X-Kaapi C programming interface

Fabien Le Mentec, Vincent Danjean, Thierry Gautier

X-KAAPI C programming interface

Fabien Le Mentec, Vincent Danjean, Thierry Gautier

TECHNICAL REPORT
N° 0417
December 2011
Project-Teams MOAIS

ISSN 0249-0803   ISRN INRIA/RT-0417--FR+ENG

X-Kaapi C programming interface

Fabien Le Mentec, Vincent Danjean, Thierry Gautier

Project-Teams MOAIS

Technical Report n° 0417 — December 2011 — 15 pages

Abstract: This report defines the X-Kaapi C programming interface.

Key-words: parallel computing, X-Kaapi, C

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

X-Kaapi C programming interface

Résumé : The rapport décrit l'interface de programmation C pour X-Kaapi

Mots-clés : calcul parallel, X-Kaapi, C

Contents

1 Software installation

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

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

Supported Platforms

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

- a GNU toolchain (GCC $\geq$ 4.3),
- the pthread library,
- Unix based environment.

It has been extensively tested on the following operating systems:

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

There is no version for Windows yet.

X-Kaapi Contacts

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

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

2 Initialization and termination

2.1 Synopsis

```
#include "kaapic.h"

int kaapic_init(int flags)
int kaapic_finalize(void)
```

2.2 Description

kaapic_init initializes the runtime. It must be called by the program before using any of the other routines. If successful, there must be a corresponding *kaapic_finalize* at the end of the program.

2.3 Parameters

flags if not zero, only start the main thread to avoid disturbing the execution until tasks are actually scheduled. The other threads are suspended waiting for a parallel region to be entered (refer to kaapic_begin_parallel in part 7 on page 14).

2.4 Return value

0 in case of success

else an error code

2.5 Example

```
#include "kaapic.h"

int main()
{
  int err = kaapic_init(1);

  ...

  kaapic_finalize();
}
```

3 Concurrency

3.1 Synopsis

```
#include "kaapic.h"

int kaapic_get_concurrency(void)
int kaapic_get_thread_num(void)
```

3.2 Description

Concurrency related routines.

3.3 Return value

kaapic_get_concurrency returns the number of parallel threads available to the X-KAAPI runtime.

kaapic_get_thread_num returns the current thread identifier. Note it should only be called in the context of a X-KAAPI thread.

3.4 Example

```
#include "kaapic.h"

int main()
{
  int err = kaapic_init(1);

 printf("#available threads: %i\n",
    kaapic_get_concurrency() );
 printf("My thread identifier is: %i\n",
    kaapic_get_thread_num() );
  ...

  kaapic_finalize();
}
```

4 Performance

4.1 Synopsis

```
#include "kaapic.h"

double kaapic_get_time(void)
```

4.2 Description

Capture the current time. Used to measure the time spent in a code region.

4.3 Parameters

None.

4.4 Return value

Time in seconds since an arbitrary time in the past.

4.5 Example

```
#include "kaapic.h"

int main() {
  double start, stop;
  int err = kaapic_init(1);
  start = kaapi_get_time();
  ...
  stop = kaapi_get_time();

  printf("Time : %f (s)\n", stop-start );
  kaapic_finalize();
}
```

5 Independent loops

5.1 Synopsis

```
#include "kaapic.h"

int kaapic_foreach(
  int first,
  int last,
  kaapic_foreach_attr_t* attr,
  int32_t nparam,
  ...
 )

int kaapic_foreach_withformat(
  int first,
  int last,
  kaapic_foreach_attr_t* attr,
  int32_t nparam,
  ...
 )
```

5.2 Description

Those routines run a parallel loop over the range [*first*, *last*)[1] The loop is given as function with its parameters. The body function has *nparam* parameters and it is passed in the ... optional effective parameter list of the foreach interface.

At runtime, the initial interval is dynamically split in K disjoint intervals $[b_i, e_i)$ such that $\cup_{i=0..K-1}[b_i, e_i) = [first, last)$. The X-Kaapi threads call *body(* b_i, e_i, *tid,* e_0, *...,* $e_{nparam-1}$*)* for each of these sub-intervals. Hence, *tid* is the thread identifier of the thread that makes the call. And the different calls can occur in parallel if they are done by different threads.

attr is a pointer to an attribute that can be pass tuning parameter to the runtime. It should be null for now, until future extensions are developed and stabilized.

5.3 Parameters

For *kaapi_foreach* interface, the format of the optional parameter list is:

body the function with signature

void (*)(int, int, int [,$type_0$*, .., *$type_{nparam-1}$*]).*

Each type $type_i$ could be:

[1] This is an **exclusive** interval in the C interface and an **inclusive** interval in the Fortran interface.

- a pointer to a memory data
- a scalar value with size equal to the size of a pointer.

e_0 first effective parameter passed to *body*.

...

$e_{nparam-1}$ last effective parameters passed to *body*.

For *kaapi_foreach_with_format* interface extend *kaapi_foreach* interface in order to pass the size, the type and the access mode of each of the effective parameter. The format of the optional parameter list is:

body the function with signature
void (*)(int, int, int [,type$_0$*, .., type*$_{nparam-1}$*]).*
Each type *type*$_i$ could be:

- a pointer to a memory data
- a scalar value with size equal to the size of a pointer.

mode, e_0**, count, type** access mode, first effective parameter passed to *body*, the number of type elements pointed by e_0 and the type of each element.

...

mode, $e_{nparam-1}$**, count, type** access mode, last effective parameter passed to *body*, the number of type elements pointed by $e_{nparam-1}$ and the type of each element.

Please refer to the data flow programming section (6 on page 11) to have a description of *mode* and *type* informations.

