Report: KACC automatic loop parallelisation

XKAAPI team

Contents

1 Introduction

XKAAPI is a runtime exporting a set of functions to create and schedule computation tasks. Scheduling relies on the workstealing paradigm where a processor steals computation tasks as it becomes idle. Among the programming models, XKAAPI allows for adaptive tasks creation, ie. tasks that are created on demand by splitting a work set.

Programming adaptive tasks is not easy since the exported interface requires knowledge about the runtime inner workings. As a result, we implemented automatic loop parallelisation in the KACC compiler, that only requires the user to annotate the corresponding code with *#pragma* directives.

This report details the current implementation status, and can be used as a base for futur work. Benchmarks and example applications are also included. It concludes by giving ingishts on issues that remain to be addressed and directions for futur studies.

2 Automatic loop parallelisation

2.1 Loop structure constraints

2.1.1 General form

A classical C/C++ *for* loop construct has the following form:

```
for (initialization; condition; increment) body ;
```

With some constraints, this is the kind of loop automatic parallelisation is applied to. A user informs KACC to parallelise a loop by annotating it with a *#pragma kaapi loop* directive:

```
#pragma kaapi loop
for (initialization; condition; increment) body ;
```

2.1.2 Initialization clause

The *initialization* clause is a standard statement. Note that it can be outside the loop:

```
initialization;
#pragma kaapi loop
for (; condition; increment) body ;
```

2.1.3 Condition clause

A valid clause is a testing statement of the form:

```
lhs binary_operator rhs
```

An *binary_operator* is valid if it belongs to the following list:

$$<, <=, >, >=, !=$$

Note that the corresponding C++ operators are considered valid.

As an arbitrary choice, the *condition* left hand side is always defined to be the loop *iterator* name. There is currently no way to override this.

2.1.4 Increment clause

The *increment* clause is a list of one or more variable updating expressions. Note that there should at least be one expression updating the *iterator* variable. An expression is valid if it has one of the following form:

```
var = var op rhs;
var += rhs; // eq. var -= rhs
++var; // eq. --var
var++; // eq. var--
```

For instance, below is a valid construct:

```
#pragma kaapi loop
for (initialization; condition; i += 2, j -= 4, ++k) body ;
```

The set of *affine* variables is built by gathering all the updated variables. In the above example, the set is thus $\{i, j, k\}$. It is important to note that a variable in this set cannot be updated by the loop body.

2.1.5 Iteration direction

The iteration can be *forward* or *backward*, defining its direction. The direction is important since it is used to determine the iteration *range*. An increasing (resp. decreasing) operator in the *increment* clause means a *forward* (resp. *backward*) iteration.

```
/* ++ is an increasing operator, forward iteration */
for (initialization; condition; ++i) body ;

/* -= is a decreasing operator, backward iteration */
for (initialization; condition; i -= 2) body ;
```

2.1.6 Iteration interval

The iteration *count* is always defined by the formula:

$$iteration\_count = \left\lceil \frac{hi\_bound - lo\_bound}{iterator\_increment} \right\rceil$$

where:

- *initial_value* is the iterator value upon loop entry,
- if this is a *forward* iteration, *hi_bound* is set to the *condition* clause right hand side and *lo_bound* is set to *initial_value*,

- if this is a *backward* iteration, *hi_bound* is set to the *initial_value* and *lo_bound* is set to the *condition* clause right hand side,
- 1 is added according to the *condition* strictness.

Thus the iteration spans the following interval:

$$\Big[initial\_value, initial\_value + iteration\_count\Big[$$

2.2 Supported pragma clauses

To drive KACC during loop parallelisation, a set of clauses can be append to the *#pragma kaapi loop* directive. Here is the list of currently supported clauses:

- *reduction(reduction_identifier: variable_name , ...)*: it works similarly to DFG task reduction. Refer to the compiler manual for more information.

2.3 Notes

A current important limitation is that none of the variables gets updated at the end of the loop, with the exception of variables present in reduction clauses. A missing *output* clause should be implemented.

There is implicitely assumed that the working sequences have a random access iterators semantic.

