X-Kaapi

Vincent Danjean, Thierry Gautier, Fabien Le Mentec, Jean-Louis Roch

MOAIS project

May 16, 2011

Contents

Foreword

X-Kaapi is developed by the INRIA MOAIS team http://moais.imag.fr. The X-Kaapi project is still under development. We do our best to produce as good documentation and software as possible. Please inform us of any bug, malfunction, question, or comment that may arrise.

This documentation presents the X-Kaapi C, C++ interface for Kaapi. X-Kaapi is a library with several API. The C interface is the lowest interface to program directly on top of the runtime. The C++ interface is an extension of Athapascan-1 interface with new features to avoid explicit declaration of shared variable. X-Kaapi may also be used with C++ through a parallel STL implementation.

Moreover, X-Kaapi has been used as the runtime support for SMPss http://www.bsc.es compiler from BSC, Barcelona.

About X-Kaapi

X-Kaapi is a **"high level"** interface in the sense that no reference is made to the execution support. The synchronization, communication, and scheduling of operations are fully controlled by the software. X-Kaapi is an **explicit parallelism language**: the programmer indicates the parallelism of the algorithm through X-Kaapi's one, easy-to-learn template functions, `Spawn` to create tasks. The programming semantics are similar to those of a sequential execution in that each "read" executed in parallel returns the value it would have returned had the "read" been executed sequentially.

X-Kaapi comes from Athapascan interface defined in 1998 and updated in the INRIA technical report RT-276. It is composed by one runtime and several application programming interface (API). The high level interfaces are: an easy-to-use C++ interface that allows to create tasks with dependencies computed automatically by the runtome; a STL C++ interface for algorithms over sequence; and the use of SMPss programming model. A low level C/C++ interface exports low level functions defined by the runtime. Especially, with the low level interface, it is possible to write very efficient adaptive algorithm. Moreover, X-Kaapi has a compatibility interface for the old Athapascan' programs.

The C++ Kaapi interface as well as the SMPss API provides **a data-flow language**. The program execution is data driven and determined by the availability of the shared data, according to the access made. In other words, a task requesting a read access on shared data will wait until the previous task processing a write operation to this data has ended.

X-Kaapi is **portable and efficient**. The portability is inherited from the use of a subset of POSIX functions which exists on top of most of the up-to-date operating system.

The efficiency with which X-Kaapi program runs has been both tested and theoretically proven. The X-Kaapi programming interface is related to a cost model that enables an easy evaluation of the cost of a program in terms of work (number of operations performed), depth (minimal parallel time) and communication volume (maximum number of accesses to remote data). X-Kaapi has been developed in such a way that one does not have to worry about specific machine architecture or optimization of load balancing between processors. Therefore, it is much simpler (and faster) to use X-Kaapi to write parallel applications than it would be to use with POSIX thread library.

Reading this Document

This document is a user's manual designed to teach one how to install and use the X-Kaapi's APIs. Its main goal is not to explain the way X-Kaapi is built.

If new to X-Kaapi, it is recommened to read all of the remaining text. However, if the goal is to immediately begin writing programs with X-Kaapi, feel free to skip the next two chapters. They simply provide an overview of:

- how to install X-Kaapi's librairies, include files, and scripts (Chapter 1),
- how to test the installation performed (Chapter **??**),
- the C++ tasks API (Chapter 2).
- the SMPss programming model (Chapter 3).
- the low level programming model (Chapter **??**).

The other sections will delve deeper into X-Kaapi' API, so that the user can benefit from all of its functionalities. They explain:

- the concepts of "tasks" and "shared memory" (Chapters **??** and 2.7, respectively);
- how to write the code of desired tasks (Chapter **??**);
- how to make shared types communicable to other processors (Chapter **??**);
- which type of access right to shared memory should be used (Section 2.9);
- how to design parallel programs through complex examples (Chapter **??**);
- how to select the proper scheduling algorithm for specific programs (Appendix **??**);
- how to debug programs using special debugging and visualizing tools (Appendix **??**).

Chapter 1

Quick start

1.1 Overview

1.1.1 X-Kaapi runtime

X-Kaapi is a runtime high performance parallelism targeting multicore and distributed architectures. It relies on workstealing paradigms. The core library comes with a full set of complementary programming interfaces, allowing for different abstraction levels. The following documents the install process, runtime options, as well as a description of APIs lying on top of the runtime and a set of examples.

1.1.2 Supported Platforms

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

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

It has been extensively tested on the following operating systems:

- GNU-Linux/x86_64,
- MacOSX/PowerPC.

There is no version for Windows yet.

1.1.3 X-Kaapi Contacts

If you whish to contact the XKaapi team you can send a mail to `thierry.gautier <dot> inrialpes.fr`, `vincent.danjean <dot> imag.fr`, `fabien.lementec <dot> gmail.com` or `christophe.laferriere <dot>`
Please also visit the www of the research project MOAIS at http://moais.imag.fr.

1.2 Installation

There are 2 ways to install X-Kaapi:

- using the debian packages,
- installing from source.

1.2.1 Using the debian packages

Below is a list of the Debian packages provided for using and programming with X-Kaapi. A brief description is given for each of them:

- xkaapi-doc
  X-Kaapi library documentation.
- libxkaapi0
  X-Kaapi shared libraries.

