KAAPI

for version 2.4.1, 16 June 2010

S. Guelton (serge.guelton@enst-bretagne.fr)

Table of Contents

1 Overview

KAAPI means Kernel for Adaptative, Asynchronous Parallel and Interactive programming. It is a C++ library that allows to execute multithreaded computation with data flow synchronization between threads. The library is able to schedule fine/medium size grain program on distributed machine. The data flow graph is dynamic (unfold at runtime). Target architectures are clusters of SMP machines.

Main features are

- It is based on work-stealing algorithms ;
- It can run on various processors ;
- It can run on various architectures (clusters or grids) ;
- It contains non-blocking and scalable algorithms ;

2 Quick installation guide of KAAPI

In order to use the `Athapascan`'s API, you need to install `KAAPI` library. The following steps will help you to do it.

2.1 Get the source

There are several ways to get `KAAPI` source files:

- You can download the latest stable version available at: `https://gforge.inria.fr/frs/?group_id=94`
- You can also get a tarball of the latest git sources (rebuilt every night): `http://kaapi.gforge.inria.fr/snapshots/kaapi-svn.tar.gz`. Please note that if current git version fails to pass the `make distchek` test, the tarball won't be built (i.e. you will download the latest tarball that pass `make distchek`).
- At least, you can use the anonymous git server if you want to keep updating sources (but sometimes, sources are totally broken here):

```
$ git clone git://git.ligforge.imag.fr/git/kaapi/kaapi.git kaapi-git
```

2.2 Compile the package

The following details the weel known `./configure; make; make install` steps.

2.2.1 Configuring

Depending on how you got `KAAPI` you may need to perform some additionnal actions.

2.2.1.1 For git users only

`KAAPI` uses GNU `autoconf` and `automake` tools to simplify its configuration. For `git` users, they must run the following script in the source directory to generate autoconf/automake/libtool files (make sure that you have these development tools available on your system). This has already been done When you use the first two other ways to get `KAAPI` sources.

NOTE: In the following, replace <*kaapi_src_dir*> by the directory where you have gotten kaapi

```
$ cd <kaapi_src_dir> && ./bootstrap
```

2.2.1.2 For all users

You can check whether a configure file has been created in the source directory.

If you want to use more configure options, please read its documentation:

```
$ ./configure '--help'
```

It is better that KAAPI source files are compiled in a different directory from the source code directory. We suggest you create a directory named build:

```
$ cd ..
$ mkdir build ; cd build
```

In the build directory, you now can launch the configure script, using '`--prefix`'=<*install dir*> option to choose an installation path which will be the root of the installation directory (header files, archive, script,etc. will be put here). This installation directory must be different from the source directory (and the build directory).

NOTE: Source (and build) directory can be removed after the KAAPI library is compiled and installed. The installation directory must be kept to be able to compile and run KAAPI applications

```
$ ../<kaapi_src_dir>/configure --prefix=$HOME/KAAPI
```

2.2.2 Compiling

Concerning Compilers : we ensure that KAAPI can be well compiled with the following compilers:

- `g++-3.4` (deprecated)
- `g++-4.1` to `g++-4.3`

WARNING: On Itanium, version 4 or greater is required, older version (especially 3.4) are not supported

To compile the library, just run `make` in the build directory:

```
$ make
```

It may take some time. You can also read make documentation to have more compiling options.

2.2.3 Installing

If the previous steps succeed, then the installation of the library is simple. (Note: depending on your installation path specified by the prefix option during the previous configuration, you may need root privileges).

```
$ make install
```

Congratulation, you can now use `Athapascan` and `KAAPI` !

3 Compile and Run Athapascan programs

This chapter details the basic steps needed to compile a source code linked with `KAAPI`, and how to run it.

3.1 Compiling Athapascan program

Once `KAAPI` is installed (see Chapter 2 [install guide], page 2), there are two ways to compile an `Athapascan` program. You can either use the help of the pkg-config program if you have it installed or you can use environment variables.

3.1.1 Setting up an environment to compile Athapascan programs

If you use environment variables or if KAAPI is not installed in system directories (usually if you did not use the "–prefix=/usr" with the configure script), you need to setup your environment.

You can either start a new shell

```
$ $HOME/kaapi/bin/kaapish --mode=devel --flags
$ # replace $HOME/kaapi by the prefix used with the configure script
```

or you can add environment variables in the current shell

```
$ eval ‘$HOME/kaapi/bin/kaapish --shell --mode=devel --flags‘
$ # replace $HOME/kaapi by the prefix used with the configure script
```

In the latter case (adding to the current environment), you can add the line in your shell rc file (‘`.bashrc`’, ‘`.tcshrc`’, ...) to avoid to type it each time.

If `kaapish` does not correctly detect your kind of shell, you can specify it

```
$ eval ‘$HOME/kaapi/bin/kaapish --shell=sh --mode=devel --flags‘
$ # replace $HOME/kaapi by the prefix used with the configure script
```

More detail about the `kaapish` command can be found in its man page kaapish(1)

3.1.2 Using the pkg-config tool

Once you setup an correct environment or if you installed KAAPI in system directories (usually if you use the ‘`--prefix=/usr`’ with the configure script), you can compile and link with the correct flags by requesting them to `pkg-config` as for any other library managed by `pkg-config`:

```
$ g++ -c atha_test.cpp ‘pkg-config --cflags kaapi‘ # compiling step
$ g++ -o atha_test atha_test.o ‘pkg-config --libs kaapi‘ # linking step
```

3.1.3 Using environment variables

Once you setup an correct environment, you can compile and link with the correct flags by using the new defined environment variables:

```
$ g++ -c atha_test.cpp $ATHAPASCAN1_CPPFLAGS # compiling step
$ g++ -o atha_test atha_test.o $ATHAPASCAN1_LDFLAGS # linking step
```

3.2 Running an Athapascan program

If you have not installed KAAPI in a system-wide location, you will need to use kaapish(1) (or karun(1) to run on several machines, see below) so that environment variables are correctly defined.

3.2.1 Running a single instance of the program

You can run the program directly:

```
$ $HOME/kaapi/bin/kaapish ./atha_test
$ # replace $HOME/kaapi by the prefix used with the configure script
```

Or you can update environment variables in your shell to be able to run your program:

```
$ eval ‘$HOME/kaapi/bin/kaapish --shell --mode=run‘
$ # replace $HOME/kaapi by the prefix used with the configure script
$ ./atha_test
```

NOTE: in case you already setup an environment to compile KAAPI program or if you installed KAAPI in system directories (usually if you use the ‘`--prefix=/usr`’ with the configure script), you already have a correct environment to run Athapascan programs. So you can run it immediately (the last line of the previous example)

Of cause, you can use a more featured program, for example `fibo_apiatha` (in this example, no running environment is previously setup, so `kaapish` is used)

```
$ $HOME/kaapi/bin/kaapish ./fibo_apiatha 30 15
                                #30: the 30-th fibonacci number
                                #15: threshold to stop recursive
```

NOTE: self documentation of KAAPI is displayed by the command line argument ‘`--help`’. All KAAPI script or program compiled with KAAPI accept this option.

