The commutativity is not required. A task that requires an accumulative semantic with respect to one of its formal parameters, must declare it with the `reduction`

clause in the `#pragma kaapi task`. The reduction clause specifies the operator to use during concurrent reduction and a list of formal parameters the clause is applied on.

Two tasks having a reduction access mode to the same memory region are concurrent. The two tasks created at line 12 and 13 in listing of figure 12 illustrate this case. The reduction operator defined in the `reduction` clause is

```
1  #pragma kaapi task read([begin:end]) reduction(+: result)
2  void accumulate (double* begin, double* end, int *result)
3  {
4    size_t size = (end-begin);
5    if (size < THRESHOLD)
6    {
7      while (begin != end)
8        *result += *begin++;
9    }
10   else {
11     size_t med = size/2;
12     accumulate( begin, begin+med, result);
13     accumulate( begin + med, end, result);
14   }
```

Figure 12: Kaapi signature for accumulate with reduction

used to merge the result produced by a task theft. In listing 12, if one thread performs all the tasks, then only the operator `+=` at line 8 is used. If one of the task created at line 12 or 13 is theft, then the thief thread performs the task and reduce results on a new variable. This new variable is initialized as described below.

Predefined reduction operators.
For all the scalar types of the C or C++ language, X-KAAPI predefines a list of known operators (table 2). For each operator, the neutral element corresponds to the neutral element of the mathematical operator. The list which is presented in table 2 follows the same predefined reduction operators as in OpenMP-3.0.

	name	neutral
+	addition	0
-	subtraction	0
*	multiplication	1
&	bitwise and	~0
|	bitwise or	0
^	bitwise xor	0
&&	logical and	1
||	logical or	0

Table 2: Predefined cumulative operators.

User defined reduction operator.
The user has the ability to define its own reduction operator using `#pragma kaapi declare reduction`:

```
#pragma kaapi declare reduction
     (reduction-identifier : function-name)
     [ identity( function-name) ]
```

The `function-name` must correspond to the name of a C/C++ function that must satisfy one of the following signature:

```
// For  C++
void function( typename& result, const typename& value )
// For C or C++
void function( typename* result, const typename* value )
```

The parameter type are inherited from the declaration of the reduction variable in the `reduction` clause to describe access mode of formal parameter.

The identity function is used to initialize a value of a temporary variable to be the neutral of the reduction operator. The signature of the identity function must be one of the following:

```
// For  C++
void function( typename& value )
// For C or C++
void function( typename* value )
```

4.4.5 Dependencies between tasks

Once task access modes are defined, the runtime is able to detect dependencies between tasks that access to a same memory region.

Read after Write. Also called true dependency. A task is reading a memory region produced by another task.

Write after Write. False dependency. Such dependency is suppressed using variable renaming: the two tasks can be executed concurrently if they work on distinct memory region, thus the runtime may allocate distinct memory regions.

Write after Read. This false dependency is an anti-dependency. Variable renaming suppress them.

Note that the X-Kaapi runtime used a lazy approach to remove false dependencies: if it is necessary to concurrently execute two tasks with false dependencies, then the runtime renames the memory region. Else, the execution is sequential, using the task creation order.

In both cases, the result of the execution remains the same whatever the execution order between tasks is.

4.5 Directive for synchronization

Because task creations are non-blocking instructions, it may be necessary to explicitly synchronize the control flow of the thread that creates tasks with the task executions.

4.5.1 #pragma kaapi sync

This directive forces to wait for the completion of all previously created tasks in the same activation of a task body: tasks created by the parent task are not considered.

For non-recursive task creations, this directive waits until all previously created tasks in the control flow are completed.

For instance, the synchronization point at line 13 of the recursive computation of the Fibonacci number in listing 4 page 14 enforces completion of the two tasks created at lines 11 and 12 only.

4.5.2 #pragma kaapi sync(<list_declaration>)

This directive forces to wait for the completion of all the last created tasks, in the same task body activation, which writes to the memory region described by a the list of declarations.

```
1  int main()
2  {
3    int subresult1;
4    int subresult2;
5    compute1( &subresult1, ...);
6    compute2( &subresult2, ...);
7
8  #pragma kaapi sync(subresult1)
9    printf("Subresult is: %i\n", subresult1 );
10
11   return 0;
12 }
```

Figure 13: Section of variable to enforce synchronization.