- libxkaapi-dev
  X-Kaapi development files for the low level C runtime.
- libxkaapi-dbg
  X-Kaapi debug symbols for the above libraries.
- libkaapixx0
  X-Kaapi C++ higher level interfaces standing on top of the X-Kaapi core library.
- libkaapixx-dev
  X-Kaapi C++ interfaces development files.

1.2.2 Installing from sources

Retrieving the 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 project git repository:
  `> git clone git://git.ligforge.imag.fr/git/kaapi/xkaapi.git xkaapi`

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. You can have a complete list of the available options by running:

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

Below is a list of the most important ones:

- `--enable-target=mt`
  Select the target platform. Defaults to 'mt', for pthread.
- `--enable-mode=debug` or `release`
  Choose the compilation mode of the library. Defaults to release.
- `--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/.
- `--prefix=`
  Overload the default installation path.

Example:

```
./configure --enable-mode=release --enable-target=mt --prefix=$HOME/install
```

If there are errors during the configuration process, you have to solve them before going further. It is likely there is a missing dependency on your system, in which case the log gives you the name 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 the runtime is correctly installed on your system:

```
$> make check
```

Compilation of the examples

The following compiles the sample applications:

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

1.3 Programming with X-Kaapi

1.3.1 X-Kaapi integration

Integrating X-Kaapi in your project requires the following steps:

- include the header files in your source code. Example:

  ```
  #include <kaapi.h> /* C version */
  #include <kaapi++> // C++ version
  ```

- add compilation options to your project using pkg-config. Note that if you changed the default install directory during the configuration process, the PKG_CONFIG_PATH environment variable must point to *install_dir/pkgconfig/*. Example:

  ```
  # for C applications
  gcc -o main `pkg-config --flags kaapi` main.c `pkg-config --libs kaapi`
  # for C++ applications
  g++ -o main `pkg-config --flags kaapixx` main.c `pkg-config --libs kaapixx`
  ```

- the following preprocessor macro must be defined to fully disable the debugging code. It can improve the generated code:

  ```
  -DNDEBUG
  ```

Refer to the API documentation for information relative to the X-Kaapi programming interfaces.

1.3.2 Examples

The directory *examples/* contains sample applications using X-Kaapi. Each sub directory contains code and its variations for each example. Both C and C++ are used. Some direclty lies on top of the core runtime, while other make use of higher level interfaces. Below is a short description for some of them:

- fibo_kaapi.c
  Recursive Fibonacci number computation, using data flow graph of tasks. C version.
- fibo_kaapi++.cpp, fibo_kaapi++_opt.cpp
  Same as above, using the *ka* C++ interface.
- fibo_atha.cpp
  Same as above, using the *Athapascan* interface [deprecated interface].
- nqueens_kaapi++.cpp
  Nqueens problem implementation using *ka* C++ interface.
- nqueens_atha.cpp
  Same as above, using the *Athapascan* interface.
- matrix_multiply_kaapi++.cpp
  Matrix multiplication based from a Cilk code. Implemented using the *ka* C++ interface.

1.4 Running X-Kaapi

1.4.1 Runtime environment variables

The runtime behavior can be driven by using the following optionnal environment variables.

- `KAAPI_CPUCOUNT`
  The number of process unit to be used by the runtime. No assumption is made regarding which unit is used. Example:

  ```
  KAAPI_CPUCOUNT=3 ./transform 100000
  ```

- `KAAPI_CPUSET`
  The set of CPU to be used. It consists of a comma separated list of cpu indices, first index starting at 0. By default, and if no KAAPI_CPUCOUNT is given, all the host cpus are used. Example:

  ```
  #use cores 3, 4, 5, 6 and 9 only:
  KAAPI_CPUSET=3,4,5,6,9 ./transform 100000
  ```

  An index range can be used via the ':' token. Example:

  ```
  #same as above using the range syntax:
  KAAPI_CPUSET=3:6,9 ./transform 100000
  ```

  You may exclude a cpu from the set by appending the '!' token. Example:

  ```
  #this will uses cores from 3 to 9, but not 4:
  KAAPI_CPUSET=3:9,!4 ./transform 100000
  ```

- `KAAPI_STACKSIZE`
  Size of the per thread stack, in bytes. By default, this size is set to 64 kilo bytes. Example:

  ```
  KAAPI_STACKSIZE=4096 ./transform 100000
  ```

- `KAAPI_WSSELECT`
  Name of the victim processor selection algorithm to use.
  - "workload": Use a user defined workload to driver the vicitm selection algorithm.
  - any other value: falls back to the default random victim selection algorithm.

  Example:

  ```
  KAAPI_WSSELECT=workload ./transform 100000
  ```

1.4.2 Monitoring performances

If configured by --with-perfcounter, the X-Kaapi library allows to output performance counters.

- `KAAPI_DISPLAY_PERF`
  If defined, then performance counters are displayed at the end of the execution. Example:

  ```
  KAAPI_DISPLAY_PERF=1 ./fibo 30
  ```

- `KAAPI_PERF_PAPIES`
  Assuming X-Kaapi was configured using –with-papi, this variable contains a comma separated list of the PAPI performance counters to use. Both counter symbolic names and numeric hexadecimal constants can be used. More information can be found on the PAPI website (http://icl.cs.utk.edu/papi/). Note that counter list cannot exceed 3 elements. Example:

  ```
  KAAPI_PERF_PAPIES=PAPI_TOT_INS,0x80002230,PAPI_L1_DCM ./fibo 30
  ```

Chapter 2

Kaapi C++ API

This chapter will browse the C++ API of X-Kaapi to create tasks with dependencies.

