XKaapi

- DFG program -

09-02-2011

Thierry Gautier

Introduction

- This tutorial is part of the following tutorials
  - ✓ 0-xkaapi-intro.pdf: XKaapi: quick user's guide
  - ✓ <u>1-xkaapi-dfg.pdf: XKaapi: programming with data flow graph</u>
  - ✓ XKaapi: KaSTL API
  - ✓ XKaapi: Low Level Adaptive Application Interface
  - ✓ XKaapi: Fortran Interface
  - ✓ XKaapi: internal representation & execution

- Please refere to 0-xkaapi-intro.pdf to get and install xkaapi software

What is Kaapi ?

- C/C++ Library for parallel programming
  - ✓ Target architecture: multicore + GPU + cluster
- Ultimate goal
  - ✓ Simplify the development of parallel application
    - architecture abstraction
  - ✓ Automatic dynamic load balancing
    - theoretically & practically performances
    - **Work Stealing based algorithms**

Design

- Kernel
  - ✓ runtime for API (or compiler)
  - ✓ work stealing internal scheduling
  - ✓ C language, fine grain implementation...
- APIs for different programming models
  - ✓ Data Flow Graph: **DFG**
    - Athapascan (deprecated), Kaapi++
  - ✓ Parallel STL like: **KaSTL**
  - ✓ Adaptive Algorithms Interface: **AAI**

Source development

- http://kaapi.gforge.inria.fr
  - ✓ tarball of the master (rc04)
- GIT: ligforge
  - ✓ url = ssh://git.ligforge.imag.fr/git/kaapi/xkaapi.git
- Usage of branches
  - ✓ origin/master: the official master branch
  - ✓ origin/<username>/<branch name>: an user branch
    - The owner is responsible of its branches
    - The user **MUST ONLY** commit on its own branches
- Mailing list:
  - http://lists.gforge.inria.fr/cgi-bin/mailman/listinfo/kaapi-leaders

Installation

1. automake / autotools etc...
   - ✓ ../xkaapi/configure --help
   - ✓ ../xkaapi/configure --prefix=<totodir>
   - ✓ Usefull options:
     - --enable-mode=release for performances
     - --enable-mode=debug for more assertions in the user level API
2. Compilation
   - ✓ make
3. Installation
   - ✓ make install (in the --prefix directory, step 1)
   - ✓ <totodir>/include; <totodir>/lib etc..
     - use pkgconfig to retreive CCFLAGS, LDFLAGS etc
4. Basic check
   - ✓ make check

List of examples

- Examples sub directory
  - ✓ cd examples;
  - ✓ make examples : build all examples
  - ✓ make <prog>, e.g. make for_each_rec_xx

- hello
  - ✓ hello_world.cpp

- for_each
  - ✓ for_each_rec_kaapi++.cpp : recursive C++ version
  - ✓ for_each_0_kaapi++.cpp : basic adaptive C++ version
  - ✓ for_each_0_kaapi.c : basic adaptive C version
  - ✓ for_each_1_kaapi++.cpp : adaptive C++ version, enable steal of thief
  - ✓ for_each_2_kaapi++.cpp : idem + preemption
  - ✓ for_each_0_kaapi++_lambda.cpp : adaptive C++ version with lambda
  - ✓ for_each_kastl.cpp: call to STL kastl implementation

Cont.

- Fibo
  - ✓ fibo_kaapi.c : low level C version
  - ✓ fibo_atha.cpp : old Athapascan C++ API version
  - ✓ fibo_kaapi++.cpp : Kaapi C++ API version
  - ✓ fibo_kaapi++_opt.cpp : Kaapi C++ API with optimized task creation
  - ✓ fibo_kaapi++_cumul.cpp : Kaapi C++ API version with cumulative write
  - ✓ fibo_kaapixx_cumul_opt.cpp : Kaapi C++ API version with optimized task creation
  - ✓ fibo_kaapi++_sync.cpp: usage of synch. to avoid sum' task creation
  - ✓ make fibo_kaapi fibo_kaapi++ fibo_atha...
- NQueens
  - ✓ nqueens_atha / nqueens_kaapi++
- Cilk
  - ✓ two examples from Cilk distribution (matrix computation/qsort)
- Have a look of subdirectories in <topsrcdir>/examples !

