X-KAAPI: a Multi Paradigm Runtime for Multicore Architectures

Thierry Gautier*, Fabien Lementec*, Vincent Faucher†, Bruno Raffin*
*INRIA, Grenoble, France
†CEA, DEN, DANS, DM2S, SEMT, DYN, Gif-sur-Yvette, France

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

- Complex architecture
  - Computing resources
    - CPU, GPU, ...
  - Memory
    - hierarchical memory (register, L1, L2, L3, main memory)
    - private / shared cache
  - Interconnection network
    - between several cores & memory

→ High complexity
  - million of components
  - heterogeneity
    - memory
    - processor

```
# Parallel architecture

- Complex architecture
  - Computing resources
    - CPU, GPU, ...
  - Memory
    - hierarchical memory (register, L1, L2, L3, main memory)
    - private / shared cache
  - Interconnection network
    - between several cores & memory

-> High complexity
  - million of components
  - heterogeneity
    - memory
    - processor
```
Goal: Write Once, Run Anywhere

• Provide performance guarantee of application
  ‣ On multiple parallel architectures
  ‣ With dynamic variation (OS jitter, application load)

• Two steps solution
  ‣ Definition of a programming model
    ◦ Task based
      - recursive task, adaptive task
    ◦ Data flow dependencies
      - computed at runtime
  ‣ Efficient scheduling algorithms
    ◦ Work stealing based with heuristic
    ◦ HEFT, DualApproximation, ...
    ◦ Theoretical analysis of performance
How to program such architecture?

MPI-1.0 → MPI-1.1 → MPI-1.3
[1994] → MPI-2.0 → MPI-2.1 → MPI-2.2 → MPI-3.0

Athapascan
distributed work stealing
data flow dependencies

Kaapi
+ fault tolerance
+ static scheduling

Kaapi
+ adaptive task
+ // loop

Kaapi
+ multiGPUs

GridSs
data flow
+ multiGPUs
+ cluster

OmpSs
data flow
+ multiGPUs
+ // loop
+ adaptive task
+ OMP RTS

Kaapi
+ multiGPUs

CellSs → SMPSs → GPUSs
+ multiGPUs

StarPU
data flow
+ multiGPUs
+ // loop

Quark
data flow
+ multiGPUs
+ adaptive task
+ cluster

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

Cilk
work stealing
independent tasks

Cilk++
+ // loop

Cilk/Intel

Cilk+

TBB 1.0 → TBB 2.0 → TBB 3.0 → TBB 4.1
+ // loop

OpenMP 1.0
+ // loop

OpenMP 2.0

OpenMP 3.0
+ Task

OMP 3.1 → OMP 4.0

Time


MPI-2.0

MPI-1.0

MPI-1.1

MPI-1.3

MPIS-1.0

MPI-1.1

MPI-1.3

[1994]
Outline

• Introduction

• Overview of Kaapi parallel programming model
  ‣ Scheduling tasks with data flow dependencing
  ‣ XKaapi’s on-demand task creation

• Evaluations
  ‣ Micro benchmarks
  ‣ EPX parallelization

• Conclusions
**Data flow dependencies**

- **Using code annotation**

  ```c
  void main()
  {
    /* data result is produced */
    compute( input, &result );
    /* data result is consumed */
    display( &result );
  }
  
  void main()
  {
    #pragma kaapi task read(input) write(result)
    compute( input, &result );
    #pragma kaapi task read(result)
    display( &result );
  }
  ```

- **Other APIs: C, C++, Fortran**

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

- **Related work**
  - StarPU [Bordeaux, France], OmpSS [BSC, Spain], Quark [UTK]
  - and new standard OpenMP-4.0 !
#include <cblas.h>
#include <clapack.h>

void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
    {
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );
        for (size_t m=k+ NB; m < N; m += NB)
        {
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                          NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }
    }
    for (size_t m=k+ NB; m < N; m += NB)
    {
        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)
        {
            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 );
        }
    }
}
XKaapi programming example