For instance:

```
$ $HOME/kaapi/bin/kaapish --help
$ $HOME/kaapi/bin/kaapish ./fibo_apiatha --help
```

One particular option is ‘`--dumprc`’ in order to dump default value of properties into the file ‘`dump.rc`’ in order to change them for next run. KAAPI looks for a properties’ file named ‘`kaapi.rc`’ in `$HOME` and `$PWD`.

3.2.2 Running multiple instances of the program on several machines

To run the same program on several machines, the user may use the `karun` command:

```
$ $HOME/kaapi/bin/karun --np <#processes> -f <hostnames> ./fibo_apiatha 30 15
```

Note that if `kaapish` is not in you `$PATH` when executing an `ssh` on remote machines (check with `ssh <machine> which kaapish`), you will need to specify it to `karun`:

```
$ $HOME/kaapi/bin/karun --wrapper-binary kaapish=/path/on/remote/node/kaapish \
    --np <#processes> -f <hostnames> ./fibo_apiatha 30 15
```

more detail about this command can be found in the man page karun(1)

NOTE: if you want to run your programm on a single machine but still want to use several threads, you can use:

```
$ $HOME/kaapi/bin/kaapish ./myprog --community -thread.poolsize n my args
                                      # n is the number of threads
```

4 Athapascan Concept

Let us dive into KAAPI through its main interface, `Athapascan`

4.1 Programming Model

`KAAPI` is a middleware working on Dynamic Acyclic Data flow graphes. Once given this graph, it can dynamically schedule it using a work-stealing algorithm.

But most distributed computing users are familiar with message passing paradigm, and `KAAPI` uses an other paradigm:

1. Describe the task of your graph / program
2. Describe the dependencies between your task

Once this is done, `KAAPI` will schedule the taks in an efficient way, making sure that

1. All dependencies are respected
2. Parallelism between independant tasks is used as much as possible

Of course, this will not work as expected on all kind of graph, but it has been *proven* to be an asymptotically optimal way to schedule tasks from a divided and conquer algorithm, based on `fork` and `join` calls.

4.1.1 Task description

A task in `Athapascan` is more or less a function object with no side effect. A task execution is somewhat similar to a standard procedure call (Tasks are dynamically created at run time). The only difference is that the created task's execution is fully asynchronous, meaning the creator is not waiting for the execution of the created task to finish to continue with its own execution. So an `Athapascan` program can be seen as a set of tasks scheduled by the library and distributed among nodes for its execution.

4.1.1.1 Task definition

A task corresponds to the execution of a function object, i.e. an object from a class (or structure) having the `void operator()(...)` defined:

```
struct user_task
{
  void operator() ( /* formal parameters */ )
  {
    /*...*/
  }
};
```

A sequential (hence not asynchronous !) call to this function class is written in C++:

```
user_task() ( /* effective parameters */ ) ;
```

And an asynchronous call to this *task* is written in `Athapascan`:

```
a1::Fork< user_task > () ( /* effective parameters */ ) ;
```

See Chapter 6 [API], page 32 for a detailed description of the `a1::Fork` usage.

Note that this does *not* provide a way to describe dependencies between tasks. Dependencies are described through the use of *formal parameters*.

4.1.1.2 Task parameters

In C++, the formal parameters can be either passed by

copy It as a read only meaning: parameters passed by copy are read by the function, but cannot be modified. It is a *read dependency*.

reference It as a read / write meaning: parameters can be read and modified by the function. If the parameter is only written, it is a *write dependency*. Otherwise it is a *read write dependency*.

Unlike functions, tasks will be used in a distibuted environment and their parameters will

- influence their scheduling;
- need to be shared acroos the network (see Section 4.1.3.1 [serialize parameters], page 8).

Here are the different kinds of parameters allowed for a task:

`T` designate a classical `C++` type that does not affect shared memory. However this type must be communicable (see Section 4.1.3.1 [serialize parameters], page 8). It will be copied twice with every call. It is a *read dependency*.

`const T &` designate a classical constant reference to a `C++` type that does not affect shared memory. As the above, the type must be communicable (see Section 4.1.3.1 [serialize parameters], page 8), but it will suffer one less copy. It is a *read dependency*.

`Shared_...< T >`
designate a parameter that is a reference to a shared object located in the shared memory. `T` is the type of this object. `T` must also be communicable (see Section 4.1.3.1 [serialize parameters], page 8). As explained in Section 4.1.3 [shared memory], page 8, it is the `Athapascan` way of describing *read dependency*, *write dependency* or *read write dependency*.

The following kind of parameters are *not* allowed for a task.

`T*` pointers are linked to local memory. There is no sense in sharing pointers over distibuted memory.

`T&` reference are linked to local memory. There is no sense in sharing pointers over distibuted memory. An interesting exception is `const` references. We know that the object will not be modified, so it can be passed by copy.

4.1.2 Task samples

4.1.2.1 From a `C` procedure

Obviously, a `C` procedure (i.e. a `C` function with void as return type) can be directly encapsulated in a `C++` function class. It becomes a task for Athapascan. Here is an example:

```
/* c procedure */
void f ( int i )
{
    printf( "what could I compute with %d ? \n", i );
}
```

The transformation to a function class is straight forward:

```
/* encapsulated procedure */
struct f_encapsulated
{
  void operator() ( a1::Shared_r<int> i ) /* i is some formal parameter*/
  {
    f( i.read() );
  }
};
```

4.1.3 Shared memory

In the message passing paradigm, developpers manually manage send and receive of data across running programs. In `Athapascan`, the message are sent by the middleware to share variable among tasks. The user does *not* need to manage the send and receive, but he must still

4.1.3.1 Serialize parameters

Using a distributed architecture means handling data located in shared memory (mapping, migration, consistency). This implies a serialization step for the arguments of tasks. This serialization has to be explicitly done by the programmer to suit the specific needs of the program.

`Athapascan` already handles some types:

- the following `C++` basic types

  ```
  char
  short
  int
  long
  float
  double
  ```
- some types from the STL

  ```
  vector<...>
  string
  pair<...,...>
  ```

Using this communicable types, you can define other communicable types. To do this, a type *T* must have

- an empty constructor: `T()` ;
- a copy constructor: `T(const & T)` ;
- a serializing operator: `a1::OStream& operator<<( a1::OStream& output_stream, const T& x )` which puts into the *output_stream* the information needed to reconstruct an image of *x* using the `operator >>` ;
- a deserializing operator: `a1::IStream& operator>>( a1::IStream& input_stream, T& x)` which takes from the *input_stream* the information needed to construct the object *x*; it initializes *x* with the value related to the information from *input_stream*.