Compilation of examples

- Use pkg-config

```
gautier@idkoiff:~$ export PKG_CONFIG_PATH=<kaapi
install dir>/lib/pkgconfig

gautier@idkoiff:~$ pkg-config --cflags kaapi++
-I/home/gautier/KAAPI/install/xkaapi/include

gautier@idkoiff:~$ pkg-config --libs kaapi++
-L/home/gautier/KAAPI/install/xkaapi/lib -lkaapi++ -
lkaapi
```

Typical use:

```
gautier@idkoiff:~$ g++ -o mytest mytest.cpp `pkg-config
--cflags kaapi++` `pkg-config --libs kaapi++`
```

That all !

Running example

- KAAPI_CPUCOUNT=1 ./fibo_kaapi++ 30

```
Fibo(30)=832040
Time: 4.326541e-01
```

- KAAPI_CPUCOUNT=2 ./fibo_kaapi++ 30

```
Fibo(30)=832040
Time: 2.143562e-01
```

- KAAPI_CPUSET=0:4,6 ./fibo_kaapi++ 30
  - use cores 0,1,2,3,4 and 6 of the machine

Sources organization

- xkaapi/src
  - ✓ everything about workstealing / graph representation is here
- xkaapi/examples
  - ✓ user level examples
- xkaapi/api
  - ✓ Athapascan C++ interface [deprecated]
  - ✓ Kaapi C++ interface
  - ✓ [Fortran interface] etc..

C++ API called Kaapi++

#include “kaapi++“

- Namespace ka::
- 3 main concepts
  - ✓ Task signature
    - declare access to data
      - Read: «R»; Write: «W»; Read Write: «RW»; Cumulative Write: «CW»
  - ✓ Task implementation
    - one implementation for each architecture
      - E.g. CPU implementation ; GPU implementation (still in progress)
  - ✓ A data is shared between 2 tasks **iff** tasks have the same pointer in effective parameters

Library initialization

- Mostly always the same:

```
int main(int argc, char** argv)
{
  try {
    /* Join the initial group of computation */
    ka::Community com = ka::System::join_community( argc, argv );

    /* Start computation by spawning the main task */
    ka::SpawnMain<doit>()(argc, argv);

    /* Leave the community */
    com.leave();

    /* */
    ka::System::terminate();
  }
  catch (const std::exception& E) {
    std::cerr << "Catch Kaapi exception: " << E.what() << std::endl;
  }
  catch (...) {
    std::cerr << "Catch unknown exception: " << std::endl;
  }
  return 0;
}
```

Task

- Task signature
  - ✓ Define the number of parameters / type / access mode of each parameter

```
/* Kaapi Hello task: print an integer n */
struct TaskHello: public ka::Task<1>::Signature<int> {};
```

- Task implementation
  - ✓ specify the implementation architecture (CPU/GPU)

```
/* CPU implementation */
template<>
struct TaskBodyCPU<TaskHello> {
               void operator() ( int n )
               {
                 std::cout << "Hello World !, n=" << n << std::endl;
               }
};
```

Task creation

- Key word: spawn

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
     /* */
     ka::Spawn<TaskHello>()( atoi(argv[1]) );
   }
};
```

- Exercise 1: write & compile HelloWorld.cpp

Task creation

- Key word: spawn

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
    /* */
    ka::Spawn<TaskHello>()( atoi(argv[1]) );
   }
};
```

- Exercise II: write all the arguments (integers)

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
    for (int i=1; i<argc; ++i)
      ka::Spawn<TaskHello>()( atoi(argv[i]) );
   }
};
```

When a task is executed ?

- Task creation is a non blocking operation:
  - ✓ the task (= function call) is pushed into a stack and the control flow continues without waiting for the termination

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
     int  a;
     int* b = ...;

     ka::Spawn<TaskThatRead_or_WriteData>()( &a, &b );

     /* here:
        1- Kaapi does not guarantee execution of the task
        2- a and b can accessed and should have a correct scope
     */
   }
};
```

When a task is executed ?