2.4 Examples

3 source code listings are provided as examples:

- *for_each*: apply a function to the elements of a sequence,
- *accumulate*: sum all the sequence elements into a scalar,
- *inner_product*: compute the dot product of 2 vectors.

Listing˜1: for_each

```
/* iterate using an index
 */
static void for_each(double* v, unsigned int n)
{
  unsigned int i;
#pragma kaapi loop
  for (i = 0; i < n; ++i) v[i] += 1;
}

/* iterate using the pointer
 */
static void for_each(double* v, unsigned int n)
{
#pragma kaapi loop
  for (; n > 0; --n, ++v) *v += 1;
}
```

Listing˜2: accumulate

```
/* define a reduction function
 */
static void reduce_sum(double* lhs, const double* rhs)
{
  *lhs += *rhs;
}

/* associate an id to the reduction function
 */
#pragma kaapi declare reduction(reduce_sum_id: reduce_sum)

/* accumulate algorithm using a reduction clause
 */
static double accumulate(const double* v, unsigned int n)
{
  unsigned int i;
  double sum = 0;
#pragma kaapi loop reduction(reduce_sum_id:sum)
  for (i = 0; i < n; ++i) sum += v[i];
  return sum;
}
```

Listing˜3: inner_product

```
/* define a reduction function
 */
static void reduce_sum(double* lhs, const double* rhs)
{
  *lhs += *rhs;
}

/* associate an id to the reduction function
 */
#pragma kaapi declare reduction(reduce_sum_id: reduce_sum)

/* accumulate algorithm using a reduction clause
 */
static double inner_product(const double* u, const double* v, unsigned int n)
{
  double sum = 0;
#pragma kaapi loop reduction(reduce_sum_id:sum)
  for (; n > 0; --n, ++u, ++v) sum += (*u) * (*v);
  return sum;
}
```

3 Synthetic benchmark: sequence iteration

Figure 1 shows the speedup of a program applying a given function to all the elements of a sequence, such as:

$$v[i] = f_{nn}(v[i]);$$

The applied function basic operation count, *NN*, is indicated as a subscript in the graph legend.

Information about the experiment:

- The host is IDFREEZE,
- The command line uses *numactl –interleave=all*,
- The speedup is computed relatively to the sequential program,
- The sequence size is 4 * 1024 * 1024 double.

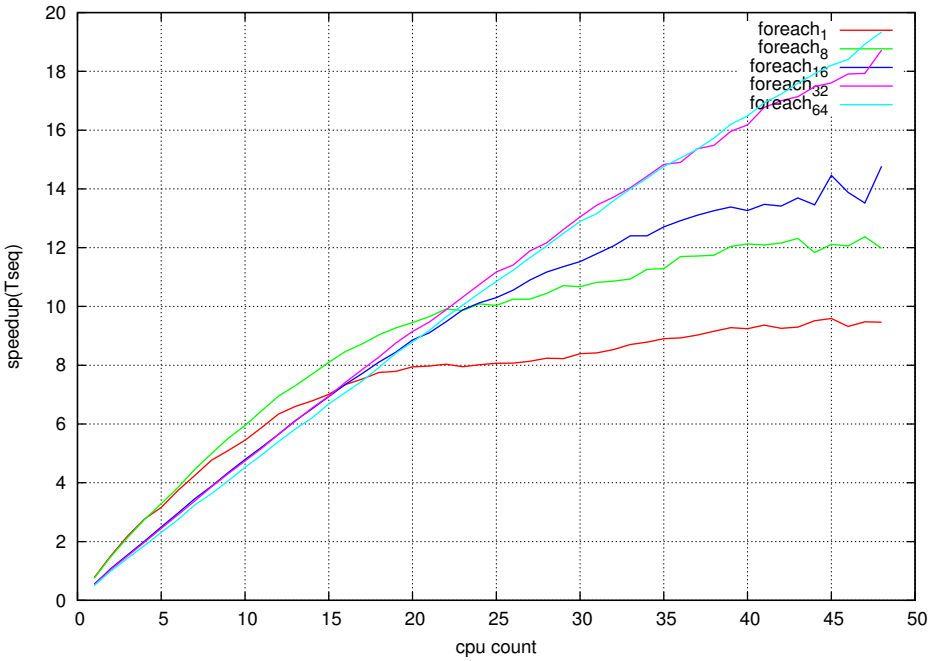

Figure 1: $foreach_{NN}$ where NN the operation count

As we can see, $\frac{Tseq}{T1}$ is low and decreases as the operation count increases. Without entering the details, this is due to the generated machine code, and has only very little to do with runtime related overheads. It thus seems important to have a look at the speedup for the same experience, but measured against the parallel program with 1 processor. This is shown in Figure 2.