This directive is a finer version of `#pragma kaapi sync`.

4.6 Recursive function call

X-KAAPI allows for recursive task creation. Several examples have been presented in section 3. The accumulate code of figure 12 (page 24) recursively splits the computation until a grain of 2. Created sub-tasks work on disjoint part of the initial input array.

This example is simple: no temporary data is required at each level of recursion. This is not the case with the recursive computation of the Fibonacci number in figure 4 page 14. As discussed in section 3.3.1, the user should ensure enough scope of data allocated during the computation.

4.6.1 Control of the allocation scope of data

All data passed by reference to tasks must be valid until the completion of the tasks. For instance, because a task creation is a non-blocking call, the example of figure 14 is incorrect: the automatic variable `r` will be suppressed before the execution of task `g`.

In that case, the user has two possibilities:

```
1 if (...)
2 {
3   int r; /* store sub result */
4   g(&r, input_values); /* create task*/
5 }
```

Figure 14: Incorrect code

- Add a synchronization point between line 4 and 5. It enforces the execution to wait for the completion of `g` (the 2 solutions at the top of figure 15).
- Allocate the data in the same scope as the tasks. In that case, the data is allocated with a scope wide enough to execute the task `g` (bottom solution in figure 15).

```
1 if (...)
2 {
3   int r;
4   g(&r, input_values);
5   #pragma kaapi sync
6 }
```

```
1 if (...)
2 {
3   #pragma kaapi parallel
4   {
5     int r;
6     g(&r, input_values);
7   }
8 }
```

```
1 if (...)
2 {
3 #pragma kaapi data  alloca(r)
4   int r;
5   g(&r, input_values);
6 }
```

Figure 15: Solutions for incorrect code of figure 14

The syntax of the `#pragma kaapi data alloca` is:

`#pragma kaapi data alloca (<list of variables>)`
: The `<list of variables>` describes a list of automatic variables to be used in task creations.

The X-KAAPI name of this directive is analog to the `alloca` function of the stdlib C library. With sequential C or C++ program, the developper may allocate data in the stack using the `alloca` function. In that case, the allocation scope is the same as the activation frame where the call to `alloca` occured. With `#pragma kaapi alloca`, the user can allocate data within the same scope as the created tasks.

4.6.2 Synchronisation and recursive tasks

In a recursive computation, the `#pragma kaapi sync` directive inside a task only synchronizes the child tasks spawned by the task instance. Tasks spawned

by an ancestor tasks are note synchronized[10].

[10]Cilk+ has the same semantic for its sync keyword.

5 Examples

5.1 Fibonacci computation

5.1.1 With sync

```
1  #include <stdio.h>
2
3  #pragma kaapi task write(result) value(n)
4  void fibonacci(long* result, const long n)
5  {
6    if (n<2)
7      *result = n;
8    else
9    {
10     long r1,r2;
11     fibonacci( &r1, n-1 );
12     fibonacci( &r2, n-2 );
13 #pragma kaapi sync
14     *result = r1 + r2;
15   }
16 }
17
18 #pragma kaapi task read(result)
19 void print_result( const long* result )
20 {
21   printf("Fibonacci(30)=%li\n", *result);
22 }
23
24 int main()
25 {
26   long result;
27 #pragma kaapi parallel
28   {
29     fibonacci(&result, 30);
30     print_result(&result);
31   }
32   return 0;
33 }
```

5.1.2 Without sync

```
1  #include <stdio.h>
2
3  #pragma kaapi task write(result) read(r1,r2)
4  void sum( long* result, const long* r1, const long* r2)
5  {
6    *result = *r1 + *r2;
7  }
8
9  #pragma kaapi task write(result) value(n)
10 void fibonacci(long* result, const long n)
11 {
12   if (n<2)
13     *result = n;
14   else
15   {
16 #pragma kaapi data alloca(r1,r2)
17     long r1,r2;
18     fibonacci( &r1, n-1 );
```

```
19      fibonacci( &r2, n-2 );
20      sum( result, &r1, &r2);
21    }
22  }
23
24  #pragma kaapi task read(result)
25  void print_result( const long* result )
26  {
27    printf("Fibonacci(30)=%li\n", *result);
28  }
29
30  int main()
31  {
32    long result;
33  #pragma kaapi parallel
34    {
35      fibonacci(&result, 30);
36      print_result(&result);
37    }
38    return 0;
39  }
```

