XKaapi : a runtime for highly parallel (OpenMP) application

Thierry Gautier
thierry.gautier@inrialpes.fr
MOAIS, INRIA, Grenoble
Agenda

- context
- objective
- task model and execution
- status
Computer

processor + memory + accelerator
Processor architecture

- multicore
- caches

4-cores Haswell

1 core

4-cores AMD Barcelona

6-cores AMD Istanbul

Intel Xeon Phi, 60 cores!
Trends

- many-many cores
- cache hierarchies
- hybrid
  - AMD Fusion, Nvidia

Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yelick
• 4 sockets Intel Xeon Phi machine?
  ‣ ~ 4 * 72 = 288 cores
  ‣ ~ 4*4*72 = 1152 hyperthreads
  ‣ ~ 4 * 3 Tflop/s
  ‣ what about memory coherency... ?

~ 2015 socket version

No more offloading
Direct access to main memory
How to program them?

**Top-down classification**

- **network**
  - MPI, PGAS (UPC), X10

- **multi-core**
  - OpenMP, X10, Cilk, Intel TBB, XKaapi, StarPU...

- **accelerator**
  - Cuda, OpenCL, OpenACC, OpenMP (4.0)

- **vecteur / unité SIMD**
  - compilateur, type étendu, OpenMP (4.0)

**Which parallel programming environment?**

- **OpenMP-4.0**
  + OpenCL?
  + Cuda?

**Challenge**: same programming model, same code.
Challenge: performance portability!

- **Performance portability**
  - on several generations of architecture
  - with load balancing issues
    - OS jitter, irregular application

- **MOAIS team promotes a 2 steps approach**
  1. parallel algorithms
    - communications!
  2. scheduling as a plugin

- **Requirement of high level runtime features**
Which runtime features?

• Programming model pressure
  ‣ Task (independent and dependent)
    - extension
  ‣ Parallel loop

• Architecture pressure
  ‣ High degree of parallelism
  ‣ Memory hierarchy

• Performance guarantee
  ‣ parallel algorithms
  ‣ scheduling
    - work stealing, HEFT, Dual approximation
    - …

➡ Management of many-(many) tasks
  ‣ parallel slackness
  ‣ spatial and temporal locality
  ‣ scalable internal algorithms and data structure
**XKaapi task model**

- **Athapascan task model**
  - description of data dependencies (read/write/accumulation)

- **Using OpenMP code annotation**

  ```c
  void main()
  {
    /* data result is produced */
    compute( input, &result );

    /* data result is consumed */
    display( &result );
  }
  ```

- **Task**
  - OpenMP structured block, function call
  - assumption: no side effect, description of access mode

- **~ Sequential semantics**
Limitation

- Size of the graph!
- Two complementary approaches
  - fine grain task management: XKaapi task creation ~ 10cycles / task
  - compact dag representation for symbolic scheduling (DAGuE [UTK]), lazy task creation

```c
#include <cblas.h>
#include <clapack.h>

void Cholesky( int N, double A[N][N], size_t NB )
{
    for (size_t k=0; k < N; k += NB)
        #pragma omp task depend(inout: A[k:NB][k:NB]) shared(A)
            clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
        #pragma omp task depend(in: A[k:NB][k:NB]) \
            depend(inout: A[m:NB][k:NB]) shared(A)
            cblas_dtrsm( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );

    for (size_t m=m+ NB; m < N; m += NB)
        #pragma omp task depend(in: A[m:NB][k:NB]) \
            depend(inout: A[m:NB][m:NB]) shared(A)
            cblas_dsyrk( CblasRowMajor, CblasLower, CblasNoTrans,
                         NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

    for (size_t n=k+NB; n < m; n += NB)
        #pragma omp task depend(in: A[m:NB][k:NB], A[n:NB][k:NB])\ 
            depend(inout: A[m:NB][n:NB]) shared(A)
            cblas_dgemm( CblasRowMajor, CblasNoTrans, CblasTrans,
                         NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
}

#pragma omp taskwait
```
Adaptive Task model