Following code shows a simple way to serialize a class:

```
struct Complex
{
  double x;
  double y;

  //empty constructor
  Complex()
    :x(0),y(0) {}

  //copy constructor
  Complex( const Complex& z)
    :x(z.x),y(z.y) {}

};

//packing operator
a1::OStream& operator<< (a1::OStream& out, const Complex& z)
{
  return out << z.x << z.y;
}

//unpacking operator
a1::IStream& operator>> (a1::IStream& in, Complex& z)
{
  return in >> z.x>> z.y;
}
```

4.1.3.2 Dependencies description

In order to respect the sequential consistency (lexicographic order semantic), `Athapascan` 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). All manipulated shared data must be declared in the prototype of the task, and to detect the synchronizations between tasks, according to lexicographic semantic, any shared parameter of a task is tagged in the prototype according to the access performed 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 distingueshed and are presented below: read, write, update and accumulation.

read right `a1::Shared_r<`*`T`*`>` is the type of a parameter whose value can only be read using its `const `*`T`*`& read()` method. This reading can be concurrent with other tasks referencing this shared object in the same mode.

write right

`a1::Shared_w<T>` is the type of a parameter whose value can only be written using its `void write(T)` method. 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.

update right

`a1::Shared_rw<T>` is the type of a parameter whoses value can be updated in place; the related value can be read and/or written using the `T& access()` method. 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.

concurrent write right

`a1::Shared_cw<T,F>` is the type of a parameter whose value can only be accumulated with respect to the user defined function class *F*. The resulting value of the concurrent write is an accumulation of all other values written by a call to this function, through the use of the `void cumul(T` method.

For a more in-depth description of the shared parameters, please refer to the Chapter 6 [API], page 32.

4.2 Programming environment

Now that the basic programming model has been described, let us see how it is implemented. To do so we need to

4.2.1 Initialize the library

The execution of an `Athapascan` program is handled by a `a1::Community`. A community restructures a group of nodes so that they can be distributed to the different parallel machines. Therefore, prior to the declaration of any Athapascan object or task, a community must be created. Currently, this community only contains the group of nodes defined at the start of the application.

Once the community has been created, tasks can be created and submited to the community (see Section 4.2.3 [spawn task], page 12).

The library header must first be included to get all prototypes: This defines the `a1` namespace.

```
#include <athapascan-1>
```

Usually the community is defined in the main method of the program.

```
int main( int argc, char** argv )
{
```

`Athapscan` reads its parameters from the program arguments They are used to initialize the `Community`.

```
    a1::Community com = a1::System::join_community( argc, argv );
```

A main task is then spawned. Its prototype is strictely defined: it must define the `void operator()(int argc, char **argv)`. It is called *doit* in the following

```
        a1::ForkMain<doit>()(argc, argv);
```

The starter is hit once we ask to leave the community. A community can only be left if it contains no task.

```
        com.leave();
```

It is often considered to do some cleanup before exiting

```
        a1::System::terminate();
        return 0;
    }
```

Now let us define our first task.

4.2.2 Define task

The task *doit* should contain the code to be executed in parallel. It is often only a matter of spawning additionnal tasks (hopefully in a recursive way). But let say we just want to print a "hello world".

First, a task is nothing else than a structure:

```
    struct doit
    {
```

Second it is also a function obect. As described in Section 4.2.1 [initialize the library], page 10, the *doit* task has specific parameters

```
        void operator()(int argc, char **argv)
        {
```

The body is known across the world (let us suppose we added `#include <iostream>` in the header)

```
            std::cout << "hello world !" << std::endl;
        }
```

For `Java` addict, do not forget to close the structure !

```
    };
```

This is your first Task. Executing it will not result in a parallel execution. We need more task to feed our scheduler. What about a simple sum ? The `C` procedure would be

```
    int sum(int n, int p);
```

Let us do some semantic analysis to make it a task:

n is read by `sum` : it will become a `a1::Shared_r<int>` !

p is also read by `sum` : it will become a `a1::Shared_r<int>` !

return value
is written by `sum` : it will become a `a1::Shared_w<int>` !

So once our task defined:

```
    struct sum_task
    {
    };
```

We can provide it with the good `operator()`

```
        void operator()( a1::Shared_r<int> n, a1::Shared_r<int> p, a1::Shared_w<int> r)
        {
        }
```

The body should be a simple call to the `C` procedure `sum`. But we need to access the content of the shared data (see Chapter 6 [API], page 32 for a list of available methods).

`n.read()`	will read the content of *n*
`p.read()`	will read the content of *p*
`r.write(val)`	will write *val* in *r*.

so the body is

```
            r.write( sum( n.read(), p.read() ) );
```

4.2.3 Spaw task

Now that we have defined our first real task *sum_task*, it is time to spawn it. let us go back to our *doit* special task, and modify it a bit

```
    struct doit
    {
        void operator()(int argc, char **argv)
        {
            if( argc != 3 )
                return;
            int n = atoi(argv[1]);
            int p = atoi(argv[2]);
            int r;
            r = sum(n,p);
            std::cout << r << std::endl;
        }
    };
```

But we want to use our task ! The task takes `a1::Shared` parameters, so

```
            a1::Shared<int> n = atoi(argv[1]);
            a1::Shared<int> p = atoi(argv[2]);
            a1:/Shared<int> r;
```

Once intialized, you can*not* access the content of the shared variable from this task ! It can only be accessed by other tasks with proper access right. This is easy to understand. Has task calls are asynchrnonous, it is hazardous to access them in an uncontrolled way. Task dependencies will take the control.

Now spawn the task to get an asynchronous call to `sum_task`

```
            a1::Fork< sum_task > () ( n, p, r );
```

Question	But what if I want the result ?You just wrote it is impossible to read from *r*!
Answer	Create a new task that will read *r* with proper access right !

```
              struct read_task
              {
                  void operator() ( a1::Shared_r<int> r )
                  {
                      std::cout << r.read() << std::endl;
                  }
              };
```

4.3 When speed matters

It is important to understand that a work stealing sceduling is efficent only if the data flow graph is deep, and grows in a fork / join style. But

- if the tasks are too small, the cost of task creation / scheduling outperforms the cost of task execution, which leads to poor performance
- if too few tasks are created, CPU power may be lost

That's why we need a *threshold*. A threshold is a constant that control when we should stop to create Tasks to execute the sequential algorithm. It often looks like this

```
struct task
{
    void operator()(/* parameters */)
    {
        if( size < threshold )
            /* sequential_call */
        else
            /* fork new task */
    }
};
```

The fibonnaci example behaves like this. Choosing the threshold is a matter of test and experiments, but it is **really** important.

4.4 Additionnal feeatures

Tasks and shared parameters are the basics of `Athapascan`. But it is not always enough to write real-life programs ! Take a look at

4.4.1 Global memory

The current release of `Athapascan` provides the `a1::MonotonicBound` template that allows several processes to read or update a bound that increases or decreases monotonically. Such a variable has a system wide identifier that should be defined during the declaration. It is communicable and could be a parameter of tasks.

To declare a `a1::MonotonicBound`, you must provide the string identifier and two tempalte parameters:

1. the type of the variable updated, *T*
2. the type of the function obect used to update the variable, *F*

```
/* raw initialization */
a1::MonotonicBound<T,F> a ("B&B bound");
/* initialization by value */
```

```
a1::MonotonicBound<T,F> b ( "another B&B bound", new T(/*...*/) );
```

The update function *F* is a function class that should has an `operator()` with the following signature.

```
template < class T> struct F
{
  bool operator()( T& res, const T& val);
};
```

The return value of the operator is true if and only if the local value has been updated. In this, case during the invocation of the release, the protocol will broadcast the new local bound to all other processors which update their bound in the same way.