5.4 Return value

In case of success the function return 0, else it returns an error code.

5.5 Example

Refer to examples/kaapic subdirectory in sources

```
#include "kaapic.h"

/* loop body */
static void body(
  int i, int j, int tid, double* array, double* value
)
{
```

```
  int k;
  for (k = i; k < j; ++k)
    array[k] += *value;
}

int main()
{
  double* array;
  double value;
  int err = kaapic_init(1);
  start = kaapi_get_time();
  /* apply body on array[0..size-1] */
  kaapic_foreach( 0, size, 2, body, array, &value );
  stop = kaapi_get_time();

  printf("Time : %f (s)\n", stop-start );
  kaapic_finalize();
}
```

The next example is equivalent to the previous:

```
#include "kaapic.h"

/* loop body */
static void body(
  int i, int j, int tid, double* array, double* value
)
{
  int k;
  for (k = i; k < j; ++k)
    array[k] += *value;
}

int main()
{
  double* array;
  double value;
  int err = kaapic_init(1);
  start = kaapi_get_time();
  /* apply body on array[0..size-1] */
  kaapic_foreach_with_format( 0, size, 2, body,
      KAAPIC_MODE_RW, array, size, KAAPIC_TYPE_DOUBLE
      KAAPIC_MODE_V, &value, 1, KAAPIC_TYPE_DOUBLE
  );
  stop = kaapi_get_time();

  printf("Time : %f (s)\n", stop-start );
  kaapic_finalize();
}
```

6 Dataflow programming

6.1 Synopsis

```
#include "kaapic.h"

int kaapic_spawn(int32_t nargs, ...)
```

6.2 Description

Create a new computation task implemented by a call to a function *body* with effective parameters e_i.

The function *body* as well as its effective parameters are pass in the optional parameter list of *kaapic_spawn*. The format of the optional parameter list is:

body : the function with signature
void (*)([type$_0$*, .., type*$_{nparam-1}$*]).*
Each type $type_i$ could be:

- a pointer to a memory data
- a scalar value with size equal to the size of a pointer.

body is called with the user specified arguments, there is no argument added by X-KAAPI:

```
        body(e0, e1, ..., )
```

6.3 Parameters

nargs the argument count;

... the *body* followed by a list of groups of 4 arguments (*mode*, *value*, *count*, *type*).

6.3.1 Format of each 4 successive arguments

Each task parameter is described by 4 successive arguments including:

- the access *mode*.
- the argument *value*,
- the element *cound*,
- the parameter *type*

6.3.2 Mode information

The parameter *mode* is one of the following:

- KAAPIC_MODE_R for a read access,
- KAAPIC_MODE_W for a write access,
- KAAPIC_MODE_RW for a read write access,
- KAAPIC_MODE_V for a parameter passed by value.

6.3.3 Type information

The *type* is one of the following:

- KAAPIC_TYPE_CHAR,
- KAAPIC_TYPE_INT,
- KAAPIC_TYPE_REAL,
- KAAPIC_TYPE_DOUBLE.

If a parameter is an array, *count* must be set to the number of the element of the array. For a scalar value, it must be set to 1.

6.4 Return value

None.

6.5 Example

Refer to examples/kaapif/dfg subdirectory in sources

```
#include "kaapic.h"

/* computation task entry point */
void fibonacci(int n, int* result)
{
  /* task user specific code */
  if (n <2)
    *result = n;
   else
   {
      int result1, result2;
      kaapic_spawn( 2, fibonacci,
        KAAPIC_MODE_V, n-1, 1, KAAPIC_TYPE_INT
        KAAPIC_MODE_W, &result1, 1,  KAAPIC_TYPE_INT
      );
```

```
        kaapic_spawn( 2, fibonacci,
          KAAPIC_MODE_V, n-2, 1, KAAPIC_TYPE_INT
          KAAPIC_MODE_W, &result2, 1, KAAPIC_TYPE_INT
        );
        kaapic_sync();
        *result = result1 + result2;
    }
}

int main()
{
  int n = 30;
  int result= 0;
  int err = kaapic_init(1);

  start = kaapi_get_time();
  /* apply body on array[0..size-1] */
  kaapic_spawn( 2, fibonacci,
      KAAPIC_MODE_V, n, 1, KAAPIC_TYPE_INT
      KAAPIC_MODE_W, &result, 1, KAAPIC_TYPE_INT
  );
  stop = kaapi_get_time();

  printf("Time : %f (s)\n", stop-start );
  kaapic_finalize();
}
```

7 Parallel regions

7.1 Synopsis

```
#include "kaapic.h"

int kaapic_begin_parallel(void)
int kaapic_end_parallel(int flag)
```

7.2 Description

kaapic_begin_parallel and *kaapic_end_parallel* mark the start and the end of a parallel region. Regions are used to wake-up and suspend the X-Kaapi system threads so they avoid disturbing the application when idle. This is important if another parallel library is being used. Whether threads are suspendable or not is controlled according by the *kaapi_init* parameter.

7.3 Parameters

falg if zero, an implicit synchronization is inserted before leaving the region.

7.4 Return value

None.

7.5 Example

```
#include "kaapic.h"

int main()
{
  int err = kaapic_init(1);

  kaapic_begin_parallel();
  ...
  kaapic_end_parallel();
  ...

}
```

8 Synchronization

8.1 Synopsis

```
#include "kaapic.h"

void kaapic_sync(void)
```

8.2 Description

Synchronize the sequential with the parallel execution flow. When this routine returns, every computation task has been executed and memory is consistent for the processor executing the sequential flow.

8.3 Return value

None.

8.4 Example

Refer to the Fibonacci example in section 6.5 on page 12.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

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

ISSN 0249-0803