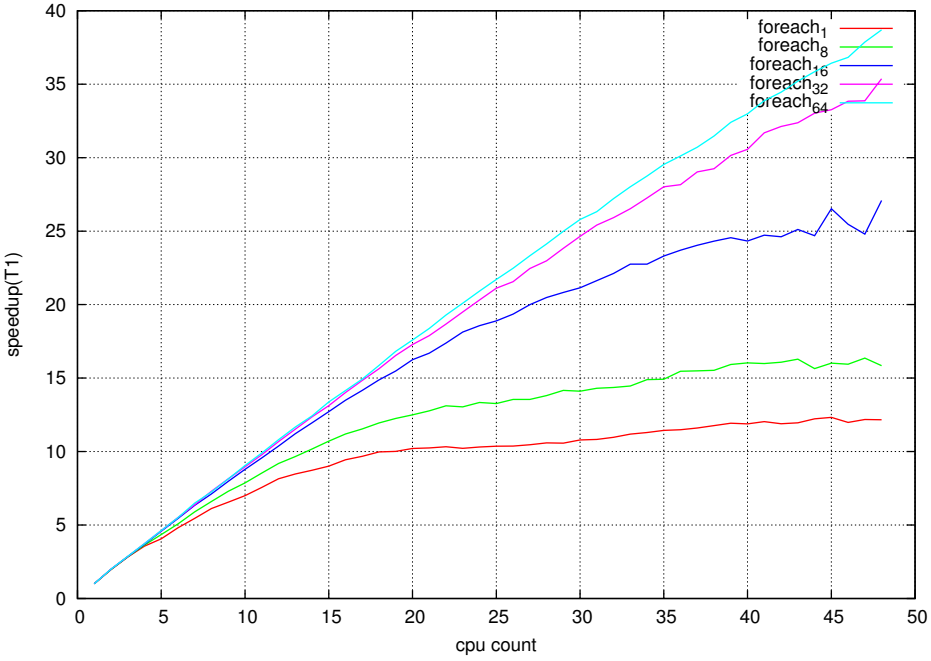

Figure 2: $foreach_{NN}$ where NN the operation count

4 Synthetic benchmark: sequence accumulation

Figure 3 shows the speedup of a program accumulating the elements of a sequence into a single reduction variable, such as:

$$\sum_{i=0}^{n} v[i] * v[i] * ... * v[i]$$

The multiplication count, *NN*, is indicated as a subscript in the graph legend.

Information about the experiment:

- The host is IDFREEZE,
- The command line uses *numactl –interleave=all*
- The speedup is computed relatively to the sequential program,
- The sequence size is 4 * 1024 * 1024 double.

As expected, we see that the program scales more as the operation count increases (ie. increasing arithmetic over memory operation ratio).