5.1.3 With reduction variable

```
1  #include <stdio.h>
2
3  #pragma kaapi task reduction(+:result) value(n)
4  void fibonacci(long* result, const long n)
5  {
6    if (n<2)
7      *result += n;
8    else
9    {
10     fibonacci( result, n-1 );
11     fibonacci( result, n-2 );
12   }
13 }
14
15 #pragma kaapi task read(result)
16 void print_result( const long* result )
17 {
18   printf("Fibonacci(30)=%li\n", *result);
19 }
20
21 int main()
22 {
23   long result;
24 #pragma kaapi parallel
25   {
26     fibonacci(&result, 30);
27     print_result(&result);
28   }
29   return 0;
30 }
```

5.2 Cholesky factorization

This example shows that one of the main difference with respect to the StartSs programming model is its ability to view sub-matrix of matrix without requiring

to reallocate the user data structure.

```
#include <cblas.h>
#include <clapack.h>


#pragma kaapi task value(Order, Uplo, N, lda) \
                   readwrite(A{ld=lda; [N][N]})
int clapack_dpotrf(const enum ATLAS_ORDER Order,
                   const enum ATLAS_UPLO Uplo,
                   const int N, double *A, const int lda);


#pragma kaapi task value(Order,Side,Uplo,TransA,Diag,M,N)\
                   value(alpha,lda,ldb) \
                   read(A{ld=lda; [N][N]}) \
                   readwrite(B{ld=ldb; [N][N]})
void cblas_dtrsm(const enum CBLAS_ORDER Order,
                 const enum CBLAS_SIDE Side,
                 const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_DIAG Diag,
                 const int M, const int N,
                 const double alpha,
                 const double *A, const int lda,
                 double *B, const int ldb);

#pragma kaapi task value(Order,Uplo,Trans,N,K,alpha,lda,beta,ldc) \
                   read(A{ld=lda; [N][N]}) \
                   readwrite(C{ld=ldc; [N][N]})
void cblas_dsyrk(const enum CBLAS_ORDER Order,
                 const enum CBLAS_UPLO Uplo,
                 const enum CBLAS_TRANSPOSE Trans,
                 const int N, const int K,
                 const double alpha,
                 const double *A, const int lda,
                 const double beta,
                 double *C, const int ldc);

#pragma kaapi task value(Order,TransA,TransB,M,N,K)\
                   value(alpha,lda,ldb,beta,ldc) \
                   read(A{ld=lda; [M][K]}, B{ld=ldb; [K][N]}) \
                   readwrite(C{ld=ldc; [M][N]})
void cblas_dgemm(const enum CBLAS_ORDER Order,
                 const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB,
                 const int M, const int N, const int K,
                 const double alpha,
                 const double *A, const int lda,
                 const double *B, const int ldb,
                 const double beta,
                 double *C, const int ldc);


// Block Cholesky factorization A <- L * L\^t
// Lower triangular matrix, with the diagonal,
// Stores the Cholesky factor.
void Cholesky( double* A, int N, size_t blocsize )
{
  for (size_t k=0; k < N; k += blocsize)
  {
    clapack_dpotrf(
```

```
61        CblasRowMajor, CblasLower, blocsize, &A[k*N+k], N
62      );
63
64      for (size_t m=k+blocsize; m < N; m += blocsize)
65      {
66        cblas_dtrsm
67        (
68          CblasRowMajor,CblasLeft,CblasLower,CblasNoTrans,CblasUnit,
69          blocsize, blocsize, 1., &A[k*N+k], N, &A[m*N+k], N
70        );
71      }
72
73      for (size_t m=k+blocsize; m < N; m += blocsize)
74      {
75        cblas_dsyrk
76        (
77          CblasRowMajor, CblasLower, CblasNoTrans,
78          blocsize, blocsize, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N
79        );
80        for (size_t n=k+blocsize; n < m; n += blocsize)
81        {
82          cblas_dgemm
83          (
84            CblasRowMajor, CblasNoTrans, CblasTrans,
85            blocsize, blocsize, blocsize, -1.0,
86            &A[m*N+k], N,
87            &A[n*N+k], N,
88            1.0,
89            &A[m*N+n], N
90          );
91        }
92      }
93    }
94  #pragma kaapi sync
95  }
96
97
98  /* Main of the program
99  */
100 void main( int argc, char** argv )
101 {
102   // matrix dimension
103   int N = 32;
104   if (argc > 1)
105     N = atoi(argv[1]);
106
107   // block count
108   int block_count = 2;
109   if (argc > 2)
110     block_count = atoi(argv[2]);
111
112   size_t blocsize = N / block_count;
113
114   double t0, t1;
115   double* A = 0;
116   if (0 != posix_memalign((void**)&A, 4096, N*N*sizeof(double)))
117   {
118     printf("Cannot allocate matrice A, errno: %i\n", errno);
119     return;
120   }
121
122   // Generate matrix A
```

```
123   generate_matrix(A, N);
124
125   // Cholesky factorization of A
126   Cholesky(A, N, blocsize);
127
128   free(A);
129
130   return 0;
131 }
```