For instance, the definition of the operator that compute the maximal value could be:

```
struct Max
{
  bool operator()( long& result, const long& value)
  {
    if (result < value)
    {
      result = value;
      return true;
    }
    return false;
  }
};
```

To read (and only read) the value of a `a1::MonotonicBound`, a lock is to be taken on the value, then a call to the `read` method.

```
a.acquire();
a.read(); // return a const T& on the value
a.release();
```

To update the value of a `a1::MonotonicBound`, use the `update` method

```
a.update(new_val);
```

4.4.2 Sharing arrays

As said in Section 4.1.1.2 [task parameters], page 7, tasks parameters are not allowed to be pointers. But High performance Computing make an extensive use of arrays, so what ?

4.4.2.1 Arrays through structures

If you hae a static array, whose size never change, the best option is to encapsulate it in a structure. To use

```
int my_array[256];
```

You define a new communicable structure

```
structure int_array_256
{
    int array[256];
    int_array_256() {}
```

```
        int_array_256(const int_array_256& a)
        {
            std::copy(a,  a +256, array);
        }
    };

    //packing operator
    a1::OStream& operator<< (a1::OStream& out, const int_array_256& i)
    {
        for(int j=0; j<256;j++)
            out  << i[j];
        return out ;
    }

    //unpacking operator
    a1::IStream& operator>> (a1::IStream& in, int_array_256& i)
    {
        for(int j=0; j<256;j++)
            in >> i[j];
        return in;
    }
```

And you can now use it as a formal parameter of your task. Yet note that the copy introduce by such a structure is rarely worth the price !

4.4.2.2 Arrays through verctors

If you want to share value from an array, the best tool is the `std::vector` class. The `std::vector` class is communicable.

```
std::vector<int> v(256);
std::fill( v.begin(), v.end(), 42);
a1::Shared< std::vector<int> > shared_vector = v;
a1::Fork<a_task> () ( shared_vector );
```

4.4.2.3 Arrays through iterators

A remote iterator is a communicable type used to keep the reference of an array across the network, thus allowing access to large amount of remote data. This does not provide any synchronization as a Shared does, so you have to manage dependencies using other means (usually a fake Shared dependencie).

To use remote iterators, you first declare them, and then init them from a local data

```
int array[256];
// for read-write access
a1::remote<int*> begin, end;
a1::init(begin, end, array, array + 256);
// for read only access
a1::const_remote<int*> cbegin, cend;
a1::init(cbegin, cend, array, array + 256);
```

The `const_remote` class will give you a read-only iterator, while the `remote` class will give you read-write access.

You can now use the remote iterators as you would use random access iterators : incrementation, copy, dereferencing and so on is allowded. Of course those iterator can be used as arguments of any STL algoroithm as native iterator would. They have the Random access iterator tag. As they are communicable, you can pass them as parameters to an athapascan'task.

To get a reference on data pointed by the iteratore, you must first ensure a copy of the real data is available on your local machine. You must do so using the `fetch` function : Not using the fetch function may result in a SEGFAULT, for you're trying to access data that may not exist in memory.

```
struct task
{
    void operator() ( a1::remote<int*> b, a1::remote<int*> e)
    {
        size_t n = e -b;
        if( n < 4)
        {
            a1::fetch(b,e);
            while(b<e)
                std::cout << *b++ << std::endl;
        }
        else
        {
            a1::Fork<task>()( b, b+n/2);
            a1::Fork<task>()( b +n/2, e);
        }
    }
};
```

No guarantee is given if the split overrides. Try to avoid it. The detailed interface is given in Chapter 6 [API], page 32.

4.4.3 Control task creation

You sometime want to merge distributed and sequential code. To do this, you must ensure that all tasks have been executed. `a1::SyncGuard` comes handy there. It is a class that ensures that every task spawned betwwen the creation of an object of type `a1::SyncGuard` and its destruction are executed.

```
struct task
{
    void operator()()
    {
        // all tasks
        {
            // from here
                a1::SyncGuard s;
                a1::Fork<task1>()();
                a1::Fork<task2>()();
            // to here
        }
        // have finished when I arrive here
        a1::Fork<task3>()();
    }
};
```

This is often usefull, but beware, it may lower the parallism !

Once a task is created, you have no control over where it will be executed. It is not a problem if the task had no side effect. But they sometime have, so you can specify attribute to the `Fork` operator. The most useful attribute is `a1::SetLocal` which ensure the task will be executed by the process that spawned it. It can be used when preformance matters, or to access local memory.

```
a1::Fork<task>(a1::SetLocal)(/* task arguments */);
```

Here is a little trick to read from local memory. It uses both `a1::SetLocal` and `a1::SyncGuard`. See

```
int global_int;
struct task
{
    void operator()(a1::Shared_r<int> i)
    {
        global_int = i.read();
    }
};

struct doit
{
    void operator()(int , char **)
    {
        a1::Shared<int> i;
        {
            a1::SyncGuard s;
            // some_task writes in i
            a1::Fork<some_task>()(i);
            // task cannot be stolen !
            a1::Fork<task>(a1::SetLocal)(i)
        }
        // global_int is up to date thanks to the guard
        std::cout << global_int << std::endl;
    }
};
```

4.4.4 Use classical algorithms

For ease of use, `Athapascan` offers some high level algorithms, with an interface and a semantic very similar to `STL` ones. Once `Athapascan` has been started and the main task forked, you can freely use any of those algorithms.

Thoses algortihms are described in Chapter 6 [API], page 32, but here is a quick list:

- `a1::sort`
- `a1::stable_sort`
- `a1::transform`
- `a1::for_each`
- `a1::find_if`
- `a1::find`

5 Athapascan tutorials

This chapters list some really simple programs that can help you to understand `Athpascan`'s primitives

5.1 Hello World - 1 Fork, 0 Shared

Say hello to your friends ! Understand the `a1::Fork` usage.

Basics

Each time you want to parallelize a sequential code using Athapascan, you have two things to think of:

- how can I describe the sequential processing in terms of data transformations (which will be called "tasks") ?
- what are the dependencies between those tasks ? Does one requires the result from another ?

Once you have done this, your mind can begin to relax, your finger will begin to code.

Algorithm

Algorithm is a bit petty here. The *hello world* program just displays, for a given value of *n*

```
hello world from 0 !
hello world from 1 !
...
hello world from n-2 !
hello world from n-1 !
```

The following `C++` code will more or less achieve this goal:

```
#include <iostream>
int main(int, char **)
{
  int n = 10; /* number of iteration*/
  for (int id = 0; id < n; id++)
    std::cout << "hello world from " << id << " !" << std::endl;
  return 0;
}
```

Where are the tasks

In this *very simple* code, finding the task, the **job to be done**, is rather simple. What do we want to do ? print a message with a special id (a number). So the task is the printing of the message, it has no return value, and a single int as input.

Task	print_hello
input	`int` id (read)
output	none

Where are the dependencies

Obviously, each task is independent, isn't it ? The engine will automatically deduce this from the fact that each input parameter only has read access.

Preparing the code

Now that we know what are the tasks and the dependencies, we need to describe them in a piece of code. If I follow the Athapascan guide, I read that a task is a function object with void return value. Let's rewrite previous code !

```
#include <iostream>
/* declare my task as a function object */
struct print_hello
{
  void operator()( int id )
  {
    std::cout << "hello world from " << id << " !" << std::endl;
  }
};

int main(int, char **)
{
  int n = 10; /* number of iteration*/
  for (int i = 0; i < n; i++)
    print_hello()(i); /* create and use the function object */
  return 0;
}
```

NOTE: we did not create a single instance of the function object and then make several method calls, as we usually do. This is to prepare to the following : each task will be created separately.