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

void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
        {
        #pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
            clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );
        }
    for (size_t m=k+ NB; m < N; m += NB)
        {
        #pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) readwrite(&A[m*N+k]{ld=N; [NB][NB]})
            cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit, 
                        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
        }
    for (size_t m=k+ NB; m < N; m += NB)
        {
        #pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}) readwrite(&A[m*N+m]{ld=N; [NB][NB]})
            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 kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]}) readwrite(&A[m*N+n]{ld=N; [NB][NB]})
            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 );
        }
        }
    }
```
Main Characteristics of XKaapi

• Parallelism is explicit, task based, with data flow dependencies
  ▸ Task’s creation is a non blocking operation
  ▸ Dependencies between tasks = Data flow dependencies
    ◦ Computed at runtime during workstealing requests
      ≠ StarPU, OmpSS, Quark = computed during task’s creation

• Scheduling
  ▸ by work stealing
    ◦ Cilk’s like performance guarantee
      - \( T_p = O\left( \frac{T_1}{p} + T_\infty \right) \)
      - Number of steal requests \( O(p T_\infty) \)
    ◦ + heuristics for data locality
  ▸ + others: ETF, HETF, DualApproximation

• Target architectures
  ▸ heterogeneous architecture: multi-CPU / multi-GPUs
  ▸ many-core: Intel Xeon Phi
The way XKaapi executes tasks

• One “worker thread” per core
  ‣ Able to execute XKaapi fine-grain tasks
  ‣ Holds a queue of tasks
    ☐ Related to sequential C stack of activation frames
    ☐ T.H.E. low overhead protocol, lock in rare case

• 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

• Work-stealing based scheduling
  ‣ Cilks’s work first principle
  ‣ Work-stealing algorithm = plug-in
    ☐ Default: steal a task from a randomly chosen queue
On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
  - Adaptative tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Example of use: the XKaapi `for_each` construct
  - A general-purpose parallel loop
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range

\[ T_1 : [0 - 15] \]
On-demand task creation with XKaapi

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of
    the remaining computation to be performed by a task under execution

• Exemple of use: the XKaapi for_each
  construct
  ‣ A general-purpose parallel loop
    ☐ A task == a range of iterations to compute
  ‣ Execution model
    ☐ Initially, one task in charge of the whole
      range

T₁ : [0 - 15]
On-demand task creation with XKaapi

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the XKaapi for_each construct
  ‣ A general-purpose parallel loop
    ○ A task == a range of iterations to compute
  ‣ Execution model
    ○ Initially, one task in charge of the whole range

\[ T_1 : [0 - 15] \]
On-demand task creation with XKaapi

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the XKaapi for_each construct
  ‣ A general-purpose parallel loop
    ‣ A task == a range of iterations to compute
  ‣ Execution model
    ‣ Initially, one task in charge of the whole range

\[ T_1 : [0 - 15] \]
On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
  - Adaptive tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Exemple of use: the XKaapi for_each construct
  - A general-purpose parallel loop
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range
    - Idle cores post steal requests and trigger the «split» operation to generate new tasks