5.3 2D Histogram

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* define the hist level too. assume <= 32. */
5  #define CONFIG_DATA_LOG2 13
6  /* dont touch */
7  #define CONFIG_DATA_DIM (1 << CONFIG_DATA_LOG2)
8  #define CONFIG_BLOCK_DIM (CONFIG_DATA_DIM / 8)
9
10
11 /* histogram reduction */
12 #pragma kaapi declare                          \
13   reduction(hist_redop: reduce_hist)           \
14   identity(init_hist)
15
16 static void init_hist(unsigned int* p)
17 {
18   *p = 0;
19 }
20
21 __attribute__((unused))
22 static void reduce_hist(unsigned int* lhs, const unsigned int* rhs)
23 {
24   *lhs += *rhs;
25 }
26
27 #pragma kaapi task                             \
28   value(bdim, hdim, lda)                       \
29   read(block{ld = lda; [bdim][bdim]})          \
30   reduction(hist_redop: hist[hdim])
31 static void compute_block_hist
32 (
33  const unsigned int* block,
34  unsigned int bdim, /* block dimension */
35  unsigned int lda,
36  unsigned int* hist,
37  unsigned int hdim /* histogram dimension */
38 )
39 {
40   unsigned int x, y;
41   for (y = 0; y < bdim; ++y, block += lda)
42     for (x = 0; x < bdim; ++x, ++block)
43       ++hist[*block];
44 }
45
46 static void compute_hist
47 (
48  const unsigned int* block,
```

```
49  unsigned int dim,
50  unsigned int* hist
51 )
52 {
53   static const unsigned int lda = CONFIG_DATA_DIM;
54   static const unsigned int block_size =
55     CONFIG_BLOCK_DIM * CONFIG_BLOCK_DIM;
56
57   const unsigned int block_count = dim / CONFIG_BLOCK_DIM;
58   unsigned int i, j;
59
60   for (i = 0; i < CONFIG_DATA_DIM; ++i)
61     init_hist(hist + i);
62
63   for (i = 0; i < block_count; ++i)
64   {
65     for (j = 0; j < block_count; ++j)
66     {
67       /* compute start of block */
68       const unsigned int* const p =
69         block + i * block_size + j * CONFIG_BLOCK_DIM;
70       compute_block_hist
71         (p, CONFIG_BLOCK_DIM, lda, hist, CONFIG_DATA_DIM);
72     }
73   }
74 }
75
76
77 /* data helpers */
78 static unsigned int* gen_data(unsigned int dim)
79 {
80   const unsigned int size = dim * dim;
81   unsigned int* const block = malloc(size * sizeof(unsigned int));
82
83   /* arrange so that each value appears dim times */
84   unsigned int i;
85   for (i = 0; i < size; ++i) block[i] = i & (dim - 1);
86   return block;
87 }
88
89 static int check_hist(const unsigned int* hist)
90 {
91   unsigned int i;
92   for (i = 0; i < CONFIG_DATA_DIM; ++i)
93     if (hist[i] != CONFIG_DATA_DIM) return -1;
94   return 0;
95 }
96
97
98 /* main */
99 int main(int ac, char** av)
100 {
101   static unsigned int hist[CONFIG_DATA_DIM];
102   unsigned int* const block = gen_data(CONFIG_DATA_DIM);
103   int err;
104
105   if (block == NULL) return -1;
106
107 #pragma kaapi parallel
108   compute_hist(block, CONFIG_DATA_DIM, hist);
109
110   free(block);
```