2.1 Overview

The programmer that want to parallelize its program with X-Kaapi C++ API must first identify the tasks of its program. A task is a procedure call, *i.e* a function call without return value. The creation of a task is a non blocking operation: the callee continues its execution without waiting the execution of the task. The second step is to determine for each task, the way it accesses to the memory through its parameters: if the task read, write or update (read and write) memory referenced by its parameters. Once identified, the execution of the program generates a sequence of tasks.

With these informations (task and access mode), the X-Kaapi runtime is able to detect dependencies between a sequence of tasks. *Read after Write* dependency, or true dependency, corresponds to the case a first task writes a memory region while a second task reads this memory region.

A memory region in X-Kaapi is represented by a pointer and a view object which describes the whole set of addresses accessed from the pointer. A pointer represents a specific memory region. For instance, a programer that want to write a X-Kaapi's task sharing a double floating point number with an other task, may pass a pointer to this floating point.

Read after Write dependencies, also called *true dependencies*, correspond to the data flow of the program: at runtime X-Kaapi is free to schedule tasks in any order that respect the *true dependencies*. Write after write or write after read are called *false dependencies*. X-Kaapi may or may not respect these dependencies[1].

Let us assume, we want to create a task for a C++ procedure `F` that takes in input `d` a double floating point, and that returns $d^2 + 1$. The first step is to specify the signature of the task in the same way a C++ function has a signature.

C++ function signature	X-Kaapi task Signature

`void F(`
`  double n,`
`  double* result`
`);`
	`struct TaskF: public ka::Task<2>::Signature<`
	`  double,         /* input parameter */`
	`  ka::W<double>   /* output parameter */`
	`> {};`

Most of the verbosity of the C++ API is due because X-Kaapi C++ API is a pure C++ library with template expression.

The implementation of the function `F` corresponds to the C++ function definition concept. In X-Kaapi, it is called *task body specialization* that allow to give a specific code for different architectures. Currently X-Kaapi recognizes two architectures: the CPU and GPU architecture. In C++, the X-Kaapi *task body specialization* is a template specialization of the class `TaskBodyCPU` or `TaskBodyGPU`.

[1]May be at the expense of memory allocation in order to rename variable to break such dependencies.

C++ function definition	X-Kaapi task body specialization

`void F (`
`  double n,`
`  double* result`
`)`
`{`
`  *result = n*n+1;`
`}`
	`template<> struct TaskBodyCPU<TaskF> {`
	`  void operator() (`
	`    double n,`
	`    ka::pointer_w<double> result`
	`  )`
	`  {`
	`    *result = n*n+1;`
	`  }`
	`};`

In the same way, the reader can write both the task signature and the task body specialization of a task that to print the result produced by `TaskF`.

C++ function definition	X-Kaapi task body specialization

`void Print (`
`  const double* result`
`)`
`{`
`  std::cout << "The result is :"`
`            << *result << std::endl;`
`}`
	`template<> struct TaskBodyCPU<TaskPrint> {`
	`  void operator() (`
	`    ka::pointer_r<double> result`
	`  )`
	`  {`
	`    std::cout << "The result is :"`
	`              << *result << std::endl;`
	`  }`
	`};`

Now, it is time to write the main program that describes the whole computation. First, it is necessary to initialize the X-Kaapi library with a call to `ka::System::initialize` and to terminate the use of X-Kaapi at the end of the execution.

C++ main	X-Kaapi main

`int main(int argc, char** argv)`
`{`
 	
`  double n = atod(argv[0]);`
`  double result;`
 	
`  F(n, &result);`
`  Print(&result);`
 	
`  return 0;`
`}`
	`int main(int argc, char** argv)`
	`{`
	`  ka::System::initialize( &argc, &argv );`
	
	`  double n = atod(argv[0]);`
	`  double result;`
	
	`  ka::Spawn<TaskF>()(n, &result);`
	`  ka::Spawn<TaskPrint>()(&result);`
	
	`  ka::System::terminate();`
	`  return 0;`
	`}`

On terminaison, the X-Kaapi runtime wait the execution of all previously spawned tasks: it is not necessary to add an extra synchronization point in the previous code in order to wait execution of `TaskF` and `TaskPrint`.

The next sections will present in details the C++ API: task model and the C++ object to program in X-Kaapi.

2.2 Memory region

In order to allow parallel computation within existing application data structure, it is necessary to specify the memory region of a data structure or a sub region of an existing data structure. In X-Kaapi, the API allows to define array based memory region of an existing application data structure. In the current implementation only 1 dimensional (1D) and 2 dimensional (2D) data structures are allowed.

Using a definition of the memory region permits to the runtime to access in concurrence to distinct sub parts of an array. The array memory region is like a simple array with accessor to sub elements, definition of a sub memory region and assignment. In all cases, no copy of array is made.

2.2.1 One dimensional array

2.2.2 Multi dimensional array

2.3 Tasks

The granularity of an algorithm is explicitly given by the programmer through the **creation** of **tasks** that will be **asynchronously** executed. A task is an object representing the association of a procedure and effective parameters. Tasks are dynamically created at run time. A task (creation + execution) in X-Kaapi can be seen as a standard procedure call. The only difference is that the created task's execution is fully asynchronous, meaning the creator is not automatically blocked during the execution of the created task to finish to continue with its own execution. The creator that want to wait until the completion of executed tasks must call explicitly synchronization points.