Using Athapascan

Before going further, we have to prepare the use of KAAPI library. It means we have to do some initialization stuff etc.

1. Initialize the library
2. Create the main task
3. Clean up

```
#include <iostream>
#include <athapascan-1.h> // add athapascan header

struct print_hello
{
  void operator()( int id )
  {
    std::cout << "hello world from " << id << " !" << std::endl;
  }
};

struct do_main // the do_main is also a task
```

```
{
  void operator()(int argc, char **argv) // copy main here
  {
    int n = 10; /* number of iteration*/
    for (int i = 0; i < n; i++)
      print_hello()(i); /* create and use the function object */
  }
};

int main(int argc, char **argv)
{
  a1::Community com = a1::System::join_community(argc,argv); // init library
  do_main()(argc,argv); // main call
  com.leave(); // ensure no more task left
  a1::System::terminate(); // clean up
  return 0;
}
```

NOTE: `com.leave()` is important, it computes the termination of the program, checking whether the local task list is empty or not.

After reading the documentation, I understood that there were some kind of shared variables that were used to compute the dependencies. I know that I only need to read the content of id, so I'll put it into a `a1::Shared_r<int>`

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

struct print_hello
{
  void operator()( a1::Shared_r<int> id ) // use shared read access
  {
    // use the read() method to get the content of the shared
    std::cout << "hello world from " << id.read() << " !" << std::endl;
  }
};

struct do_main // the do_main is also a task
{
  void operator()(int argc, char **argv) // copy main here
  {
    int n = 10; /* number of iteration*/
    for (int i = 0; i < n; i++)
    {
      a1::Shared<int> id(i); // put variable in shared memory
      print_hello()(id);
    }
  }
};
```

```
int main(int argc, char **argv)
{
  a1::Community com = a1::System::join_community(argc,argv);
  do_main()(argc,argv);
  com.leave();
  a1::System::terminate();
  return 0;
}
```

NOTE: In fact, the shared memory is not needed here. Passing normal variable as parameters is equivalent to read access for dependencies computing.

I also understood that my tasks were designed to be forked instead of begin called. I must replace my function call by `a1::Fork` !

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

struct print_hello
{
  void operator()( a1::Shared_r<int> id )
  {
    std::cout << "hello world from " << id.read() << " !" << std::endl;
  }
};

struct do_main
{
  void operator()(int argc, char **argv)
  {
    int n = 10; /* number of iteration*/
    for (int i = 0; i < n; i++)
    {
      a1::Shared<int> id(i);
      a1::Fork<print_hello>()(id); // fork a new task
    }
  }
};

int main(int argc, char **argv)
{
  a1::Community com = a1::System::join_community(argc,argv);
  a1::ForkMain<do_main>()(argc,argv); // main fork, only executed by one nod
  com.leave();
  a1::System::terminate();
  return 0;
}
```

This is your first piece of athapascan code !

NOTE: To compile this code, please have a look to the install and compile documentation (see Section 3.1 [compile], page 4)!

NOTE: To get better performance , the for loop should be written recursively, and a threshold should be used. More on this in next tutorial !

5.2 Fibonacci - multiple Fork and Shared

This tutorial will help you to understand how to use sahed memory.

Algorithm

The Fibonacci series is defined as:

- F(0) = 0
- F(1) = 1
- F(n) = F(n - 1) + F(n - 2) for all n > 2

There are different algorithms to resolve Fibonacci numbers, some having a linear time complexity O(n). The algorithm we present here is a recursive method.

NOTE: It is a very bad implementation as it has an exponential complexity, O(2n), (as opposed to the linear time complexity of other algorithms). However, this approach is easy to understand and to paralleled.

Sequential implementation

First, let's have a glance at the regular naive recursive sequential program:

```
#include <iostream>

int fibonacci(int n)
{
    if (n<2)
        return n;
    else
    return fibonacci(n-1)+fibonacci(n-2);
}

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);
    int res =  fibonacci( n );
    std::cout << "result= " << res << std::endl;
    return 0 ;
}
```

We will follow the steps described in first tutorial :

1. identify the tasks
2. identify the dependencies

Where are the tasks

A task is a data transformation. It is easy to see that the Fibonacci function itself is a task, taking a number as input, and returning a number as output.

Task fibonacci

Input `int` n (read)

Output `int` res (write) = fibonacci(n)

But with a little more experience, you will see that the `operator+` combining the result of the two function call is also a task. So we write

Task add

Input `int` i (read), `int` j (read)

Output `int` res (write) = i + j

Where are the dependencies

You cannot add fibonacci(n-1) and fibonacci(n-2) before the function call ends : the input of task add are written by tasks fibonacci(n-1) and fibonacci(n-2) : they shared data ! We describe this in KAAPI by declaring the variable in shared memory, and then describing the access rights used to access the data.

Preparing the code

To make the usage of KAAPI easier, I will rewrite the code in this way :

```
#include <iostream>

int fibonacci(int n)
{
    int res; // output variable
    if (n<2)
    {
        res = n;
    }
    else
    {
int arg1 = n-1; // input variable
int tmp1 = fibonacci(arg1);
int arg2 = n-2; // input variable
int tmp2 = fibonacci(arg2);
res = tmp1 + tmp2;
    }
    return res;
}

int main(int argc, char** argv)
{
    int n = atoi(argv[1]);
    int res =  fibonacci( n );
    std::cout << "result= " << res << std::endl;
    return 0 ;
}
```

Using Athapascan

The implementation of the add task is a good example of KAAPI task : it is simple, but uses two different of shared variable. The sequential function is

```
int add (int i, int j)
{
  int res;
  res = i + j;
  return res;
}
```

As a function object with no return value, it could be

```
struct add
{
  void operator()( int i /* read*/,
    int j/* read*/, int& res/*write*/)
    // use a reference instead of return value
  {
    res = i + j ;
  }
};
```

You note that i and j are read, res is only written, so I will replace the type by the equivalent shared type with the correct access right:

`int` *[read]*	`a1::Shared_r<int>`
`int` *[write]*	`a1::Shared_w<int>`

```
struct add
{
  void operator()( a1::Shared_r<int> i,
    a1::Shared_r<int> j,
    a1::Shared_w<int> res )
    // replace with shared variable
  {
    res.write( i.read() + j.read() );
  }
};
```

NOTE: You cannot directly access the content of a shared, you must use the appropriate method. Of course, you cannot `read()` the content of a `shared_w` ...