- **On-demand task creation**
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

- **Typical example : parallel loop**
  - A task == a range of iterations
  - Initially, one task in charge of the whole range
Adaptive Task model

• On-demand task creation
  ‣ Adaptive tasks can be split at run time to create new tasks
  ‣ Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

• Typical example : parallel loop
  ‣ A task == a range of iterations
  ‣ Initially, one task in charge of the whole range

T1 : [0 - 15]
Adaptive Task model

- **On-demand task creation**
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

- **Typical example : parallel loop**
  - A task == a range of iterations
  - Initially, one task in charge of the whole range
Adaptive Task model

• On-demand task creation
  › Adaptive tasks can be split at run time to create new tasks
  › Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

• Typical example: parallel loop
  › A task == a range of iterations
  › Initially, one task in charge of the whole range

T1 : [0 - 15]
Adaptive Task model

- **On-demand task creation**
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

- **Typical example: parallel loop**
  - A task == a range of iterations
  - Initially, one task in charge of the whole range

---

Diagram:

- Idle Core
- Idle Core
- Aggregate steal requests
- split \((T1, \text{nb-stealers}+1)\)
- \(T1: [0 - 15]\)
Adaptive Task model

- **On-demand task creation**
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

- **Typical example : parallel loop**
  - A task == a range of iterations
  - Initially, one task in charge of the whole range

![Diagram of adaptive task model](image-url)
Adaptive Task model

- **On-demand task creation**
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a «splitter» function called by idle cores on running task
    - steal part of the remaining computation

- **Typical example : parallel loop**
  - A task == a range of iterations
  - Initially, one task in charge of the whole range

![Diagram of Adaptive Task model with task creation and split mechanisms]
The way XKaapi executes tasks

• Work-stealing based scheduling
  ‣ One “worker thread” per core
    - Holds a queue of tasks

• push = Task creation is cheap!
  ‣ Reduces to pushing C function pointer + its arguments into the worker thread queue
    - ~ 10 cycles / tasks on AMD Many Cours processors
  ‣ Recursive tasks are welcome

• pop = Task execution by the owner is cheap!
  ‣ pop the next task in the queue without computing dependencies
    - sequential execution is a valid order

• steal = the most costly operation
  ‣ Compute ready tasks due dependencies
    - Cilks’ work first principle!
Steal operation

• Theoretical basis
  ‣ Work first principle: if application is highly parallel, steal operations are rare
  ‣ Move overhead from the work to the critical path

• Adaptive task

• Detection of ready task
  ‣ visit all tasks in a work queue following sequential order of creation
    - for each task, visit all data accesses to detect dependencies with previous accesses
  ‣ detection of dependencies can be cached between steal operations
    - allows to build the data flow graph for HEFT or GDUAL scheduler
    - new implementation in XKaapi-3.0, on going work

• Basic work stealing protocol
  ‣ idle core locks the victim queue

• Extension: work stealing requests aggregation
  ‣ idle core steals tasks for itself and all waiting cores (on the same victim)
  ‣ only one visitor of the victim queue
Case of study: EPX

- EPX (EUROPLEXUS) code [CEA - IRC - EDF - ONERA], V. Faucher
  - Fluid-Structure systems subjected to fast transient dynamic loading
- Grand prix SFEN 2013
  - [http://energies.sfen.org/emploi/le-grand-prix-sfen](http://energies.sfen.org/emploi/le-grand-prix-sfen)
Multicore version of EPX

• Complex code
  ‣ 600 000 lines of code (Fortran + MPI)

• Two main sources of parallelization (~70% of the computation)
  ‣ Sparse Cholesky factorization
    - skyline representation
    - dependent tasks with data flow dependencies
  ‣ 2 Independent loops
    - LOOPELM:
      - iteration over finite elements to compute nodal internal forces
    - REPERA:
      - iteration for kinematic link detection