```
T_1 : [0 - 15]
```
On-demand task creation with XKaapi

• Adaptive tasks in XKaapi
  ▸ Adaptative tasks can be split at run time to create new tasks
  ▸ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the XKaapi for_each construct
  ▸ A general-purpose parallel loop
    □ A task == a range of iterations to compute
  ▸ Execution model
    □ Initially, one task in charge of the whole range
    □ Idle cores post steal requests and trigger the «split» operation to generate new tasks

Idle Core
Idle Core

Aggregate steal requests

split (T₁, nb_stealers+1)

T₁ : [0 - 15]
On-demand task creation with XKaapi

- Adaptive tasks in XKaapi
  - Adaptative tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Exemple of use: the XKaapi for_each construct
  - A general-purpose parallel loop
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range
    - Idle cores post steal requests and trigger the «split» operation to generate new tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>[3 - 7]</td>
</tr>
<tr>
<td>T₂</td>
<td>[8 - 11]</td>
</tr>
<tr>
<td>T₃</td>
<td>[12 - 15]</td>
</tr>
</tbody>
</table>

Idle Core

Aggregate steal requests

split (T₁, nb_stealers+1)
Outline

• Introduction

• Overview of Kaapi parallel programming model
  ‣ Scheduling tasks with data flow dependencing
  ‣ XKaapi’s on-demand task creation

• Evaluations
  ‣ Micro benchmarks
  ‣ EPX parallelization

• Conclusions
Overhead of task management [AMD48]

Cilk+

long fib(long n)
{
    if (n < 2)
        return (n);
    else{
        long x, y;
        x = cilk_spawn fib(n - 2);
        y = fib(n - 2);
        cilk_sync;
        return (x + y);
    }
}

OpenMP

void fibonacci(long* result, const long n)
{
    if (n<2)
        *result = n;
    else{
        long r1,r2;
        #pragma omp task write(&r1)
        fibonacci(&r1, n-1);
        fibonacci(&r2, n-2);
        #pragma omp taskwait
        *result = r1 + r2;
    }
}

TBB

struct FibContinuation: public tbb::task {
    long* const sum;
    long x, y;
    FibContinuation( long* sum_ ) : sum(sum_) {};
    tbb::task* execute() {
        sum = x+y;
        return NULL;
    }
};

void fib(long n)
{
    if (n < 2)
        return (n);
    else {
        long x, y;
        x = cilk_spawn fib(n - 2);
        y = fib(n - 2);
        cilk_sync;
        return (x + y);
    }
}

Kaapi

struct FibTask: public tbb::task {
    long n;
    long * sum;
    FibTask( const long n_, long * const sum_ ) :
        n(n_), sum(sum_) {};
    tbb::task* execute() {
        if (n<2){
            *sum = n;
            return NULL;
        } else {
            FibContinuation& c =
                *new ( allocate_continuation() ) FibContinuation(sum);
            FibTask& b = *
                new ( c.allocate_child() ) FibTask(n-1,&c.y);
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            // Set ref_count to "two children".
            c.set_ref_count(2);
            c.spawn( b );
            return this;
        }
    }
};

void fib(long n)
{
    if (n < 2)
        return (n);
    else {
        long x, y;
        x = cilk_spawn fib(n - 2);
        y = fib(n - 2);
        cilk_sync;
        return (x + y);
    }
}
# Overhead of task management [AMD48]

- Fibonacci (35) naive recursive computation
- AMD Many Cours, 2.2GHz, 48 cores, 256GB main memory

<table>
<thead>
<tr>
<th>Serial</th>
<th>Cilk+</th>
<th>TBB (4.0)</th>
<th>OpenMP (gcc)</th>
<th>XKaapi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0905s (x 1)</td>
<td>1.063 (x 11.7)</td>
<td>2.356 (x 26)</td>
<td>2.429s (x 27)</td>
<td>0.728s (x 8)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#Cores</th>
<th>Cilk+ (s)</th>
<th>TBB 4.0 (s)</th>
<th>XKaapi (s)</th>
<th>OpenMP (gcc) (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.063</td>
<td>2.356</td>
<td>0.728</td>
<td>2.43</td>
</tr>
<tr>
<td>8</td>
<td>0.127</td>
<td>0.293</td>
<td>0.094</td>
<td>51.06</td>
</tr>
<tr>
<td>16</td>
<td>0.065</td>
<td>0.146</td>
<td>0.047</td>
<td>104.14</td>
</tr>
<tr>
<td>32</td>
<td>0.035</td>
<td>0.072</td>
<td>0.024</td>
<td><strong>NO TIME</strong></td>
</tr>
<tr>
<td>48</td>
<td>0.028</td>
<td>0.049</td>
<td>0.017</td>
<td><strong>NO TIME</strong></td>
</tr>
</tbody>
</table>
PLASMA: multicore library for dense linear algebra

- 2 versions of algorithms
  - "static": hand coded scheduler of operations over threads
  - "Quark": library to schedule tasks with data flow dependencies

Comparizon:
- PLASMA Dpotrf over:
  - static scheduler
  - quark scheduler
  - xkaapi thanks to quark APIs ported on top of xkaapi

Dense Cholesky factorization

AMD Magny Cours - 48 cores

- BS = 128
- BS = 224
Dense Cholesky factorization

8 GPU Fermi - M2050 - 12 cores

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

Preliminary results / random work stealing
Parallelization of EPX

- Multicore parallelization of EPX (EUROPLEXUS) code [CEA - IRC - EDF - ONERA]
  - ANR RepDyn funding.
- Fluid-Structure systems subjected to fast transient dynamic loading
Parallelization of EPX

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

• Two main sources of parallelization (~70% of the computation)
  ‣ Sparse Cholesky factorization
    ○ skyline representation