- Some guarantees:
  - ✓ A task begins its execution when all its input arguments are produced (data flow constraints)
  - ✓ The parallel execution always produces the same result as the sequential execution (up to round off)
  - ✓ At the end of the program, all created tasks have been executed
- Notion of *reference order* between tasks
  - ✓ Used to define execution order between any two tasks
    - total order
  - ✓ Semantic of Kaapi is based on it
    - originaly defined in Athapascan [Pact98]

Reference order

- Recursive definition
  - ✓ In a task body
    - created tasks are enqueued in a FIFO queue
    - each task = function call = function pointer + arguments
- When task body finish
  - ✓ the runtime dequeues each task (FIFO) and executes it

```
struct TaskBodyCPU<TaskF> {
  void operator() ( argF )
  {
    a1 = G1(..);
    ka::Spawn<TaskF1>()( argF1 );
    a2 = G2(...)
    ka::Spawn<TaskF2>()( argF2 );
    a3 = G3(...)
  }
};
```

```
execute( TaskF, argF ) =
{
  { /* task body execution */
    a1 = G1(...);
    e1 = eval( argF1 );
    enqueue( TaskF1, e1 );
    a2 = G2(...);
    e2 = eval( argF2 );
    enqueue( TaskF2, e2 );
    a3 = G3(...);
  }
  /* epilogue: execute spawned tasks */
  (TaskF1,e1) = dequeue();
  execute( TaskF1, e1 );
  (TaskF2,e2) = dequeue();
  execute( TaskF2, e2 );
}
```

Double HelloWorld (...)

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
     /* */
     ka::Spawn<TaskHello>()( atoi(argv[1]) );
     /* */
     ka::Spawn<TaskHello>()( atoi(argv[2]) );
   }
};
```

- Possible traces of execution:

```
> ./helloworld 1 2
Hello World !, n=1
Hello World !, n=2
```

Always this order iff only one core

```
> ./helloworld 1 2
Hello World !, n=2
Hello World !, n=1
```

If this order then at least two cores

How to enforce execution order?

- Cilk's like synchronisation:
  - ✓ **ka::Sync();**
  - ✓ force execution of all the spawned tasks in the current running task
- Inline data flow constraint
  - ✓ **ka::Sync(<pointer>);**
  - ✓ wait until the value pointed by the pointer is produced
- Add dependencies between tasks
  - ✓ wait to see "parameter passing rules" in 2 slides

ka::Sync() keyword

```
/* The "doit" main task */
template<class T>
struct doit {
   void operator()(int argc, char** argv )
   {
     ka::Spawn<TaskHello>()( atoi(argv[1]) );
     ka::Sync();
     ka::Spawn<TaskHello>()( atoi(argv[2]) );
   }
};
```

Enforce order

- Possible traces of execution:

```
> ./helloworld 1 2
Hello World !, n=1
Hello World !, n=2

> ./helloworld 1 2
Hello World !, n=1
Hello World !, n=2
```

Always this order

Parameter passing rules

- Parameter passing rules : way *effective parameters* are bind to *formal parameters* of task
  - by value (copy): HelloWorld.cpp
  - by reference (using ka::pointer or a C++ pointer)

- By *value*:
  - ✓ a copy is made into the task

- By *reference*
  - ✓ No copy
  - ✓ But task must declare its accesses to shared data
  - ✓ **read**: *read access*, the task can read the value
  - ✓ **write**: *write access*, a reader will see the value write
  - ✓ **read write**: <u>*exclusive access*,</u>one task has access to data
  - ✓ **cumulative write:** several write will participate to produce the final value

Task signature

- Number of parameters fixed at compilation time