A task is defined by three components: a name, a signature and its implementation on CPU or GPU.

2.3.1 Task's Signature Definition

A task must have a *signature* which specify the type and access mode the task mades to its effective parameters.

```
 struct UserTaskName : public ka::Task<N>::Signature<
           [...types and access modes...]
  >
  {};
```

The number of effective parameters must be specify: here it is `N`. The types and access modes is a list of specifications of each formal parameter. The specification is either:

- A C++ type such as `int` or `double`, or any user defined type.
- A typed access mode that specify:
  - an access mode:
    - `ka::W<T>` for write access mode, meaning that the task will write a new value of type T.
    - `ka::R<T>` for read access mode, meaning that the task will only read the previous value of type T, without possibility to modify the data.
    - `ka::RW<T>` for exclusive access mode, meaning that the task will read or write value of type T.
    - `ka::CW<T>` for concurrent write access mode, meaning that the task will write value of type T with accumulation law.
  - a C++ type that should be defined between < and > in the previous access mode.

Let us assume we want to define a task that will print a result (double floating point value) within a string. The result will be produced by an other task. The task has two parameters: one for the string and one for the result. The string is passed by value and it is not necessary to defined an access mode.

```
struct TaskPrint : public ka::Task<2>::Signature<
      std::string,         // type of the first formal parameter
      ka::R<double>        // the second parameter is read. type is a double.
   >
{};
```

Once defined, the task signature can be used to implement the body of the task. This step is called the *task body definition*.

2.3.2 Task's Body Definition

A task corresponds to the execution of a C++ function object, *i.e.* an object from a class having the `void operator()` defined, specialized for a given architecture. Currently, X-Kaapi only defines two architectures: CPU and GPU. Thanks to the task signature definition, the programmer must specializes one of the CPU or GPU body (procedure).

This specialization for a given architecture correspond to a template specialization of the `TaskBodyCPU` or (not excluive) `TaskBodyGPU` classes. The template classes must be specialized with the `UserTaskName`, *i.e.* the name of the class that has been used to defined the signature.

The types and the number of parameters must match the task' signature. For instance:

```
template<>
struct TaskBodyCPU<TaskPrint>
{
  void operator() ( std::string msg, double d )
  {
    std::cout << msg << d << std::endl;
  }
};
```

The type of the formal parameters must match the type defined in the signature of the tasks. Else, during the compilation, the C++ compiler will reports[2] an error.

Formal parameters in task's body definition

The figure 2.2 resumes the conversion rules between task' signature definition and the task's boy definition. The

type of *signature parameter*	required type for the *body definition's parameter*
`T`	`T`
`T`	`const T&`
`ka::R[P] < T >`	`ka::pointer_r[p] < T >`
`ka::W[P] < T >`	`ka::pointer_w[p] < T >`
`ka::RW[P] < T >`	`ka::pointer_r[p]w[p] < T >`
`ka::CW[P] < T >`	`ka::pointer_cw[p] < T >`
`ka::W < ka::range1d< T > >`	`ka::range1d_w < T >`
`ka::R < ka::range1d< T > >`	`ka::range1d_r < T >`
`ka::RW < ka::range1d< T > >`	`ka::range1d_rw < T >`
`ka::RPWP < ka::range1d< T > >`	`ka::range1d_rpwp < T >`
`ka::W < ka::range2d< T > >`	`ka::range2d_w < T >`
`ka::R < ka::range2d< T > >`	`ka::range2d_r < T >`
`ka::RW < ka::range2d< T > >`	`ka::range2d_rw < T >`
`ka::RPWP < ka::range2d< T > >`	`ka::range2d_rpwp < T >`

Figure 2.1: Conversion rules between task' signature definition and the definition of the task' body.

section 2.4 will presents the available operators and type members for pointer's types, including the special case for the types `range1d` and `range2d`

2.3.3 Task's Creation

The type of the effective parameters must match the type of the corresponding formal parameters. The precise matching rules are defined in section **??**. For example, a value cannot be used if the formal parameter requires a shared data; a task that declares a write access parameter cannot be access if the effective parameter is declared as a read access mode parameter.

A task is an object of a user-defined type that is instantiated with `Spawn`:

```
    ka::Spawn< UserTaskName > ()  ( [... effective parameters ...] ) ;
      // TaskBodyCPU<UserTaskName>::operator() is executed asynchronously.
      // The synchronizations are defined by the access on
      // a shared the memory region ; the semantic respects the
      // reference order.
```

Example The task `hello_world` displays a message on the standard output:

[2]Note that in that case, the output message may be not easily readable.

```
// task' signature
struct hello_world : ka::Task<0>::Signature { };

// CPU task's body
template<>
struct TaskBodyCPU<hello_world>
{
  void operator() ( )
  {
    cout << "Hello world !" << endl ;
  }
};

int main( int argc, char**argv )
{
  ka::System::initialize(&argc, &argv);

  ka::Spawn< hello_world >() (); // non blocking call

  ka::System::terminate(); // blocking call: wait completion of all created tasks
  return 0;
}
```

2.3.4 Task Execution

The control flow that creates a task does not block until the completion ofthe task. The task execution is ensured by the X-Kaapi system. The following properties are respected:

- The task execution will respect the synchronization constraints due to the shared memory access;
- All the created tasks will be executed once and once only,
- The X-Kaapi system guarantees that every shared data accessed for either reading or updating is available in the local memory before the execution of the task begins.