    ○ XKaapi Parallel program = 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
    ○ Parallelization = on-demand task creation through the XKaapi’s parallel foreach functor

• Two instances
  ‣ MEPPEN
  ‣ MAXPLANE

• AMD Many Cours, 2.2Ghz, 48 cores, 256GB main memory
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

![Graph showing time vs. number of cores and speedup]

---

**Legend**
- repera
- loopelm
- Cholesky
- other

**Axes**
- X: Number of cores
- Y: Time (seconds)

**Legend**
- LOOPELM
- REPERA
- Ideal

**Graph showing speedup vs. core count**
- X: Core count
- Y: Speedup (/T1)
Conclusions

• XKaapi
  ‣ Low overhead runtime for dependent tasks (data flow), scheduling algorithms
  ‣ Good performances on different architectures
    ‣ multi-CPPUs, multi-CPPUs-multi-GPUs, Intel Xeon Phi

• Improving parallelization of EPX
  ‣ Analysis of the remainder sequential part
  ‣ Scalability on thousands of cores

• Integration of our runtime under OpenMP-4.0 directives
  ‣ Supercomputer = high performance network + multicores + accelerators
  ‣ Software stack
    - Network: OpenMP
    - Multicores: OpenMP
    - Accelerators: Cuda, OpenCL, OpenACC
    - SIMD Units: Compiler or extension

• More informations: http://kaapi.gforge.inria.fr
  ‣ [IWOMP2012]: replacement of libGOMP for OpenMP-3.1/GCC with better management of tasks
  ‣ [IWOMP2013]: extension of loop scheduler based on “on-demand task creation”
  ‣ [IPDPS2013]: multi-CPPUs / multi-GPUs (12 cores machine + 8 GPUs)
  ‣ [SBAC-PAD13]: Comparion Intel Xeon Phi / Intel Sandybridge with OpenMP, Cilk+ and Kaapi
Thank you for your attention!

http://kaapi.gforge.inria.fr
Sparse Cholesky Factorization (EPX)

- OpenMP / XKaapi
  - 59462 with 3.59% of non zero elements

![Graph showing speedup (Tp/Tseq) vs core count for OpenMP and XKaapi with ideal line.]
Performance evaluation: VTK filters

- Parallel version of the VTK visualization toolkit [M. Ettinger, MOAIS]
  - A framework to develop parallel applications for scientific visualization
  - A VTK «filters» == a computation performed on a 2D/3D scene
  - parallel loop, static OpenMP schedule

![Graphs showing speedup with different workloads](image-url)
Comparison with OMPSS, StarPU

- DGEMM matrix size 10240, block size 1024
Comparison with OMPSS, StarPU

• DPOTRF matrix size 10240, block size 1024
Comparison with OMPSS, StarPU

• DPOTRF matrix size 40960, BS 1024

No OMPSS result due to a memory problem for big matrices (bug if bigger than 10280)
Improving OpenMP task implementation

- [IWOMP 2012]
- Barcelona OpenMP Task Suite
  ‣ Using libKOMP = our libGOMP implementation (on top of XKaapi)
  ‣ A set of representative benchmarks to evaluate OpenMP tasks implementations

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments used</th>
<th>Domain</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>prot100(aa)</td>
<td>Dynamic programming</td>
<td>Aligns sequences of proteins</td>
</tr>
<tr>
<td>FFT</td>
<td>n=33,554,432</td>
<td>Spectral method</td>
<td>Computes a Fast Fourier Transformation</td>
</tr>
<tr>
<td>Floorplan</td>
<td>input.20</td>
<td>Optimization</td>
<td>Computes the optimal placement of cells in a floorplan</td>
</tr>
<tr>
<td>NQueens</td>
<td>n=14</td>
<td>Search</td>
<td>Finds solutions of the N Queens problem</td>
</tr>
<tr>
<td>MultiSort</td>
<td>n=33,554,432</td>
<td>Integer sorting</td>
<td>Uses a mixture of sorting algorithms to sort a vector</td>
</tr>
<tr>
<td>SparseLU</td>
<td>n=128 m=64</td>
<td>Sparse linear algebra</td>
<td>Computes the LU factorization of a sparse matrix</td>
</tr>
<tr>
<td>Strassen</td>
<td>n=8192</td>
<td>Dense linear algebra</td>
<td>Computes a matrix multiply with Strassen’s method</td>
</tr>
<tr>
<td>UTS</td>
<td>medium.input</td>
<td>Search</td>
<td>Computes the number of nodes in an Unbalanced Tree</td>
</tr>
</tbody>
</table>

• Evaluation platforms
  ‣ AMD48: 4x12 AMD Opteron (6174) cores
  ‣ Intel32: 4x8 Intel Xeon (X7560) cores

• Softwares
  ‣ gcc 4.6.2 + libGOMP
  ‣ gcc 4.6.2 + libKOMP
  ‣ icc 12.1.2 + Intel OpenMP runtime (KMP)
Running OpenMP BOTS with libKOMP

**Speed-Up** of BOTS kernels on the AMD48 platform

<p>|</p>
<table>
<thead>
<tr>
<th>kernel</th>
<th>libGOMP</th>
<th>libKOMP</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>38.8</td>
<td><strong>40.0</strong></td>
<td>37.0</td>
</tr>
<tr>
<td>FFT</td>
<td>0.5</td>
<td><strong>12.2</strong></td>
<td>12.0</td>
</tr>
<tr>
<td>Floorplan</td>
<td>27.6</td>
<td><strong>32.7</strong></td>
<td>29.2</td>
</tr>
<tr>
<td>NQueens</td>
<td>43.7</td>
<td><strong>47.8</strong></td>
<td>39.0</td>
</tr>
<tr>
<td>MultiSort</td>
<td>0.6</td>
<td><strong>13.2</strong></td>
<td>11.3</td>
</tr>
<tr>
<td>SparseLU</td>
<td>44.1</td>
<td><strong>44.4</strong></td>
<td>35.0</td>
</tr>
<tr>
<td>Strassen</td>
<td>20.8</td>
<td><strong>22.4</strong></td>
<td>20.5</td>
</tr>
<tr>
<td>UTS</td>
<td>0.9</td>
<td><strong>25.3</strong></td>
<td>15.0</td>
</tr>
</tbody>
</table>

**Evaluation platforms**
- AMD48: 4x12 AMD Opteron (6174) cores
- Intel32: 4x8 Intel Xeon (X7560) cores

**Softwares**
- gcc 4.6.2 + libGOMP
- gcc 4.6.2 + libKOMP
- icc 12.1.2 + Intel OpenMP runtime (KMP)
```c
#include <cblas.h>
#include <clapack.h>