```
111
112   err = check_hist(hist);
113   if (err == -1) printf("invalid\n");
114
115   return err;
116 }
```

A C and C++ support

The support of C and C++ relies on the version of the EDG frontend used internally by the ROSE framework. C is seems to entirely supported[11]. C++ program is not well recognized by the EDG frontend[12].

For recognized C++ program, we extend the capacity to declare tasks for method invocations. The object pointed by `this` is implicitly passed with read-write mode, except for `const` method.

[11] See ROSE/EDG documentation.

[12] We have problems to recognized complex C++ program that mix template and specialization.

B Grammars

In the following grammar, we assume terminal symbols:

```
identifier := [a-zA-Z_][a-zA-Z_0-9]*
integral := [0-9]+
```

B.1 Grammar for description of task's parameters

The following grammar describes the rules to define access mode for each parameter in the `#pragma kaapi task` compiler directive.

```
list_param :=
    access_param
  | access_param list_param

access_param :=
    mode '(' list_declaration ')
 |  mode_cw '(' redop ':' list_declaration ')

list_declaration :=
    range_declaration
  | list_declaration ',' range_declaration

range_declaration :=
    identifier
  | identifier dimension
  | identifier '{' complex_view '}
  | '[' identifier ':' identifier ']'
  | '[' identifier ':' identifier ')'

complex_view :=
    element_view
  | complex_dimension ',' element_view

element_view :=
    'storage' '=' storage_class
  | 'ld' '=' identifier
  | dimension

mode :=
  | mode_w | mode_r | mode_x | mode_cw
mode_w :=
    'write'     | 'w'  | 'output'
mode_r :=
    'read'      | 'r'  | 'input'
mode_x :=
    'exclusive' | 'x'  | 'inout'
mode_cw :=
    'reduction' | 'cw'
mode_v :=
```

```
    'value'     | 'v'

redop :=
    '+' | '-' | '*' | '&' | '|' | '^' | '&&' | '||'
  | identifier

storage_class :=
    'rowmajor'    | 'C' | 'C++'
  | 'columnmajor' | 'Fortran'
```

B.1.1 Note on access mode

Most of the different access modes (`mode_xx :=` rules) have three formats: long, short and a format compatible with the SMPSs programming model.

If the reduction operator for `reduction` access mode is left unspecified, they the default operator is + for all the scalar type (integral value and floating point value). For other instances of type or class that do not have the + operator defined, this is an compiler error. For scalar values (integral value and floating value), the default identity operator is to set the value '0' which is the neutral element with respect to the associative default + operator. For other type or class which have a defined + operator, then the neutral is assumed to be set by calling the default constructor.

B.1.2 Note on range declaration

In the range declaration rule, the programmer has the opportunity to define the set of memory address that the task will access. The set of memory address may have any shape. This set serves to detect dependencies between tasks. The implementation has currently 2 limits:

1. The grammar limits the shape to continuous D dimensional arrays, with an implementation limit to 2D data structures (see next section about the grammar).

2. The detection of dependency between tasks uses a special memory address of the range declaration which is named the **referent**. This address is the address of the identifier in case of the D-dimensional format. And for 1D definition using the format `[identifier .. identifier]` this is the memory address given by the value of the first identifier (a pointer).

The grammar may be extended in the future in order to take into account more complex shape declarations.

B.1.3 Dimension definition

The dimension in range specification is described by the following grammar. Expression rules in the grammar are a subset of the C expression rules.