```
struct TaskFibo: public ka::Task<2>::Signature<ka::W<int>, int> {};
```

- Access mode
  - ✓ ka::W<T>: write
  - ✓ ka::R<T>: read
  - ✓ ka::RW<T>: read write
  - ✓ ka::CW<T> : cumulative write with global reduction op
- Should correspond to declaration into formal parameters of task
  - ✓ ka::pointer_w<T>
  - ✓ ka::pointer_r<T>
  - ✓ ka::pointer_rw<T>
  - ✓ ka::pointer_cw<T,F>

Task

- Task signature
  - ✓ define the number of parameters / type / access mode of each parameter

```
/* Kaapi Fibo task: takes a pointer to the result + an integer n */
struct TaskFibo: public ka::Task<2>::Signature<ka::W<long>, long> {};
```

- Task implementation
  - ✓ specify the implementation architecture (CPU)

```
/* CPU implementation */
template<>
struct TaskBodyCPU<TaskFibo> {
  void operator() ( ka::pointer_w<long> res, long n ) { ... }
};
```

Dependencies

- A task must decribe their modes of access to data passed in effective parameters
  - ✓ Task Signature
- At runtime: the execution = sequence of tasks
  - ✓ *reference order* of execution
- Kaapi will always respect the following dependencies:
  - ✓ a reader will see the value written by the last task in the reference order:
    - W -> R, {CW}* -> R, or RW -> R : **R**ead **A**fter **W**rite
  - ✓ other false dependencies (**W**riter **A**fter **R**ead) may be solved by making copies of data
    - runtime decision

Cost of dependencies?

- Dependency analysis is required to execute two tasks in parallel
  - ✓ tasks with dependencies are executed following the reference order
    - “ a reader will see value writes by the last writer ”
  - ✓ tasks without dependencies may be executed in parallel
    - the runtime decide when and where 2 concurrent tasks are executed in parallel
- With work stealing scheduling:
  - ✓ execution following reference order of execution
    - dequeue management, no dependency analysis
  - ✓ dependencies are only computed during steal operation

What is really shared ?

- Two tasks share a common data IFF they access to the same data in memory

- Current implementation = limitation
  - ✓ **same data == same pointer**

```
ka::pointer<T> a;
ka::pointer<T> b = a+100;
ka::Spawn<TaskRW1>()(a); /* rw on a */
ka::Spawn<TaskRW2>()(b); /* rw on b */
```

- TaskRW1 & TaskRW2 are independent, even if access to data pointed by ‘a’ an ‘b’ overlap !!!
  - ✓ No (yet) ***region*** of memory

Illustration

- Task creation: Spawn

```
struct TaskFibo: public ka::Task<2>::Signature<ka::W<long>, long > {};

struct TaskDelete: public ka::Task<1>::Signature<ka::RW<long> > {};

struct TaskPrint: public ka::Task<1>::Signature<ka::R<long> > {};

    /* */
    ka::pointer<long> res = new long;

    /* */
    ka::Spawn<TaskFibo>()( res, n );

    /* */
    ka::Spawn<TaskPrint>()(res);

    /* */
    ka::Spawn<TaskDelete >()(res); // delete memory
```

W → R (true) dependency

R → RW (false) dependency

- The runtime automatically detects data flow dependencies between tasks
- Write after Read dependencies may be solved by copy

Fibo (bad) example

```
template<> struct TaskBodyCPU<TaskSum>
{
  void operator()( ka::pointer_w<long> r, ka::pointer_r<long> a, ka::pointer_r<long> b )
  { *r = *a + *b; }
};

template<> struct TaskBodyCPU<TaskDelete>
{
  void operator()( ka::pointer_rw<long> ptr )
  { delete ptr; }
};

template<> struct TaskBodyCPU<TaskFibo>
{
  void operator() ( ka::pointer_w<long> ptr, const long n )
  {
    if (n < 2)
      *ptr = n;
    else {
      ka::pointer<long> ptr1 = new long;
      ka::pointer<long> ptr1 = new long;

      ka::Spawn<TaskFibo>() ( ptr1, n-1 );
      ka::Spawn<TaskFibo>() ( ptr2, n-2 );

      ka::Spawn<TaskSum>() ( ptr, ptr1, ptr2 );