void Cholesky( double* A, int N, size_t NB )
{
#ifdef OMPTask
#pragma omp task depend
#endif
#pragma omp parallel
{
#pragma omp for (size_t k=0; k < N; k += NB)
#pragma omp single
{
#pragma omp task depend
#endif
#pragma omp for (size_t m=k+ NB; m < N; m += NB)
#pragma omp task depend
#endif
#pragma omp for (size_t n=k+NB; n < m; n += NB)
}
}
```
## Overhead of task management [Xeon Phi]

- Fibonacci (38) naive recursive computation
- 60 cores of Intel Xeon Phi 5100

<table>
<thead>
<tr>
<th>#threads</th>
<th>Cilk+ (s)</th>
<th>OpenMP (gcc) (s)</th>
<th>XKaapi (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.21</td>
<td>65.64</td>
<td>15.52</td>
</tr>
<tr>
<td>10</td>
<td>3.34</td>
<td>33.12</td>
<td>1.58</td>
</tr>
<tr>
<td>20</td>
<td>1.66</td>
<td>17.54</td>
<td>0.79</td>
</tr>
<tr>
<td>60</td>
<td>0.56</td>
<td>6.30</td>
<td>0.27</td>
</tr>
<tr>
<td>120</td>
<td>0.38</td>
<td>3.86</td>
<td>0.18</td>
</tr>
<tr>
<td>240</td>
<td>0.37</td>
<td>3.18</td>
<td>0.18</td>
</tr>
</tbody>
</table>
Stealing a task

- A thief thread do
  - Iteration through the tasks in a victim queue
    - iteration order = creation time order
  - Computation of data flow dependencies
    - with previously visited tasks
  - Detection of ready tasks

- Lazy computation of data flow dependencies
  - In work stealing “theory”, called work first principle
    - overhead is move from the work of the program to the critical path
1 GPU

- Nvidia Event
- Stream H2D
- Stream D2H
- Stream Kernel
#include <cblas.h>
#include <clapack.h>

void Cholesky( double* A, int N, size_t NB )
{
    for (size_t k=0; k < N; k += NB)
    {
#pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
        clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

#pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) \ readwrite(&A[m*N+k]{ld=N; [NB][NB]})
        cblas_dtrsm( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
                    NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }
...}
False dependencies resolution

• Also called:
  ‣ Write after Read
  ‣ Writer after Write

• Here =>
  ‣ Task1 and Task2 cannot be concurrent except if ‘a’ is duplicated
    - also called ‘variable renaming’