Let's write the code for the fibonacci task. As usual, we will write it in two steps. First use the function object design :

```
struct fibonacci
{
  void operator()( int n /*read*/, int& res /*write*/ )
  {
    if( n < 2 )
    {
      res = n ; /* write operation here */
    }
    else
    {
      int res1;
      fibonacci()(n-1,res1); // fibo task call
      int res2;
      fibonacci()(n-2,res2); // fibo task call
      add()( res1, res2, res ); // add task call
    }
  }
};
```

Then use the shared memory and task spawning :

`int` *[write]*	`a1::Shared_w<int>`
`add`*()(...)*	`a1::Fork<add>()`
`fibonacci`*()(...)*	`a1::Fork<fibonacci>()`

NOTE: As explained in previous tutorial, it is useless to put a read-only variable in shared memory if there is no other dependencies on this variable.

```
struct fibonacci
{
  void operator()( int n , a1::Shared_w<int> res )
  {
    if( n < 2 )
    {
      res.write(n) ; // replace the affectation by a write call
    }
    else
    {
      a1::Shared<int> res1;
      a1::Fork<fibonacci>()(n-1,res1); // fibo task call
      a1::Shared<int> res2;
      a1::Fork<fibonacci>()(n-2,res2); // fibo task call
      a1::Fork<add>()( res1, res2, res ); // add task call
    }
  }
};
```

Here you are, both tasks have been written, you just have to include the standard main of an Athapascan program to get the following result :

```
//! run as: karun -np 2 --threads 2 ./fibo_apiatha 36 4

/****************************************************************************
```

```
 *
 *  Shared usage sample : fibonnaci
 *
 *************************************************************************/


#include <iostream>
#include "athapascan-1" // this is the header required by athapascan


// ---------------------------------------------------------------------
/* Sequential fibo function
 */
unsigned long long fiboseq(unsigned long long n)
{ return (n<2 ? n : fiboseq(n-1)+fiboseq(n-2) ); }

unsigned long long fiboseq_On(unsigned long long n){
  if(n<2){
    return n;
  }else{

    unsigned long long fibo=1;
    unsigned long long fibo_p=1;
    unsigned long long tmp=0;
    unsigned long long i=0;
    for( i=0;i<n-2;i++){
      tmp = fibo+fibo_p;
      fibo_p=fibo;
      fibo=tmp;
    }
    return fibo;
  }
}

/* Print any typed shared
 * this task has read acces on a, it will wait until previous write acces on it a
*/
template<class T>
struct Print {
  void operator() ( a1::Shared_r<T> a, const T& ref_value, a1::Shared_r<double> t
  {
    /*  Util::WallTimer::gettime is a wrapper around gettimeofday(2) */
    double delay = Util::WallTimer::gettime() - t.read();

    /*  a1::System::getRank() prints out the id of the node executing the task */
    Util::logfile() << a1::System::getRank() << ": -----------------------------
    Util::logfile() << a1::System::getRank() << ": res  = " << a.read() << std::e
    Util::logfile() << a1::System::getRank() << ": time = " << delay << " s" << s
    Util::logfile() << a1::System::getRank() << ": -----------------------------
    KAAPI_LOG( a.read() != ref_value, "**** Error **** : bad value" );
  }
};


/* Sum two integers
 * this task reads a and b (read acces mode) and write their sum to res (write ac
 * it will wait until previous write to a and b are done
 * once finished, further read of res will be possible
 */
struct Sum {
  void operator() ( a1::Shared_w<unsigned long long> res,
                    a1::Shared_r<unsigned long long> a,
                    a1::Shared_r<unsigned long long> b)
  {
    /* write is used to write data to a Shared_w
     * read is used to read data from a Shared_r
     */
    res.write(a.read()+b.read());
  }
};

/* Get current time
 */
struct GetTime {
  void operator() ( a1::Shared_rw<double> t)
  {
    t.access() = Util::WallTimer::gettime();
  }
};

/* Athapascan Fibo task
 * - res is the return value, return value are usually put in a Shared_w
 * - n is the order of fibonnaci. It could be a Shared_r, but there are no depend
 * - threshold is used to control the grain of the application. The greater it is
 *   a high value of threshold also decreases the performances, beacause of athap
 */
struct Fibo {
  void operator() ( a1::Shared_w<unsigned long long> res, int n, int threshold, a
  {
    if (n < threshold) {
      res.write( fiboseq(n) );
    }
    else {
      a1::Shared<unsigned long long> res1;
      a1::Shared<unsigned long long> res2;
      a1::Shared<double> t;

      /* the Fork keyword is used to spawn new task
       * new tasks are executed in parallel as long as dependencies are respected
       */
      a1::Fork<Fibo>() ( res1, n-1, threshold, t );
      a1::Fork<Fibo>() ( res2, n-2, threshold, t );

      /* the Sum task depends on res1 and res2 which are written by previous task
       * it must wait until thoses tasks are finished
       */
      a1::Fork<Sum>()  ( res, res1, res2 );
    }
  }
};


/* Main of the program
*/
struct doit {

  void do_experiment(unsigned int n, unsigned int seuil, unsigned int iter )
  {
    double t = Util::WallTimer::gettime();
    unsigned long long ref_value = fiboseq_On(n);
    double delay = Util::WallTimer::gettime() - t;
    Util::logfile() << "[fibo_apiatha] Sequential value for n = " << n << " : " <
                    << " (computed in " << delay << " s)" << std::endl;
    a1::Shared<double> time(0.0);
    for (unsigned int i = 0 ; i < iter ; ++i)
    {
      /* notice how useless the init value is */
      a1::Shared<unsigned long long> res(31415);

      a1::Fork<GetTime>(a1::SetLocal)(time);

      a1::Fork<Fibo>()( res, n, seuil, time );

      /* a1::SetLocal ensures that the task is executed locally (cannot be stolen
      a1::Fork<Print<unsigned long long> >(a1::SetLocal)(res, ref_value, time);
    }
  }

  void operator()(int argc, char** argv )
  {
    unsigned int n = 30;
    if (argc > 1) n = atoi(argv[1]);
    unsigned int seuil = 2;
    if (argc > 2) seuil = atoi(argv[2]);
    unsigned int iter = 3;
    if (argc > 3) iter = atoi(argv[3]);

    Util::logfile() << "In main: n = " << n << ", seuil = " << seuil << ", iter =
    do_experiment( n, seuil, iter );
  }
};


/* user store for global variable
*/
void fibo_userglobal( Util::OStream& out )
{
  static const char* msg = "ceci est la variable globale de fibonnaci";
  out.write(Util::WrapperFormat<char>::format, Util::OStream::IA, msg, strlen(msg
}


/* main entry point : Athapascan initialization
*/
#if defined(KAAPI_USE_IPHONEOS)
void* KaapiMainThread::run_main(int argc, char** argv)
#else
int main(int argc, char** argv)
#endif
{
  try {
#if defined(KAAPI_USR_FT)
    FT::set_savehandler( &fibo_userglobal );
#endif

    /* Join the initial group of computation : it is defining
       when launching the program by a1run.
    */
    a1::Community com = a1::System::join_community( argc, argv );

    /** Print pid/gid
    */
    KAAPI_LOG(true, "[main] pid=" << getpid());

    /* Start computation by forking the main task */
    a1::ForkMain<doit>()(argc, argv);

    /* Leave the community: at return to this call no more athapascan
```

```
           tasks or shared could be created.
        */
        com.leave();

        /* */
        a1::System::terminate();
      }
      catch (const a1::InvalidArgumentError& E) {
        Util::logfile() << "Catch invalid arg" << std::endl;
      }
      catch (const a1::BadAlloc& E) {
        Util::logfile() << "Catch bad alloc" << std::endl;
      }
      catch (const a1::Exception& E) {
        Util::logfile() << "Catch : "; E.print(std::cout); std::cout << std::endl;
      }
      catch (...) {
        Util::logfile() << "Catch unknown exception: " << std::endl;
      }

      return 0;
    }
```

6 Athapascan Application Programming Interface

This is a quick reference to most classes and functions defined in the header '`<athapascan-1>`'. Do not forget to had

```
#include <athapscan-1>
using namespace a1; // optional
```

to your source code !