```
dimension :=
    one_dimension
  | one_dimension dimension
```

```
one_dimension :=
    '[' expression ']'

expression :=
  additive_expression

additive_expression :=
    multiplicative_expression
  | additive_expression '+' multiplicative_expression
  | additive_expression '-' multiplicative_expression

multiplicative_expression
    cast_expression
  | multiplicative_expression '*' cast_expression
  | multiplicative_expression '/' cast_expression
  | multiplicative_expression '%' cast_expression

cast_expression
    unary_expression
  | '(' type_name ')' cast_expression

unary_expression
    primary_expression
  | SIZEOF unary_expression
  | SIZEOF '(' type_name ')'

primary_expression
    identifier
  | integral
  | '(' expression ')'
```

B.2 Grammar for reduction operator

The compiler directive `#pragma kaapi declare <reduction_declaration>` follows the next grammar. This grammar may be extended to match the OpenMP directive to declare reduction operators using pseudo variables.

```
reduction_declaration :=
    reduction_definition
  | reduction_definition identity_definition

 reduction_definition :=
  | 'reduction' '(' identifier ':'  function_name ')'

 identity_definition :=
    'identity'  '(' function_name  ')'
  | 'identity'  '(' '{' initializer-list '}' ')'
```

The role of the identity declaration is to specify the initialization of a variable with the value that corresponds to the neutral element with respect to the reduction operator. For instance: `0` for operator `+` over scalar type. For user

defined types, this function should be defined. In C++, if the identity function is not defined, the C++ empty constructor is used to initialize the value. In C, the default identity function sets to zero the data structure.

The signature of the reduction function must be:

```
// For C or C++
void function_name( type_name* result, const type_name* value );

// For C++
void function_name( type_name& result, const type_name& value );
```

The signature for the identity function follows the same rule:

```
// For C or C++
void identity_name( type_name* value );

// For C++
void identity_name( type_name& value );
```

B.3 Grammar for `data alloca` directive

The following grammar is involved in the directive

```
#pragma kaapi data alloca <list of variables>

      list_of_variables :=
          identifier
        | list_of_variables ',' identifier
```

This directive must be defined in a basic block before the definition of automatic variables. This is a compiler error to pass a non-automatic name (*e.g.* global variable) or to declare the pragma after the definition of the variables.

C KaCC compiler and runtime options

C.1 Compilation process with KaCC

The compilation process with KaCC is the following:

1. Each X-KAAPI directive that begins with `#pragma kaapi` is interpreted and the code is rewritten to call functions of the X-KAAPI library. Invalid use of X-KAAPI directive and some warning are outputed.
2. The backend compiler (C or C++) is called on the rewritten code to generate an object file.
3. To create an executable, KaCC passes the X-KAAPI library to the linker.

C.2 KaCC options

The KaCC compiler recognizes all the options of the GCC compiler suite (gcc, g++) and add an extra option:

--keep : keep the intermediate files. For each file, one intermediate file is generated. It is prefixed with `kaapi-` and is the translation of the file with replacement of function calls by task creations.

C.3 Backend compiler selection

The backend compiler is the default C or C++ compiler defined during the installation step (see options with `configure -help`). It can be changed by setting the environment variables `CC` and `CXX`.

C.4 Runtime environment variables

KAAPI_CPUSET : if defined, and if the operating system supports it, then this variable specifies the set of cores to use in X-KAAPI programs. If not defined, the operating system decides to schedule X-KAAPI threads onto cores. The syntax is a list of core identifiers or a range of core identifiers:

```
KAAPI_CPUSET=1,2,6,8,21
KAAPI_CPUSET=0:11,20:31
```

The identifier corresponds to system identifier for the core.

KAAPI_CPUCOUNT : if defined, the variable specifies the number of cores to use. This number should be less than the number of cores specified with `KAAPI_CPUSET`.

D SMPSs task model compatibility

A SMPSs parallel program running on a multicore is composed of several threads sharing the same address space. The threads execute tasks which may share values in the address space, that we call thread shared memory. A task is a function call: a function and the list of its effective parameters. In SMPSs [1] the definition of task is based on the annotation of the C function as in X-KAAPI. Figure 16 illustrates the methodology. The original sequential code `accumulate` is annotated to be a task using SMPSs annotation (`#pragma css task`). Each formal parameter is described by the access the task performs: `input` means that the task requires it to be ready before execution; `output` means that the task will produce it. And `inout` means both. As depicted in the code, a formal parameter may be defined with a size in order to identify the memory region accessed by the task.