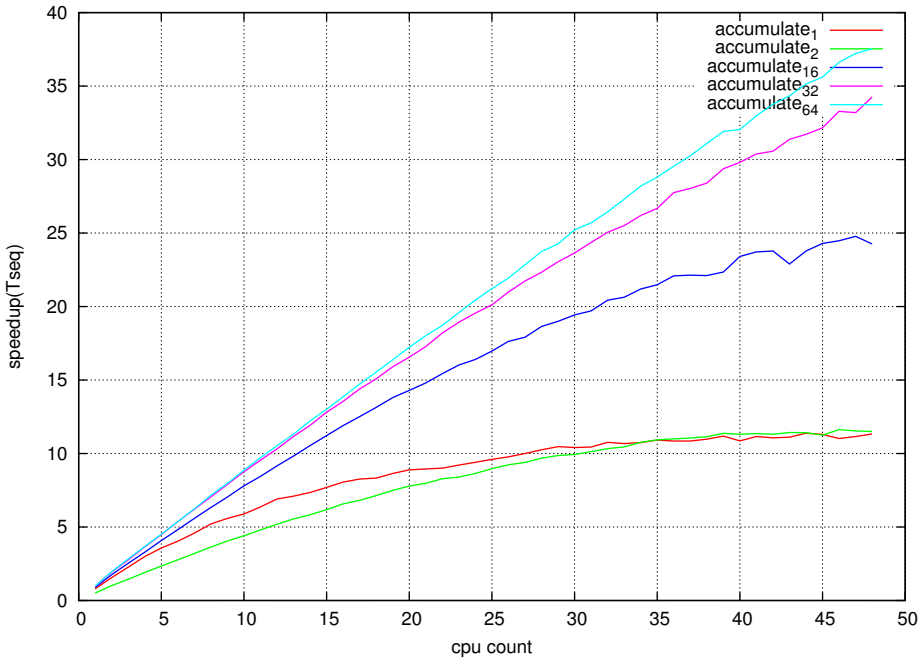

Figure 3: $accumulate_{NN}$ where NN the operation count

Note that the sequential program performs better than the parallel one 1 core. This penalty is inversely proportionnal to the operation count per sequence element, and is non negligeable if the count is low. We emphasis that the XKAAPI runtime overhead is minor in this performance issue. Here are the reasons:

- Analysing the generated assembly code shows that a less efficient register usage is done, resulting in more memory operation per iteration. This has a dramatic impact on performance if the operation count is low.
- KACC adds one arithmetic operation per iteration of a parallel loop.
- Extracting from the XKAAPI workqueue incurs overhead. In this benchmark, we should note that work extraction costs are largely amortized.

5 OpenDE library

5.1 Overview

OpenDE is a rigid body dynamics simulation opensource library. The website is available at:

http://www.ode.org

This is a real world application not written with parallelism in mind. The C/C++ code does not make a use of heavy or complicated syntaxic constructs, and is easily parsed by the source to source framework. Most of the *for* loop structures are suitable for automatic parallelisation. Thus, it is a good candidate to apply our work on.

A GIT repository tracking original code modifications is available here:

https://github.com/texane/opende_kaapi

5.2 Preprocessing step

In this work, we focused on the *step.cpp* file. It contains a function *dInternalStepIsland_x2* that gets called at each simulation time step. This function is divided into several smaller steps. Among other things, those steps perform one or more iterations on the body vector.

In the example used for the benchmark, the *preprocessing* step consumes nearly 37% of an iteration time. From now on, we only focus on it. It consists of the following pseudocode:

```
foreach (body)
{
  body->tag = i;
}

foreach (body)
{
  compute_inverse_inertia_tensor(body);
  if (body->isGiroscopic)
  {
    compute_inertia_tensor(body);
    compute_rotational_force(body);
  }
}

foreach (body)
{
  add_gravity(body);
}
```

5.3 Benchmarks

Figure 4 shows the speedup of the *preprocessing* step of the *demo_step* application shipped with OpenDE. The code can be retrieved on the *opende_kaapi* repository, commit:

bb96672ada398f0f7e39d03a091d8195133895f3

Information about the experiment:

- The host is IDKOIFF (OpenGL related problems on IDFREEZE),
- The command line uses *numactl –interleave=all*,
- The speedup is computed relatively to the sequential program,

Figure 4 shows that the speedup is limited. 3 reasons may explain that:

- memory cache related issues,
- workstealing engine issues,
- the problem size is too small.