      ka::Spawn<TaskDelete>() ( ptr1 );
      ka::Spawn<TaskDelete>() ( ptr2 );
    }
  }
};
```

Pointer object allocation

- Scope of the data pointed by ka::pointer
  - ✓ **<u>at least</u>** the life time of the last task accessing the data
  - ✓ due to task execution order => out the C++ scope where the task is created
    - in Fibo example: new + spawn of task to delete memory
- 2 standard possibilities to manage dynamic allocations
  - ✓ use new / delete: as in the previous "fibo (bad) example"
    - 'operator new' at creation of the pointer
    - 'operator delete' -> **the user has responsability** to spawn the last task to delete the data
  - ✓ use **ka::auto_pointer**<TYPE> / scoped pointer (?)
    - 'operator new' at creation of the pointer
    - the runtime automatically spawns the task to delete the data

Fibo (better) example

```
template<> struct TaskBodyCPU<TaskSum>
{
  void operator()( ka::pointer_w<long> r, ka::pointer_r<long> a, ka::pointer_r<long> b )
  { *r = *a + *b; }
};

template<> struct TaskBodyCPU<TaskFibo>
{
  void operator() ( ka::pointer_w<long> ptr, const long n )
  {
    if (n < 2)
      *ptr = n;
    else {
      ka::auto_pointer<long> ptr1 = new long;
      ka::auto_pointer<long> ptr2 = new long;

      ka::Spawn<TaskFibo>() ( ptr1, n-1 );
      ka::Spawn<TaskFibo>() ( ptr2, n-2 );

      ka::Spawn<TaskSum>() ( ptr, ptr1, ptr2 );
    }
  }
};
```

How to improve fibo ?

- Problem1:
  - ✓ Huge number of new / delete at runtime
    - sequential C++ version: automatic variable in the stack for intermediate sub results
    - ➡ use '**ka::auto_variable**' to declare variable in the Kaapi stack of tasks
      - very similar to sequential C++ automatic variable
    - stdlib.h / alloca <=> sequential C++ automatic variable
    - **ka::Alloca <=> ka::auto_variable**

- Illustration on Fibonacci

Fibo (much better) example

```
template<> struct TaskBodyCPU<TaskSum>
{
  void operator()( ka::pointer_w<long> r, ka::pointer_r<long> a, ka::pointer_r<long> b )
  { *r = *a + *b; }
};

template<> struct TaskBodyCPU<TaskFibo>
{
  void operator() ( ka::pointer_w<long> ptr, const long n )
  {
    if (n < 2)
      *ptr = n;
    else {
      ka::auto_variable<long> res1;
      ka::auto_variable<long> res2;

      ka::Spawn<TaskFibo>() ( &res1, n-1 );
      ka::Spawn<TaskFibo>() ( &res2, n-2 );

      ka::Spawn<TaskSum>() ( ptr, &res1, &res2 );
    }
  }
};
```

Same with “alloca”

```
template<> struct TaskBodyCPU<TaskSum>
{
  void operator()( ka::pointer_w<long> r, ka::pointer_r<long> a, ka::pointer_r<long> b )
  { *r = *a + *b; }
};

template<> struct TaskBodyCPU<TaskFibo>
{
  void operator() ( ka::pointer_w<long> ptr, const long n )
  {
    if (n < 2)
      *ptr = n;
    else {
      ka::pointer<long> ptr1 = ka::Alloca<long>(1);
      ka::pointer<long> ptr2 = ka::Alloca<long>(1);

      ka::Spawn<TaskFibo>() ( ptr1, n-1 );
      ka::Spawn<TaskFibo>() ( ptr2, n-2 );

      ka::Spawn<TaskSum>() ( ptr, ptr1, ptr2 );
    }
  }
};
```

How to improve fibo ?

- Problem2:
  - ✓ Two many tasks spawned at each recursion level:
    - one of the recursive spawn may be inline by sequential call

