X-KAAPI Fortran programming interface

Fabien Le Mentec, Vincent Faucher, Thierry Gautier

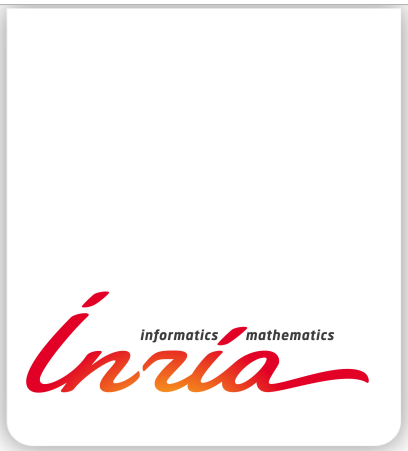
TECHNICAL REPORT

N° xxxx

November 2011

Project-Teams MOAIS

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



X-Kaapi Fortran programming interface

Fabien Le Mentec*, Vincent Faucher†, Thierry Gautier*

Project-Teams MOAIS

Technical Report n° xxxx — November 2011 — 16 pages

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

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

* INRIA, MOAIS project team, Grenoble France, `http://moais.imag.fr`
† CEA, DEN, DANS, DM2S, SEMT, DYN, F-91191 Gif-sur-Yvette, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Interface de programmation Fortran pour X-Kaapi

Résumé : Pas de résumé

Mots-clés :

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 comes also with a full set of complementary programming interfaces: C, C++ and STL-like interfaces. The C++ and STL interfaces, at a higher level than the C interface, may be directly used for developing parallel programs or libraries.

Supported Platforms

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

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

It has been extensively tested on the following operating systems:

- GNU-Linux 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

```
INTEGER*4 FUNCTION KAAPIF_INIT(INTEGER*4 FLAGS)
INTEGER*4 FUNCTION KAAPIF_FINALIZE()
```

2.2 Description

KAAPIF_INIT initializes the runtime. It must be called once per program before using any of the other routines. If successful, there must be a corresponding *KAAPIF_FINALIZE* at the end of the program.

2.3 Parameters

- *FLAGS*: if not zero, start only 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 KAAPIF_BEGIN_PARALLEL).

2.4 Return value

Refer to the *Error codes* section.

2.5 Example

Refer to examples/kaapif/foreach

```
PROGRAM MAIN
  INTEGER*4 E

  E = KAAPIF_INIT(1)
  ...
  E = KAAPIF_FINALIZE()

END PROGRAM MAIN
```

3 Concurrency

3.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_GET_CONCURRENCY()
INTEGER*4 FUNCTION KAAPIF_GET_THREAD_NUM()
```

3.2 Description

Concurrency related routines.

3.3 Return value

KAAPIF_GET_CONCURRENCY returns the number of parallel thread available to the X-Kaapi runtime.

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

For both functions, a negative value means an error occured. Refer to the *Error codes* section.

3.4 Example

Refer to examples/kaapif/foreach

4 Adaptive grains

4.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_SET_GRAINS
(
  INTEGER*4 PAR_GRAIN,
  INTEGER*4 SEQ_GRAIN
)
```

4.2 Description

KAAPIF_SET_GRAINS sets the adaptive loop grains. Grains are used to amortize the act of extracting work for both the parallel (ie. during a steal) and sequential (ie. during a pop) executions. Guessing those grains is problem specific. The general intuition is that they should be inversely propotionnal to a single task processing time (given a task processing time is constant across execution). The parallel grain should be greater than the sequential grain, since a steal operation requires a bigger task to be amortized.

Note that this routine sets global variables used by a subsequent calls to *KAAPIF_FOREACH* family functions. To avoid reentrancy issues, *KAAPIF_SET_GRAINS* should be called just before the corresponding *KAAPIF_FOREACH* function.

4.3 Parameters

- *PAR_GRAIN*: below this size, a task cannot be split for subsequent parallel execution. default to 32.
- *SEQ_GRAIN*: the size used by the sequential execution to extract work from its local workqueue. default to 16.

4.4 Return value

Refer to the *Error codes* section.

4.5 Example

```
PROGRAM MAIN
  INTEGER*4 E

  ...
  E = KAAPIF_INIT(1)
```

```
    E = KAAPIF_SET_GRAINS(32, 32)
    E = KAAPIF_FOREACH(...)
    ...

END PROGRAM MAIN
```

5 Performance

5.1 Synopsis

```
REAL*8 FUNCTION KAAPIF_GET_TIME()
```

5.2 Description

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

5.3 Parameters

None.

5.4 Return value

The current time, in microseconds.

5.5 Example

Refer to examples/kaapif/foreach

```
PROGRAM MAIN
  REAL*8 START
  REAL*8 STOP
  INTEGER*4 E

  E = KAAPIF_INIT(1)
  START = KAAPIF_GET_TIME()
  ...
  STOP = KAAPIF_GET_TIME()
  E = KAAPIF_FINALIZE()

  WRITE(*, *) STOP - START

END PROGRAM MAIN
```

6 Independent loops

6.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_FOREACH
(
  BODY,
  INTEGER*4 FIRST, INTEGER*4 LAST,
  INTEGER*4 NARGS,
  ...
)

INTEGER*4 FUNCTION KAAPIF_FOREACH_WITH_FORMAT
(
  BODY,
  INTEGER*4 FIRST, INTEGER*4 LAST,
  INTEGER*4 NARGS,
  ...
)
```

6.2 Description

Those routines run a parallel loop over the range [*FIRST*, *LAST*] (note this is an **inclusive** interval). The loop body is defined by *BODY* whose arguments are given in parameters. It must have the following prototype:

```
SUBROUTINE BODY(I, J, TID, ...)
```

- $[I, J]$ the subrange to process (note that interval is inclusive)
- *TID* the thread identifier

6.3 Parameters

- *BODY*: the function body to be called at each iteration
- *FIRST*, *LAST*: the iteration range indices, inclusive.
- *NARGS*: the argument count
- ...: the arguments passed to *BODY*. For *KAAPIF_FOREACH_WITH_FORMAT*, refer to the *KAAPIF_SPAWN* documentation.

6.4 Return value

Refer to the *Error codes* section.

6.5 Example

Refer to examples/kaapif/foreach
Refer to examples/kaapif/foreach_with_format

```
! computation task entry point
SUBROUTINE OP(I, J, TID, ARRAY)
  DO K = I, J
    ! process ARRAY(K)
    ...
  END DO
  RETURN
END

PROGRAM MAIN
  INTEGER*4 E

  ...
  ! apply the OP routine on ARRAY[1:SIZE]
  E = KAAPIF_FOREACH(OP, 1, SIZE, 1, ARRAY)

  ...
  ! version with format
  ! as above, ARRAY is the only argument
  !
  E = KAAPIF_FOREACH_WITH_FORMAT
  (
  ! iterated range and argument count are the same as above
  OP, 1, SIZE, 1,
  ! ARRAY an array of SIZE double
  ! ARRAY elments are read and written by the task
  ARRAY, KAAPIF_TYPE_DOUBLE, SIZE, KAAPIF_MODE_RW
  )
  ...

END PROGRAM MAIN
```

7 Dataflow programming

7.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_SPAWN
(
  BODY,
  INTEGER*4 NARGS,
  ...
)
```

7.2 Description

Create a new computation task implemented by the function *BODY*.

BODY is called with the user specified arguments, there is no argument added by XKAAPI:

```
SUBROUTINE BODY(ARG0, ARG1, ...)
```

Each task parameter is described by 4 successive arguments including:

- the argument *VALUE*,
- the parameter *TYPE*,
- the element *COUNT*,
- the access *MODE*.

TYPE is one of the following:

- KAAPIF_TYPE_CHAR=0,
- KAAPIF_TYPE_INT=1,
- KAAPIF_TYPE_REAL=2,
- KAAPIF_TYPE_DOUBLE=3.

If a parameter is an array, *COUNT* must be set to the array size. For a scalar value, it must be set to 1.

MODE is one of the following:

- KAAPIF_MODE_R=0 for a read access,
- KAAPIF_MODE_W=1 for a write access,
- KAAPIF_MODE_RW=2 for a read write access,
- KAAPIF_MODE_V=3 for a parameter passed by value.

7.3 Parameters

- *BODY*: the task body.
- *NARGS*: the argument count.
- ...: the *VALUE*, *TYPE*, *COUNT*, *MODE* tuple list.

7.4 Return value

Refer to the *Error codes* section.

7.5 Example

Refer to examples/kaapif/dfg

```
! computation task entry point
SUBROUTINE OP(A, B)
  ! task user specific code
  ...
  RETURN
END

PROGRAM MAIN
  INTEGER*4 E
  ...
  ! spawn a task implemented by the OP routine
  E = KAAPIF_SPAWN(OP, 2,
  ! argument[0]
   &                 42
   &                 KAAPIF_TYPE_DOUBLE,
   &                 1,
   &                 KAAPIF_MODE_V,
  ! argument[1]
   &                 42,
   &                 KAAPIF_TYPE_DOUBLE,
   &                 1,
   &                 KAAPIF_MODE_V)
  ...
END PROGRAM MAIN
```

8 Parallel regions

8.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_BEGIN_PARALLEL()
INTEGER*4 FUNCTION KAAPIF_END_PARALLEL(INTEGER*4 FLAGS)
```

8.2 Description

KAAPIF_BEGIN_PARALLEL and *KAAPIF_END_PARALLEL* mark the start and the end of a parallel region. Regions are used to wakeup 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. Wether threads are suspendable or not is controlled according by the KAAPIF_INIT parameter.

8.3 Parameters

- *FLAGS*: if zero, an implicit synchronization is inserted before leaving the region.

8.4 Return value

Refer to the *Error codes* section.

8.5 Example

Refer to examples/kaapif/dfg

```
PROGRAM MAIN
  INTEGER*4 E

  ...
  E = KAAPIF_BEGIN_PARALLEL()
  ...
  E = KAAPIF_END_PARALLEL(1)
  ...

END PROGRAM MAIN
```

9 Synchronization

9.1 Synopsis

```
INTEGER*4 FUNCTION KAAPIF_SYNC()
```

9.2 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.

9.3 Return value

Refer to the *Error codes* section.

9.4 Example

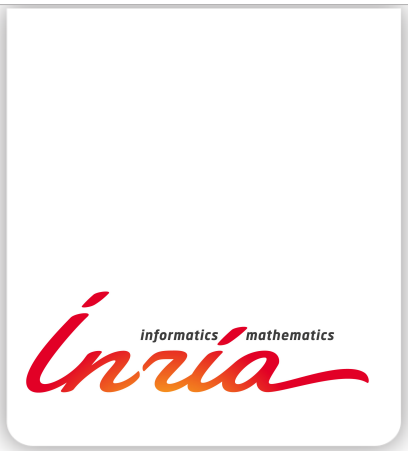
Refer to examples/kaapif/dfg

```
PROGRAM MAIN
  INTEGER*4 E

  ...
  E = KAAPIF_SYNC()
  ...

END PROGRAM MAIN
```

10 Error codes

When indicated, a routine may return one of the following error code:

- KAAPIF_SUCCESS=0: success
- KAAPIF_ERR_FAILURE=-1: generic error code
- KAAPIF_ERR_EINVAL=-2: invalid argument
- KAAPIF_ERR_UNIMPL=-3: feature not implemented



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