2.4 Pointer type

A pointer with access mode is restricted to some operators: write access mode allows to write value of the pointed value, while read access mode allows to read the pointed value. In all the case, it is permitted to do arithmetics with pointers of the same mode.

2.4.1 Pointer creation

2.4.2 Arithmetics with pointer

Let us note by `X` the type of the pointer `p` or `q`, and by `v` an integral type. For instance, `X` may be a `ka::pointer<T>` or a `ka::pointer_wp<T>`.

name	expression	type requirements	return type
addition	`p += v`	`v` integral type	`X&`
addition	`p + v`	″ ″	`X`
increment	`p++`	″ ″	`X`
increment	`++p`	″ ″	`X&`
subtraction	`p -= v`	″ ″	`X&`
subtraction	`p - v`	″ ″	`X`
decrement	`p--`	″ ″	`X`
decrement	`--p`	″ ″	`X&`
subtraction	`p-q`	`p,q` of type `X`	`difference_type`

Figure 2.2: List of operators available for X-Kaapi's pointer arithmetics.

2.4.3 Pointer with access rights

ka::pointer_r<T>

ka::pointer_w<T>

ka::pointer_rw<T>

ka::pointer_cw<T>

2.5 Synchronization and memory barrier

Each X-Kaapi task can access three levels of memory:

- the **stack**, a local memory private to the task. This local memory contains the parameters and local variables (it is the classical C or C++ stack).
  This stack is automatically deallocated at the end of the task.
- The **heap**, the local memory of the node (Unix process) that executes the task. Objects are allocated or deallocated in or from the heap directly using C/C++ primitives: `malloc`, `free`, `new`, `delete`...
  Therefore, all tasks executed on a given node share one common heap[3]: consequently, if a task does not properly deallocate the objects located in its heap, then some future tasks may run short of memory on this node.
- The **shared memory**, accessed concurrently by different tasks. The shared memory in X-Kaapi is a non-overlapping collection of physical objects (of communicables types) managed by the system.

2.6 Optimization

2.6.1 X-Kaapi stack allocation

2.6.2 Task's body definition with contextual information

2.7 Hybrid programming with X-Kaapi

[3]In the current implementation, the execution of a task on a node is supported by a set of threads that share the heap of a same heavy process representing the node.

2.8 Shared Memory: *Access Rights* and *Access Modes*

Shared memory is accessed through typed references. One possible type is `Shared`. The consistency associated with the shared memory access is that each "read" sees the last "write" according to the lexicographic order.

Tasks **do not make any side effects** on the shared memory of type `Shared`. Therefore they can only access the shared data on which possess a reference. This reference comes either from the declaration of some shared data or from some effective parameter. A reference to some shared data is an object with the following type: `a1::Shared_RM < T >`. The type `T` of the shared data must be communicable (see Section **??** page **??**). The suffix `RM` indicates the access right on the shared object (read – `r` –, write – `w` – or cumul – `c` –) and the access mode (local or postponed – `p` –) `RM` can be one of the following suffixes:
`r`, `rp`, `w`, `wp`, `cw`, `cwp`, `r_w` and `rp_wp`.
Access rights and modes are respectively described in section 2.9 page 17 and 2.10 page 22.

2.9 Declaration of Shared Objects

If `T` is a communicable type, the declaration of an object of type `a1::Shared<T>` creates a new shared datum (in the shared memory managed by the system) and returns a reference to it.

Depending on whether the shared object is initialized or not, three kinds of declarations are allowed:

- `a1::Shared< T > x( new T( ... ) );`
  The reference `x` is initialized with the value pointed to by the constructor parameter. Note that the memory being pointed to will be deallocated by the system and should not be accessed anymore by the program. `x` can not be accessed by the task that creates it. It is only possible to Fork other tasks with `x` as an effective parameter.
  Example:

```
a1::Shared<int> x ( new int( 3 ) );
  // x is initialized with the value 3.

double* v = new double ( 3.14 );
a1::Shared<double> sv ( v );
  // sv is initialized with the value v;
  // v can not  be used anymore in the program
  // and will be deleted by the system.
```

- `a1::Shared< T > x;`
  The reference `x` is declared but not initialized. Thus, the first task that will be forked with `x` as parameter will have to initialize it, using a write statement (2.9 page 17). Otherwise if a task recieves this reference as a parameter and attempts to read a value from it, the returned value is a value built from the default constructor of the communicable type T.
  Example:

```
a1::Shared<int> a (new int(0) );
  // a is a shared object initialized with the value 0.
a1::Shared<int> x ( 0 );
  // x is a NON initialized shared object.
```

- `a1::Shared< T > x;`
  The reference `x` is only declared as a reference, with no related value. `X` therefore has to be assigned to another shared object before forking a task with `x` as a parameter. Such an assignment is symbolic, having the same semantics as pointer assignment with delegation, *i.e.* the right value of the operand will being a not initialized shared.
  Example:

```
a1::Shared<int> x;
  // x is just a reference, not initialized.
a1::Shared<int> a (new int(0) );
  // a is a shared object initialized with the value 0.
x = a;
  // x points to the same shared object as a.
```