6.1 Fork

Declaration

```
template<class Task, class Attribute >
class Fork
{
    Fork(Attribute = DefaultAttribut );

    void operator()( ... );
};
```

Template parameters

Task This template parameter is used to know which task will be spawned. It must implement a `operator()(...)` method. The parameters of the `Fork` method `operator()` are the same as the `operator()` from *Task*.

Attribute This template parameter is never specified direclty. Instead, it is deduced from the call to the `Fork` constructor. The class given can chage the bahavior of the forked task. Possible values are

DefaultAttribute
The default behavior, nothing particular

SetLocal Force the forked task to be executed locally

Methods

`Fork(Attribute = DefaultAttribut)`
The constructor of the `Fork` class. It is always used to construct a temporary object, from which you call the `operator()` method. The `Attribute` parameter determines the behavior of forked task, as described before.

`void operator() (...)`
The parameters of the method `operator()` are the same as the `operator()` from *Task*. This method spawns a task of type *Task* into the local stack, waiting to be executed by current process, or to be stolen by a remote process.

6.2 Shared

`Shared`

Declaration

```
template<class T>
class Shared
{
    Shared();
    Shared(T );
    Shared(T* );
    const T& get_data();
};
```

Template Parameters

T	Type of the variable to share. Type *T* must be communicable.

Methods

`Shared()` Create an empty shared variable.

`Shared(T` *t*`)`
: Put a copy of *t* into the shared memory.

`Shared(T* )`
: Put a copy of **t* into the shared memory.

`get_data()`
: Return content of shared, use cautiously because no check on the internal state is done

`Shared_r`

Declaration

```
template<class T>
class Shared_r
{
    const T& read();
};
```

Template Parameters

T	Type of the variable to share. Type *T* must be communicable. The variable held can only be read.

Methods

`read()` Returns the content (in read only mode) of the object. The returned value cannot be modified. Further write must wait until the end of current task.

`Shared_w`

Declaration

```
template<class T>
```

```
class Shared_w
{
    void write(T t);
};
```

Template Parameters

T Type of the variable to share. Type *T* must be communicable. The variable held can only be written.

Methods

`write()` Write the content of *t* into the object. Further read must wait until the end of current task. In case of concurrent `write`, no guarantee on the order is given.

`Shared_rw`

Declaration

```
template<class T>
class Shared_rw
{
     T& access()
};
```

Template Parameters

T Type of the variable to share. Type *T* must be communicable.

Methods

`access()` Return the content of *t* of the object. The returned value can be modified. Further read or write must wait until the end of current task.

`Shared_cw`

Declaration

```
template<class T, class F>
class Shared_cw
{
     void cumul(T t)
};
```

Template Parameters

T Type of the variable to share. Type *T* must be communicable.

F Type of the update function object. *F* must have an empty constructor and a `void operator()( T&, const T&);` method.

Methods

`cumul(T t)`
: Add the content of *t* to the object in the sense of *F*. There is no constriant on the order between various `cumul` call on the same shared object. Further read must wait until the end of current task.

6.3 MonotonicBound

Declaration

```
template<class T, class F>
class MonotonicBound
{
    MonotonicBound(const string& name, T* initial_value = 0 );
    void update(const T& t);
    void acquire();
    void release();
    const T& read() const;
};
```

Template Parameters

T Type of the variable to share. Type *T* must be communicable.

F Type of the update function object. *F* must have an empty constructor and a `bool operator() (T& result, const T& value );` method. this method returns true if *result* was updated.

Methods

`MonotonicBound(const string& name, T* initial_value = 0 )`
: Constructor of the class. *name* must be a unique identifier. If given, the *initial_value* is now owned by the object and need not to be realeased.

`update(const T& t)`
: Update the content of the object using the *F* update function. It is responsible to get the update the local value of the bound. The invocation of update should be surround by invocations `acquire/release`. Multiple update may be invoked between invocations to `acquire/release`.

`acquire()`
: Is responsible to get the initial value of the bound from processor that manage it. If such an initial value exists, then the method take the last updated value as the current value.

`release()`
: Is responsible to update all copies of the bound is the local value has been updated. After the invocation of release, the local processor that has release the bound should re-invoke `iacquire`: the read value will be the updated value. Other processor will be able to read the updated value in a bounded time.

`read()` is responsible returns a reference to the last value acquired by the invocation to `acquire`. The invocation of read should be surround by invocations to `acquire/release`. Multiple read may be invoked between invocations to `acquire/release`.

6.4 SyncGuard

Declaration

```
class SyncGuard
{
    SyncGuard();
    ~SyncGuard();
};
```

Methods

`SyncGuard()`
Constructor of the class. When an object of class `SyncGuard` is created, it creates a special fram in which all further tasks willbe spawned. When the object is destroyed, the frame is closed and all generated tasks are executed.

`~SyncGuard()`
Destructor of the class. When called, it forces the execution of all tasks in its frame. This method only returns when all tasks (and children) have been executed.

6.5 Algorithms

`for_each`

Description

```
template <class InputIterator, class Function>
void for_each (InputIterator first, InputIterator last, Function f
  typename std::iterator_traits<InputIterator>::difference_type th
```

Algorithm Applies function *f* to each of the elements in the range [*first*,*last*).

The behavior of this template function is equivalent to:

```
template<class InputIterator, class Function>
void for_each(InputIterator first, InputIterator last, Function f)
{
  while ( first!=last ) f(*first++);
}
```

Parameters

first
last Input iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains

all the elements between first and last, including the element pointed by first but not the element pointed by last.

f Unary function taking an element in the range as argument. This can either *not* be a pointer to a function. It *must* be an object whose class overloads `operator()`. Its return value, if any, is ignored.

threshold integer used as grain size. It is comutedn automatically if not provided, but should be set by hand for best performances.

- as an additionnal requirement, *f* and data from *first* to *last* must all be communicable !!
- Currently, only pointers iterator are supported.
- no hypothesis can be made on the order of appliance for *f*.

`transform`

Description

```
template < class InputIterator, class OutputIterator, class UnaryO
void transform ( InputIterator first1, InputIterator last1,
                    OutputIterator result, UnaryOperator op
                    size_t threshold = 0 );

template < class InputIterator1, class InputIterator2,
           class OutputIterator, class BinaryOperator >
void transform ( InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, OutputIterator r
                    BinaryOperator binary_op,
                    size_t threshold = 0 );
```

Algorithm The first version applies op to all the elements in the input range ([*first1*,*last1*)) and stores each returned value in the range beginning at result. The second version uses as argument for each call to binary_op one element from the first input range ([*first1*,*last1*)) and one element from the second input range (beginning at *first2*).