• ANR REPDYN
  ‣ 2009-2012
  ‣ parallelization using XKaapi
    - tasks with data flow dependencies
    - on demand task creation for // loop
    - Fortran API

• Phd Student [2013-2015]
Case of study: MEPPEN

- Main characteristics
  - Most of the time in independent loops LOOPELM and REPERA
  - AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory
Case of study: MAXPLANE

- **Main characteristics**
  - Most of the time in sparse Cholesky factorization
  - AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory
Sparse Cholesky Factorization (EPX)

- **OpenMP / XKaapi = independent tasks versus dependent tasks**
  - 59462 with 3.59% of non zero elements

![Graph showing speedup (Tp/Tseq) vs core count](image)

**Y-axis**: Speedup (Tp/Tseq)  
**X-axis**: Core count
XKaapi status

- software stack
- multi-gpu
- Intel Xeon Phi
- gdual scheduling for heterogeneous
Software stack

- Europlexus
- VTK
- Sofa

- Fortran API
- C++ API

- OpenMP-4.0
- libGOMP
- libIOMP5
- libKOMP
- GCC
- Intel ICC
- KLANG

- PLASMA
- Quark

XKaapi runtime: adaptive task with data flow

- Pthread
- Atomic
- Multicore
- GPU

- ViTE
- PAJE
Multi-GPUs Support

- [IPDPS 2013] work with J. Lima, N. Maillard, B. Raffin
- DPOTRF
  > using recursive tasks (on panel factorization) to reduce critical path


**MIC Support**

- [SBAC-PAD 2013] work with F. Broquedis, J. Lima, B. Raffin
  - Experiments with XKaapi on Intel Xeon Phi Coprocessor
- DPOTRF

---

**Intel Xeon Phi 5110, matrix size 8192, BS=256**

![Graph showing performance comparison between different libraries](image)

*Preliminary results / random work stealing*
Dual Approximation

- [Europar 2014], work with D. Trystram, R. Bleuse, J. Lima
  - Scheduling data flow program in XKaapi: A new affinity-based algorithm for heterogeneous architectures.
- Optimization: communication and performances

[Graphs showing performance and memory transfer for different algorithms and configurations]
Conclusions

• XKaapi
  ‣ runtime of OpenMP + extensions
    - libGOMP/GCC: very good support
    - libIOMP5/Intel or Clang-omp from Intel
  ‣ adaptive task
  ‣ scheduling
    - with / without « performance model »

• Next steps
  ‣ programming adaptive algorithms
    - compiler extension for OpenMP
  ‣ improving scheduling of OpenMP task’ based program
    - temporal locality
  ‣ scalable algorithms
    - many-(many) core
  ‣ reduce the size of the library
XKaapi History

- Athapascan
  - distributed work stealing
  - data flow dependencies

- Kaapi
  - + fault tolerance
  - + static scheduling
  - + adaptive task
    // loop

- Kaapi
  - + multiGPUs

- GridSs
  - data flow

- OmpSs
  - data flow
  + multiGPUs
  + cluster

- StarPU
  - data flow

- Quark
  - data flow

- CellSs
  - data flow

- SMPSs
  - data flow

- XKaapi
  - data flow
  + //loop
  + adaptive task
  + OMP RTS

- Cilk
  - work stealing
  - independent tasks

- Cilk+
  - + //loop

- Cilk/Intel

- Cilk++
  - + //loop

- TBB 1.0
  - TBB 2.0
  - TBB 3.0
  - TBB 4.1

- OpenMP 1.0
  - //loop

- OpenMP 2.0
  - //loop

- OpenMP 3.0
  + Task

- OMP 3.1

- OMP 4.0

- Time
  - 1994
  - 1996
  - 1997
  - 1998
  - 1999
  - 2000
  - 2001
  - 2002
  - 2003
  - 2004
  - 2005
  - 2006
  - 2007
  - 2008
  - 2009
  - 2010
  - 2011
  - 2012
  - 2013