An other declaration with **copy** is provided for convenience:

- `a1::Shared< T > x( src  );`
  The reference `x` is initialized with the value `src` to by the constructor parameter. Note that the memory being references by `src` is copied onto an object allocated into the heap which will be deallocated by the system. The value associated to `x` can not be accessed by the task that creates it. It is only possible to Fork other tasks with `x` as an effective parameter.
  Example:

```
a1::Shared<int> x ( 3 );
  // x is initialized with the value 3 after a copy

std::vector<double> v(120367);
// here initialisation of v
a1::Shared<double> sv ( v );
  // sv is initialized with the value v after a copy

std::vector<double>* pv = new std::vector<double>(120367);
// here initialisation of pv
a1::Shared<double> sv ( pv );
  // sv is initialized with the value pv and take the owner ship of pv
  // pv can not  be used anymore in the program
  // and will be garbaged by the system.
```

The following operations are allowed on an object of type `Shared`:

- Declaration in the stack, as presented above.
- Declaration in the heap, using the operator `new` to create a new shared object. In the current implementation, the "link" between a task and a shared data version is made through the C++ constructor and destructor of the shared object. So, to each construction must correspond a destruction, else dead-lock may occur. Therefore, in the case of allocation in the heap, the `delete` operator corresponding to the already exectured `new` has to be performed.
- Affectation: a shared object can be affected from one to another. This affectation is symbolic, having the same semantics as pointer affectation. The "real" shared object is then accessed through two distinct references.

2.10 Shared Access Rights

In order to respect the sequential consistency (lexicographic order semantic), X-Kaapi has to identify the value related to a shared object for each read performed. Parallelism detection is easily possible in the context that any task specifies the shared data objects that it will access during its execution (on-the-fly detection of independent tasks), and which type of access it will perform on them (on-the-fly detection of a task's precedence). Therefore, an X-Kaapi task can not perform side effects. All manipulated shared data must be declared in the prototype of the task. Moreover, to detect the synchronizations between tasks, according to lexicographic semantic, any shared parameter of a task is tagged in the prototype of `t` according to the access performed by `t` on it. This tag indicates what kind of manipulation the task (and, due to the lexicographic order semantics, all the sub-tasks it will fork) is allowed to perform on the shared object. This tag is called the access right; it appears in the prototype of the task as a suffix of the type of any shared parameter. Four rights can be distinguished and are presented below: read, write, update and accumulation.

2.10.1 Read Right: `Shared_r`

`a1::Shared_r< T >` is the type of a parameter thats value can only be read. This reading can be concurrent with other tasks referencing this shared object in the same mode.

In the prototype of a task, the related type is:
`a1::Shared_r< T > x`
Such an object gets the method:
`const T& read () const;`
that returns a constant reference to the value related to the shared object `x`.

For example, using the Class `complex` that is defined in **??**:

```
class print {
  void operator() ( a1::Shared_r< complex > z ) {
     cout << z.read().x << " + i." << (z.read()).y;
  }
};
```

2.10.2 Write Right: `Shared_w`

`a1::Shared_w< T >` is the type of a parameter whose value can only be written. This writing can be concurrent with other tasks referencing this shared data in the same mode. The final value is the last one according to the reference order. In the prototype of a task, the related type is:
`a1::Shared_w< T > x`
Such an object gets the method:
`void write ( T* address );`
that assigns the value pointed to by `address` to the shared object.

This method assigns the value pointed to by `address` to the shared object. No copy is made: the data pointed by ¡address¿ must be considered as lost by the programmer. Further access via this address are no more valid (in particular, the deallocation of the pointer: it will be performed by the system itself).

Example:

```
class read_complex {
  void operator() ( a1::Shared_w< complex > z ) {
     complex* a = new complex;
     cin >> a.x >> a.y;
     z.write ( a );
  }
};
```

Note To clarify the rule that each `read` ”sees” the last `write` due to lexicographical order being observed, follow the example below:

```
1. #include <athapascan-1>
2. #include <iostream>
3.
4. struct my_read {
5.   void operator()( a1::Shared_r<int> x ) {
6.     std::cout << "x=" << x.read() << std::endl;
7.  }
8. };
9. struct my_write {
10.  void operator()( a1::Shared_w<int> x, int val ) {
11.    x.write( new int(val) );
12.  }
13.};
14.int doir( int argc, char** argv ) {
15.  a1::Shared<int> i( new int( 1 ) );
16.  a1::Fork<my_write>()( i, 1 );
17.  a1::Fork<my_read>()( i );
18.  a1::Fork<my_write>()( i, 2 );
19.  a1::Fork<my_read>()( i );
20.  a1::Fork<my_write>()( i, 3 );
21.  a1::Fork<my_read>()( i );
22.  return 0;
23.}
```

It is possible that the operations in line `20` and then in line `21` will execute before the preceeding lines because the rule described above is not broken. So do not be surprised to see the following result on the screen:

```
x=3
x=2
x=1
```

Keep this in mind, especially when measuring the time of computations. In that case of adding some extra synchronization variables to the code. But be careful because this can decrease the efficiency with which the program runs.

2.10.3 Update Right: `Shared_r_w`

`a1::Shared_r_w< T >` is the type of a parameter thats value can be updated in place; the related value can be read and/or written. Such an object represents a critical section for the task. This mode is the only one where the address of the physical object related to the shared object is available. It enables the user to call sequential codes working directly with this pointer.