The behavior of this function template is equivalent to:

```
template < class InputIterator, class OutputIterator, class UnaryO
void transform ( InputIterator first1, InputIterator last1,
                    OutputIterator result, UnaryOperator op
{
  while (first1 != last1)
    *result++ = op(*first1++);  // or: *result++=binary_op(*first1
}
```

The function allows for the destination range to be the same as one of the input ranges to make transformations *in place*.

Parameters

first1
last1

Input iterators to the initial and final positions of the first sequence. The range used is [*first1*,*last1*), which contains all the elements between *first1* and *last1*, including the element pointed by *first1* but not the element pointed by *last1*.

first2 Input iterator to the initial position of the second range. The range includes as many elements as [*first1*,*last1*).

result Output iterator to the initial position of the range where function results are stored. The rangeincludes as many elements as [*first1*,*last1*).

op Unary function taking one element as argument, and returning some result value. This can *not* be a pointer to a function. It *must* be an object whose class overloads `operator()`.

binary_op Binary function taking two elements as argument (one of each of the two sequences), and returning some result value. This can *not* be a pointer to a function. It *must* be an object whose class overloads `operator()`.

shold Value of the Grain size used to bound parallelism. If none given, it will be automatically computed, set it by hand for best performances.

- as an additionnal requirement, *binary_op* and data from *first1* to *last1*, *first2*, *result* must all be communicable !!
- Currently, only pointers iterator are supported.
- no hypothesis can be made on the order of appliance for *binary_op*.

`find`

Description

```
template <class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, cons
                     typename std::iterator_traits<InputIterator>:
```

Algorithm

Returns an iterator to the first element in the range [*first*,*last*) that compares equal to *value*, or *last* if not found.

The behavior of this function template is equivalent to:

```
template<class InputIterator, class T>
InputIterator find ( InputIterator first, InputIterator last, cons
{
  for ( ;first!=last; first++) if ( *first==value ) break;
  return first;
}
```

Parameters

first *last*	Input iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.
value	Value to be compared to each of the elements.
threshold	Value of the Grain size used to bound parallelism. If none given, it will be automatically computed, set it by hand for best performances.

Return value

An iterator to the first element in the range that matches *value*. If no element matches, the function returns *last*.

- as an additionnal requirement, data from *first1* to *last1* and *value* must all be communicable !!
- Currently, only pointers iterator are supported.

`find_if`

Description

```
template <class InputIterator, class T>
InputIterator find_if ( InputIterator first, InputIterator last, c
                          typename std::iterator_traits<InputIterator>:
```

Algorithm

Returns an iterator to the first element in the range [*first*,*last*) for which applying *pred* to it, is *true*.

The behavior of this function template is equivalent to:

```
template<class InputIterator, class Predicate>
InputIterator find_if ( InputIterator first, InputIterator last, P
{
  for ( ; first!=last ; first++ ) if ( pred(*first) ) break;
  return first;
}
```

Parameters

first *last*	Input iterators to the initial and final positions in a sequence. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.

pred Unary predicate taking an element in the range as argument, and returning a value indicating the falsehood (with false, or a zero value) or truth (true, or non-zero) of some condition applied to it. This can *not* be a pointer to a function. It *must* be an object whose class overloads `operator()`.

threshold Value of the Grain size used to bound parallelism. If none given, it will be automatically computed, set it by hand for best performances.

Return value

An iterator to the first element in the range for which the application of *pred* to it does not return false (zero). If *pred* is false for all elements, the function returns *last*.

- as an additionnal requirement, *pred* and data from *first* to *last* must all be communicable !!
-
- Currently, only pointers iterator are supported.

`sort` and `stable_sort`

Description

```
template <class RandomAccessIterator>
void stable_sort ( RandomAccessIterator first, RandomAccessIterato
    typename std::iterator_traits<RandomAccessIterator>::differenc

template <class RandomAccessIterator, class Compare>
void sort ( RandomAccessIterator first, RandomAccessIterator last,
    typename std::iterator_traits<RandomAccessIterator>::differenc
```

Algorithm

Sorts the elements in the range [*first*,*last*) into ascending order. `stable_sort` also grants that the relative order of the elements with equivalent values is preserved.

The elements are compared using `operator<`.

Parameters

first
last Random-Access iterators to the initial and final positions of the sequence to be sorted. The range used is [*first*,*last*), which contains all the elements between *first* and *last*, including the element pointed by first but not the element pointed by "last".

threshold Value of the Grain size used to bound parallelism. If none given, it will be automatically computed, set it by hand for best performances.

- as an additionnal requirement, data from *first* to *last* must all be communicable !!

- Currently, only pointers iterator are supported.

6.6 Utilities

`Util::logfile()`
manage logs.

Description

```
std::ostream& logfile();
```

Return Value

returns a `std::ostream` reference that can be used to print out any kind of text. The text will be forwrded to the master node and displayed with a smaal header containing information about

- the global id of the source node
- the date when the log was taken (according to local clock)

`Util::WallTimer::gettime()`
take time

Description

```
double gettime();
```

Return Value

time since the birth of Unix in milliseconds

`resize_vector(std::vector< a1::Shared<...> >, size_type)`
resize a vector of Shared.Its behavior is similar to vector::resize() method

Description

```
template<class T>
void resize_vector(std::vector< a1::Shared<T> >& v,
    typename std::vector< a1::Shared<T> >::size_type sz);
```

Parameters

v	vector of shared to be resized
sz	new size of the vector

6.7 Dynamically Loaded Modules - enhance kaapi

This tutorial will show you how to develop an external module and dynamically load it in any already compiled kaapi program.

6.7.1 API - interface required for a Dynamically loaded module

Basically, a Dynamically loaded module is a standard `Kaapi` module. In order to make it dynamically loadable, you must also provide a `factory` function.

`create_module` [Function]

`extern "C" Util::KaapiComponent* create_module();` factory function that return a new instance of the `Util::KaapiComponent` derived class. The kaapi module loader will call the `delete_module` to free allocated memory.

`delete_module` *kaapi_component* [Function]

`extern "C" void delete_module(Util::KaapiComponent* kaapi_component` The counterpart of the `create_module` function. Instead of allocating memory for a new module, it frees the memory of given pointer.

A detailed description of the `Util::KaapiComponent` is given in the doxygen documentation. As a simple reminder, it's interface is given here:

```
#include <utils_component.h>
class Util::KaapiComponent
{

    public:
    int initialize() throw();
    int terminate() throw();
    void add_options( Util::Parser* , Util::Properties* );
    void declare_dependencies();
};
```

Note that the `declare_dependencies` is of no use for dynamically loaded modules. These modules are loaded after all the static ones, in the order given at the command line.

6.7.2 Command Line Interface - interface to load extra modules from the kaapi command line

This sections describes the usage of the *dl* module in charge of the dynamic loading of extra modules. It uses the kaapi way of passing arguments to modules: '`--module-name`' '`-option-name`' *value* A description of a module option can be found by running any kaapi program with

```
$ kaapi_prog --help module-name
```

The *dl* module provides two options:

'`-verboseon`'
: set it to *true* or *false* to get status report from the *dl* module.

'`-load`'
: description of the module to load, in the form

```
(path_to_the_module.so(:arguments of the module)?,)*path_to_the_module.so(:a
```

Function, methods and classes index