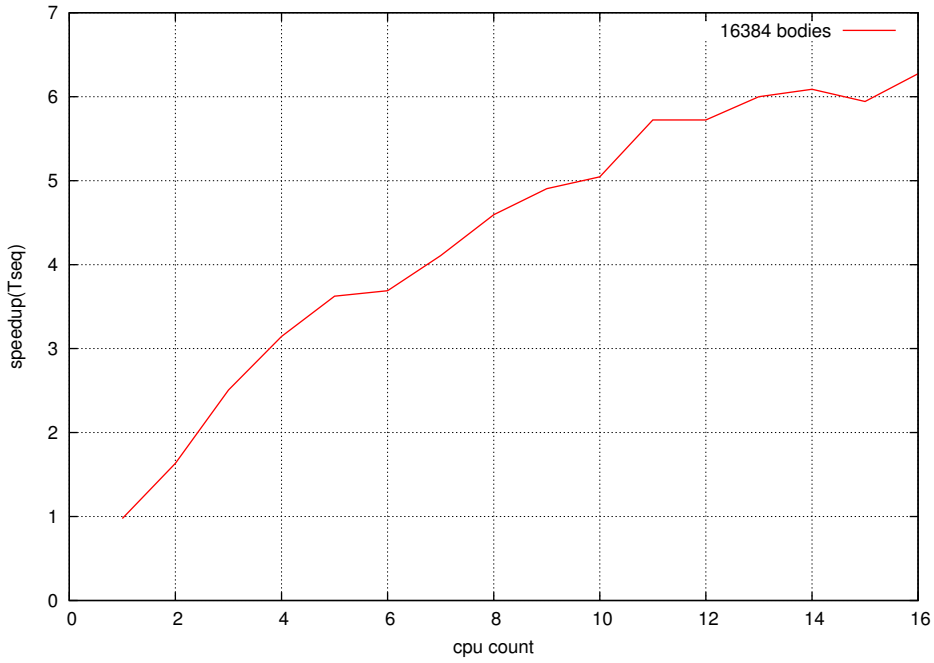

Figure 4: opende speedups

Since XKAAPI does not yet handle memory affinity, a work interval is unlikely to be distributed to the core that worked on it in the previous algorithm. Consequently, having multiple consecutive loops instead of one may have cache related effect. More importantly, the probability for a processor P_i to steal a memory word M_j decreases as the processor count increases. Thus, cache effects are related to the processor count.
To verify this claim, we could have increased the rigid body count to make the stolen interval bigger than the cache. Unfortunately, OpenDE crashes as we allocate larger body vectors.
We chose a different approach: merging the different loops into a single one, such that the *preprocessing* step pseudo code becomes:

```
foreach (body)
{
  body->tag = i;
  compute_inverse_inertia_tensor(body);
  if (body->isGiroscopic)
  {
    compute_inertia_tensor(body);
    compute_rotational_force(body);
  }
  add_gravity(body);
}
```