In the prototype of a task, the related type is: `a1::Shared_r_w< T > x`. Such an object gets the method: `T& access ( );` , that returns a reference on the data contained by the shared referenced by `x`. Note that `&(x.access())` is constant during all the execution of the task and can not be changed by the user.

Example:

```
class incr_1  {
  void operator() ( a1::Shared_r_w< int > n ) {
     n.access() = n.access() + 1;
  }
}
```

2.10.4 Accumulation Right: `Shared_cw`

`a1::Shared_c< T >` is the type of a parameter whose value can only be accumulated with respect to the user defined function class `F`. `F` is required to have the prototype:

```
struct cumul_fn {
  void operator() ( T& x, const T& y ) {
    ... // body to perform x <-- accu(x, y)
  }
};
```

Example:

```
struct add {
   void operator () (int& x, int& y) {
      x+=y;
   }
};
```

The resulting value of the concurrent write is an accumulation of all other values written by a call to this function. After the first accumulation operation is executed, the initial value of x becomes either the previous value or remains the current value, depending on the lexicographic access order. If the shared object has not been initialized, then no initial value is considered. Since an accumulation enables a concurrent update of some shared object, the accumulation function `F` is assumed to be both *associative* and *commutative*. Note that only the accumulations performed with respect to a same law F can be concurrent. If different accumulation functions are used on a single shared datum, the sequence of resulting values obeys the lexicographic order semantics.

In the prototype of a task, the related type is: `a1::Shared_cw< F,  T > x` . Such an object gets the method: `void cumul (T& v );` that accumulates (according to the accumulation function `F`) `v` in the shared data referenced by `x`. For the first accumulation a copy of `v` may be taken if the shared data version does not contain some valid data yet.

Example:

```
// A generic function class that performs
// the multiplication of two values.
template < class T >
class multiply {
  void operator( T& x, const T& val ) {
    x = x * y;
  }
};

// A task that multiplies by 2 a shared object
class mult_by_2  { //
  void operator() ( a1::Shared_cw< multiply<int>,  int > x) {
     x.cumul ( new int(2) );
  }
};
```

Note: Keep in mind that a program written in X-Kaapi can benefit at run-time from the associative and communative properties of the accumulation function `F`. It is therefore possible that the execution of the following code:

```
#include <athapascan-1>
#include <iostream>
struct F {
  void operator()( int & x, const int & val ) {
    std::cout << " x=" << x << ", val="  val << std::endl;
    x += val;
  }
};
struct add {
  void operator()( a1::Shared_cw<F,int> x, int val ) {
    x.cumul( val );
  }
};
int doit( int argc, char** argv ) {
  a1::Shared<int> i( new int( 1 ) );
  a1::Fork<add>()( i, 2 );
  a1::Fork<add>()( i, 3 );
  return 0;
}
```

will result in:

```
x=3 val=2
x=5 val=1
```

It may seem as though the program was implemented according to the sequential depth-first algorithm:

```
......
a1::Shared<int> i( new int( 3 ) );
a1::Fork<add>()( i, 2 );
a1::Fork<add>()( i, 1 );
......
```

This is not the case. Naturally the above code is semantically correct as well and could produce the same result as the previous program. It is therefore important to realize that since the function `F` is associative and commutative, the precise manner in which the reductions are performed cannot be predicted, even in the case where initial values are known.

2.11 Shared Access Modes

In order to improve the parallelism of a program when only a reference to a value is required - and not the value itself - X-Kaapi refines its access rights to include *access modes*. An access mode categorizes data by restricting certain types of access to the data. By default the access mode of a shared data object is "immediate", meaning that the task may access the object using any of the `write`, `read`, `access` or `cumul` methods during its execution. An access is said to be "postponed" (access right suffixed by p) if the procedure will not directly perform an access on the shared data, but will instead create other tasks that may access the shared data. In functional languages, such a parallelism appears when handling a reference to a future value.

With this refinement to the access rigths, X-Kaapi is able to decide with greater precision whether or not two procedures have a precedence relation. A procedure requiring a shared parameter with a direct read access, `r`, has a precedence relation with the last procedure to take this same shared parameter with a write access. However, a procedure taking some shared parameter with a postponed read access, `rp` , has no precedence relation. It is guaranteed by the access mode that no access to the data will be made during the execution of this task. The precedence will be delayed to a sub-task created with a type `r` . In essence, the type Shared can be seen as a synonym for the type `a1::Shared_rp_wp<T>`; it denotes a shared datum with a read-write access right, but on which no access can be locally performed. An object of such a data type can thus only be passed as an argument to another procedure.

2.11.1 Conversion Rules

When forking a task `t` with a shared object `x` as an effective parameter, the access right required by the task `t` has to be owned by the caller. More precisely, the Figure 2.3 page 22 enumerates the compatibility, at task creation, between a reference on a shared object type and the formal type required by the task procedure declaration. Note that this is available only for task creation, and not for standard function calls where the C++ standard rules have to be applied.

type of *formal parameter*	required type for the *effective parameter*

`a1::Shared_r[p]    < T >`	`a1::Shared_r[p]       < T >`
	`a1::Shared_rp_wp      < T >`
	`a1::Shared< T >`

`a1::Shared_w[p]    < T >`	`a1::Shared_w[p]       < T >`
	`a1::Shared_rp_wp      < T >`
	`a1::Shared< T >`

`a1::Shared_cw[p]< F,T >`	`a1::Shared_cw[p]  < F,  T >`
	`a1::Shared_rp_wp      < T >`
	`a1::Shared< T >`

`a1::Shared_rp_wp   < T >`	`a1::Shared_rp_wp      < T >`
	`a1::Shared< T >`

`a1::Shared_r_w     < T >`	`a1::Shared_rp_wp      < T >`
	`a1::Shared< T >`

Figure 2.3: Compatibility rules to pass a reference on some shared data as a parameter to a task.