```
template<> struct TaskBodyCPU<TaskFibo>
{
  void operator() ( ka::pointer_w<long> ptr, const long n )
  {
    if (n < 2)
      *ptr = n;
    else {
      ka::auto_variable<long> res1;
      ka::auto_variable<long> res2;

      ka::Spawn  <TaskFibo>() ( &res1, n-1 );
      TaskBodyCPU<TaskFibo>() ( &res2, n-2 );

      ka::Spawn<TaskSum>() ( ptr, &res1, &res2 );
    }
  }
};
```

Difference between seq call

  - ✓ Cost of this example over pure C++ sequential version
    - one of the recursive spawn may be inline by sequential call

Conclusion about “Fibo”

1. This is not the best way to compute the N-th Fibonacci number
2. All previous presented codes are in the examples/fibo directory
   - `fibo_kaapi++.cpp`: 1rst version with new + TaskDelete
   - `fibo_kaapi++_autopointer.cpp`: with ka::auto_pointer
   - `fibo_kaapi++_autovar.cpp`: with ka::auto_variable
   - `fibo_kaapi++_alloca.cpp`: with ka::Alloca
   - `fibo_kaapi++_opt.cpp`: with one seq. call and one spawn
3. Other variations in the same directory
   - cumulative write, using ka::sync,
   - and with optimization to take into account the current running thread when tasks are spawned

Pointer object allocation

- Resume of previous slides:

```
/* using heap allocation => destruction by the user */
ka::pointer<float> ptr = new float[MAX];

/* using heap allocation => destruction by the runtime */
ka::auto_pointer<float> ptr = new float[MAX];

/* taking a reference to an global application data */
ka::pointer<float> ptr = &big_application_vector;

/* using Kaapi stack allocation: WARNING limited ressource */
ka::auto_variable<float> var;

/* using Kaapi stack allocation: WARNING limited ressource*/
ka::pointer<int> ptr = ka::Alloca<int>(1);
```

✓ **Pointer arithmetics**

Allowed operations on pointer

- Let T any type
- ka::pointer_w<T> ptr;
  - ✓ left value / assignment: ”*ptr = ...”
- ka::pointer_r<T> ptr;
  - ✓ right value ~ T*: ”std::cout << *ptr;”
- ka::pointer_rw<T> ptr;
  - ✓ left value / assignment: ”*ptr = ...”
  - ✓ right value ~ T*: ”std::cout << *ptr;”
- ka::pointer<T> ptr;
  - ✓ constructor with T* value ”ka::pointer<T> ptr = new T;”
  - ✓ assignment to T* : ”ptr = new T;”
- ka::pointer_cw<T,F> ptr;
  - ✓ left value : ”*ptr += ...”
  - ✓ assumed to be associative

C++ pointer arithmetic

- Increment / decrement by interger
- Comparizon
- Difference of pointers
- Array access

Restriction on passing references

- Let e an effective reference of type ka::pointer_XX
- Let *f* an formal parameter of type ka::pointer_YY
- The following is allowed:

effective \ formal	pointer_r	pointer_rw	pointer_w	pointer_cw
pointer	yes	yes	yes	yes
pointer_r	yes	no	no	no
pointer_rw	no	no	no	no
pointer_w	no	no	no	no
pointer_cw	no	no	no	yes

Restriction on passing references

effective \ formal	pointer_r	pointer_rp	pointer_rw	pointer_rpwp	pointer_w	pointer_wp
pointer	yes	yes	yes	yes	yes	yes
pointer_r	yes	yes	no	no	no	no
pointer_rp	yes	yes	no	no	no	no
pointer_rw	no	no	no	no	no	no
pointer_rpwp	yes	yes	yes	yes	yes	yes
pointer_w	no	no	no	no	no	no
pointer_wp	no	no	no	no	yes	yes

Extension: terminal recursion

formal / effective	pointer_r	pointer_rp	pointer_rw	pointer_rpwp	pointer_w	pointer_wp
pointer	yes	yes	yes	yes	yes	yes
pointer_r	yes	yes	no	no	no	no
pointer_rp	yes	yes	no	no	no	no
pointer_rw	no	no	yes	no	yes	no
pointer_rpwp	yes	yes	yes	yes	yes	yes
pointer_w	no	no	no	no	yes	no
pointer_wp	no	no	no	no	yes	yes

Terminal recursive call

for_each_rec_xx.cpp

- Recursive for_each on an array [beg,end)
  - ✓ STL approach

- Recursive task