The resulting speedup graph is shown in Figure 5

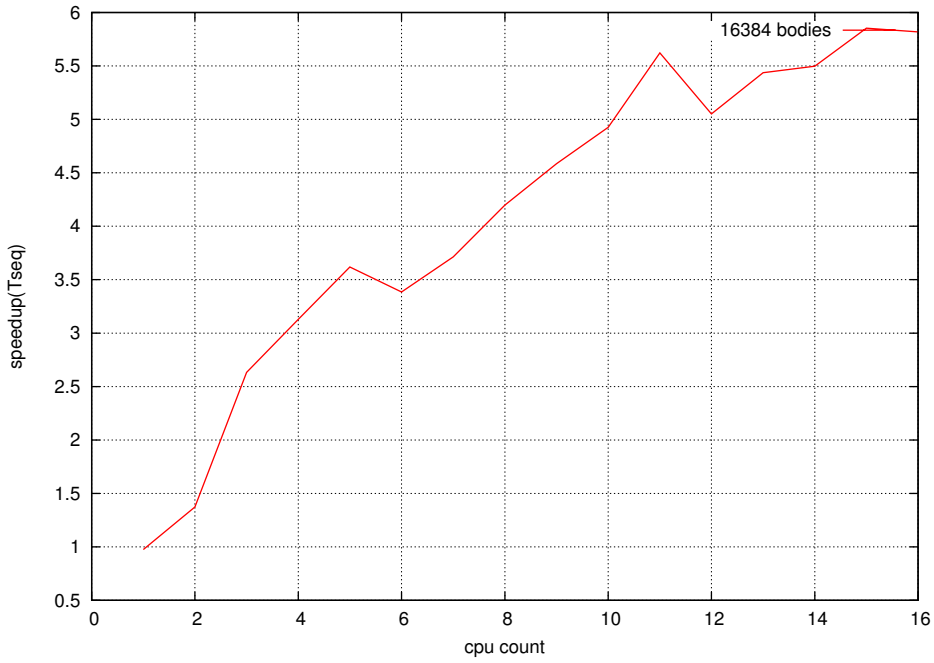

Figure 5: single loop speedup

This graph clearly invalids our claim regarding cache effects.

Another explanation could lie in the workstealing engine. However, we did not observe any incorrect behavior in the previous synthetic experiments. We thus conclude that the speedup limiting factor is the problem size.

5.4 Future work

Due to the lack of time, no significant progress has been made on the LCP solver. As the number of joints and contacts increase in the scene, the time spent solving constraint equations becomes largely dominant. The code is entierly written by hand, as are all the matrix related operations. Thus, it is not possible to use a parallel linear algebra library.

6 Conclusion and futur work

6.1 Conclusion

Automatic loop parallelisation can be useful to parallelize existing sequential applications with relatively few code modifications. There are still strong constraints on the loop structure, as well as performance issues on strongly memory bound, iterative applications. The remaining subsections list a set of directions to improve the current work.

6.2 Automatic grain tuning

There is a size below which the problem should not be splitted anymore, since the runtime costs become more important than the actual task processing ones. Currently, a threshold is set constant and large enough to armotize the runtime related costs for most problems. This code can be found in:

$XKAAPI/src/misc/kaapi_splitter.c

While it works on the tested problems, a constant threshold relates to a size which is independant of the actual task processing costs. It may prevent a small sized task to be split, while it actually has large processing costs from which parallelism could be extracted. Some work is needed to make the split criteria more flexible.

6.3 Memory affinity splitter

Currently, strongly memory bound iterative applications suffer from performance issues. There are already works in progress to address memory affinity related problems. A binding attribute is present but not yet exploited in the XKAAPI runtime. Recently, Damien Leone worked on splitting policy that conserve memory affinity information over loop iteration.

6.4 Missing loop pragma clauses

Currently, the compiler automatically guesses the loop parameters as long as the loop is expressed in a suitable form. There is yet no way for the user to override the compiler decisions. To do so, a set of pragma clauses should be added, for instance:

- *affine(variable_names)*: defines a set of variables which are considered affine
- *output(variable_names)*: defines a set of variables which must be updated at the end of the loop. Currently, no variable is updated as a loop terminates.

- *iterator(variable_name)*: defines which variable drives the loop execution.

6.5 Extend to multi level loops

Currently, KACC only targets one level loops. This is an obvious limitation for applications working on multidimensionnal structures, such as image processing or linear algebra. In the case of image processing kernels, a solution is to linearize the 2d domain.

6.6 Atomic pragma clause

Even if this can leads to scalability issues, atomic operations can be useful when parallelising a sequential program. An *atomic* directive could be added before statements or variable declarations. Note that the implementation should not be constrained by the hardware atomic instruction availability or applicability. For instance, if the target variable is not a machine word, a lock or even a critical section may be used to implement atomicity.

6.7 FORTRAN language support

KACC will eventually be used by projects written in FORTRAN (ie. EUROPLEXUS). While the ROSE source to source framework support this language, KACC has not yet been tested on Fortran source code files.

6.8 Large project support (NOT RELATED TO ADAPTIVE LOOPS)

There are some limitations in the source to source compilation framework that prevent large C++ projects to be compiled by KACC (ie. SOFA). The limitations include namespace resolution and performance issue, making the compilation process ways too slow. Those problems need to be solved.

6.9 KACC driver integration (NOT RELATED TO ADAPTIVE LOOPS)

Ideally, we would like to replace a project compiler (ie. GCC) by KACC. It currently in most cases, but features are still missing. For instance, we would like to filter the KACC processed files, by adding a *.kacc_filter* file list in each subdirectories.