2.11.2 `Shared` Type Summary

Figure 2.4 page 23 summarizes the basic properties of references on shared data.

Reference type	decl.	formal P	effectif P	read	write	cumul	modif	concurrent
`a1::Shared_r < T >`		●	●	●				●
`a1::Shared_rp < T >`		●	●	○				●
`a1::Shared_w < T >`		●	●		●			
`a1::Shared_wp < T >`		●	●		○			
`a1::Shared_cw < F,T >`		●	●			●		●
`a1::Shared_cwp < F,T >`		●	●			○		●
`a1::Shared_r_w < T >`		●		●	●		●	
`a1::Shared_rp_wp < T >`		●	●	○	○		○	
`a1::Shared < T >`	●		●	○	○		○	

Figure 2.4: Figure 6.3: Available types (and possible usages) for references on shared data. A● stands for a direct property and a ○ for a postponed one. *formal P* denotes formal parameters (type given at task declaration) and *effective P* denotes effective ones (type of object given at the task creation). *concurrent* means that more than one task may refer to the same shared data version.

2.12 Example: A Class Embedding X-Kaapi Code

A good way to write X-Kaapi applications is to hide X-Kaapi code in the data structures. Proceeding that way will allow you to keep your main program free from parallel instructions (making it easier to write and understand). We are now going to write a shared data structure on top of the `std::vector` class.

```
#include <athapascan-1>
#include <vector>
/**
        class shared_vector is a class hiding Athapascan code so that
        the main code of the application could be written as if it was
        sequential. It is based upon the std::vector class
*/
// resize the shared vector
template<class T>
struct resize_shared_vector {
  void operator() (a1::Shared_r_w<std::vector<T > > v, unsigned int size)
  { v.access().resize(size); }
};
// assignment
template<class T>
struct assign_shared_vector {
  void operator() (a1::Shared_w<std::vector<T > > dest, const std::vector<T>& src)
  { *dest.access() = src; }
};
// swap two elements of a shared vector
template<class T>
struct swap_shared_vector {
  void operator() (a1::Shared_r_w<std::vector<T > > shv, int i1, int i2) {
    std::vector<T>& v = *shv.access();
    T tmp = v[i1]; v[i1] = v[i2]; v[i2] = tmp;
  }
};
// print the data of a shared vector to standard output
template<class T>
struct ostream_shared_vector {
  void operator() (a1::Shared_r<std::vector<T > > shv) {
    unsigned int size = v.read().size;
    for (int i=0; i<size; i++) std::cout << shv.read().elts[i] << " ";
  }
};

template<class T>
class shared_vector : public Shared<std::vector<T > > {
public:
  //constructors
  shared_vector()
   : Shared<std::vector<T > >(new std::vector<T>()) {}
  shared_vector(unsigned int size)
   : Shared<std::vector<T > >(new std::vector<T>(size)) {}
  void resize(unsigned int size)
  { Fork<resize_shared_vector<T> >() (*this, size); }
  void operator= (const std::vector<T> &a)
  { Fork<assign_shared_vector<T> >() (*this, a ); }
  void swap(int i1, int i2)
  { Fork<swap_shared_vector<T> > () (*this, i1, i2); }
};
// ostream operator
template<class T >
ostream& operator<<( ostream& out, const shared_vector<T>& z ) {
   Fork<ostream_shared_vector<T> > () ((Shared<std::vector<T> >) z);
        return out;
}
```

The following main file tests the shared class. As you can see, there is no more reference to specific parallel code.

```
#include <athapascan-1>
#include <sharedVector.h>

#define SIZE 100

int doit( int argc, char** argv ) {
  shared_vector<int> t1(10), t2(20);
  std::vector<int> v(SIZE);

  //fill the array
  for (int i(0); i< SIZE; i++) {
    v[i] = i;
  }

  // resize the shared vector to test the methods
  t1.resize(SIZE);

  // move the data to the shared vector
  t1 = v;

  // try to swap a data
  t1.swap(2, 27);

  return 0;
}
```

Chapter 3

SMPss with X-Kaapi

This chapter presents how to use the SMPss programming model (http://www.bsc.es) with the X-Kaapi runtime. Within X-Kaapi, SMPss has been extended in the following direction:

recursivity: with version 2.3 of SMPss, it is impossible to exploit natural parallelism from SMPss compilation chain. With X-Kaapi, the original SMPss compiler has been extended to allow recursive programs executed on parallel architecture with X-Kaapi.

affinity: X-Kaapi extended the attribut of task in order to manage better affinity between tasks and data. This is especially important when programs run on NUMA architecture.

3.1 Installation

blabla

3.2 Quick start with SMPss

blabla

3.3 Extension to standard SMPss

blabla

Chapter 4

The Kaapi low level interfaces

Reference Guide: environment variables