```
/* task signature */
template<typename T, typename OP>
struct TaskForEach : public ka::Task<3>::Signature<ka::RPWP<T>, ka::RPWP<T>, OP>
{};

/* CPU implementation */
template<typename T, typename OP>
struct TaskBodyCPU<TaskForEach<T, OP> > {
  void operator() ( ka::pointer_rpwp<T> beg, ka::pointer_rpwp<T> end, OP op)
  {
    if (end-beg < 2)
      ka::Spawn<TaskForEachTerminal<T,OP> >()( beg, end, op );
    else {
      int med = (end-beg)/2;
      ka::Spawn<TaskForEach<T,OP> >()( beg, beg+med, op );
      ka::Spawn<TaskForEach<T,OP> >()( beg+med, end, op );
    }
  }
};
```

for_each_rec_xx.cpp

- Recursive for_each on an array [beg,end)
  - ✓ STL approach

- Terminal task

```
/* task signature */
template<typename T, typename OP>
struct TaskForEachTerminal :
      public ka::Task<3>::Signature<ka::RW<T>, ka::RW<T>, OP> {};

/* CPU implementation */
template<typename T, typename OP>
struct TaskBodyCPU<TaskForEachTerminal<T, OP> > {
  void operator() ( ka::pointer_rw<T> beg, ka::pointer_rw<T> end, OP op)
  {
    std::for_each( beg, end, op );
  }
};
```

! RpWp => RW required to access to data

Some optimizations

Better task creation

- Tasks are pushed into the current thread stack
  - ✓ avoid access to the "*current thread*"

1. keep the same signature

```
/* Kaapi Fibo task: takes a pointer to the result + an integer n */
struct TaskFibo: public ka::Task<2>::Signature<ka::W<int>, int> {};
```

2. add an extra / optional formal parameter in the implementation

```
/* CPU implementation */
template<> struct TaskBodyCPU<TaskFibo> {
  void operator() ( ka::Thread* thread,
                    ka::pointer_w<int> res, int n ) { ... }
};
```

3. spawn using the thread: **thread->**

```
/* Recursive calls in Fibonacci */
thread->Spawn<TaskFibo>()( res1, n-1 );
thread->Spawn<TaskFibo>()( res2, n-2 );
thread->Spawn<TaskSum>() ( res, res1, res2 );
```

Passing big value

- Value = effective parameter is copied 2 times
  - ✓ to the internal task argument: Spawn
  - ✓ to the user function: function call

```
/* Stupid task */
struct TaskFAT: public ka::Task<2>::Signature<ka::W<int>, Matrix> {};
template<> struct TaskBodyCPU<TaskFAT> {
  void operator() ( ka::pointer_w<int> res, Matrix M ) { ... }
};
ka::Spawn<TaskFAT>()( smallint, bigmatrix );
```

- Use const T& declaration: copied once

```
/* Not so stupid task */
struct TaskFAT: public ka::Task<2>::Signature<ka::W<int>, Matrix> {};
template<> struct TaskBodyCPU<TaskFAT> {
  void operator() ( ka::pointer_w<int> res, const Matrix& M ) { ... }
};
ka::Spawn<TaskFAT>()( smallint, bigmatrix );
```

Passing big value cont'd

- If not enough → use pointer

```
/* not stupid task */
struct TaskFAT:
   public ka::Task<2>::Signature<ka::W<int>,
                                 ka::W<Matrix> > {};
template<> struct TaskBodyCPU< TaskFAT > {
  void operator() ( ka::pointer_w<int> res, ka::pointer_w<Matrix>M )
  { ... }
};

ka::Spawn<TaskFAT>()( smallint, &bigmatrix );
```

- overhead: same as using a C++ pointer
- the user should consider the life data

http://kaapi.gforge.inria.fr

Kaapi is a software developped at
http://moais.imag.fr