```
#pragma css task input(n, array[n]) output(result)
void accumulate (int n, int* array, int *result)
{
  int sum = 0;
  for (int i=0; i<n; ++i)
    sum = sum + array[i];
  *result = sum;
}
```

Figure 16: SMPSs Accumulate code

All tasks are asynchronous as in X-KAAPI.

The SMPSs programming model must be initialized before (`#pragma css init` / `#pragama css finish`), and allows reduction with locking mechanisms [1].

D.1 Compatibility

The X-KAAPI compiler parses and executes correctly all the SMPSs directives without any change in the source code.

D.2 Incompatibility

Moreover, due to its ability to define memory region of submatrices, X-KAAPI can be used where SMPSs requires source code modification (for instance in all inplace SMPSs factorization algorithms -LU, Cholesky, QR, ...-) where data (submatrix) must be reallocated into contiguous memory region.

The semantic of reduction variable in X-KAAPI allows to avoid locking mechanisms at the expense of a memory copy.

References

[1] SMP superscalar user's manual v.2.3 (2010), http://www.bsc.es/plantillaH.php?cat_id=391

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

[3] Arvind, Culler, D.E.: Dataflow architectures. Annual Review of Computer Science 1(1), 225–253 (1986), http://www.annualreviews.org/doi/abs/10.1146/annurev.cs.01.060186.001301

[4] 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 (December 2009), http://portal.acm.org/citation.cfm?id=1656506.1656514

[5] Besseron, X., Gautier, T.: Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications. In: MCO'08 (2008)

[6] Blumofe, R.D., Leiserson, C.E.: Space-efficient scheduling of multithreaded computations. SIAM J. Comput. 27, 202–229 (February 1998), http://portal.acm.org/citation.cfm?id=276234.276250

[7] Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing 35(1), 38 – 53 (2009), http://www.sciencedirect.com/science/article/pii/S0167819108001117

[8] 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. pp. 212–223. PLDI '98, ACM, New York, NY, USA (1998), http://doi.acm.org/10.1145/277650.277725

[9] 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 the 1998 International Conference on Parallel Architectures and Compilation Techniques. pp. 88–. PACT '98, IEEE Computer Society, Washington, DC, USA (1998), http://portal.acm.org/citation.cfm?id=522344.825701

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

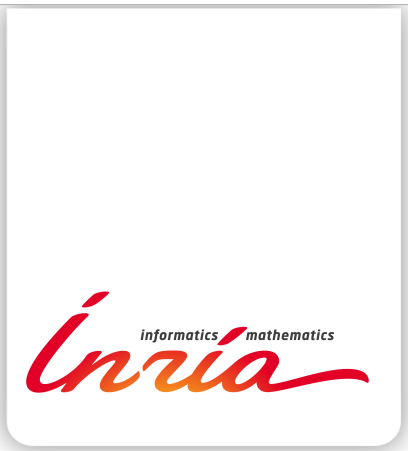
[11] Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software 27(4), 422–455 (Dec 2001), http://doi.acm.org/10.1145/504210.504213

[12] Jafar, S., Krings, A.W., Gautier, T.: Flexible rollback recovery in dynamic heterogeneous grid computing. IEEE Transactions on Dependable and Secure Computing (2008)

[13] Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: CLUSTER. pp. 142–151. IEEE (2008)

[14] Rinard, M.C., Lam, M.S.: The design, implementation, and evaluation of jade. ACM Trans. Program. Lang. Syst. 20, 483–545 (May 1998), `http://doi.acm.org/10.1145/291889.291893`

[15] Traoré, D., Roch, J.L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel STL algorithms. In: EuroPar'08 (2008)



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803