Linear Algebra 2: Parallel programming tools for exact linear algebra

Clément PERNET
JOINT WORK WITH THIERRY GAUTIER,

LIG/INRIA-MOAIS, Grenoble Université, France

ECRYPT II: Summer School on Tools,
Mykonos, Grece,
June 1st, 2012

Introduction

Back in the times, when everything was sequential

Introduction

Back in the times, when everything was sequential

Introduction

Back in the times, when everything was sequential

Introduction

Back in the times, when everything was sequential



Introduction

Back in the times, when everything was sequential

Introduction

Fortunately the great time of parallelism has come...

Introduction

Fortunately the great time of parallelism has come...

Introduction :

Parallel architecture: heterogeneity

- multicore [>8 cores], ccNUMA
- network [mostly infiniband]
- GPU, separate address space
- Intel MIC
- FPGA
- ...

Main characteristics:

- complexity: memory hierarchy, number of cores
- changing hardware: Net. on Chip, Integration CPU/GPU...

Challenge

How to programm heterogeneous architectures ?

Criteria

- good performances
- portability across architectures
- abstraction for simplicity

Challenging key point: scheduling as a plugin

- Program: description of the parallelism
  e.g. which code portions are tasks
- Runtime: scheduling, mapping decision

3 main programming models:

1. Parallel loop [data parallelism]
2. Fork-Join (independent tasks) [task parallelism]
3. Dependent tasks with data flow dependencies [task parallelism]

Outline

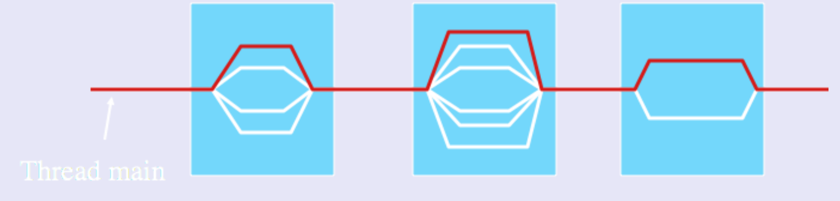
- Parallel programming models
  - Parallel loop model
  - Fork-join model
  - Data flow Tasks model
  - Existing solutions
- Comparison Fork-Join vs Data flow
  - Overhead of task management

Parallel loop model

$$\forall i \in [0, n[\text{ do } f(i),$$

- where $i \neq j \Rightarrow f(i)$ and $f(j)$ are independent,
- i.e. result is independent of the execution order of $f(i)$ and $f(j)$.

Reference software: OpenMP 1.0



OMP

```
for (int step = 0; step < 2; ++step){
#pragma omp parallel for
    for (int i = 0; i < count; ++i)
        A[i] = (B[i+1] + B[i-1] + 2.0*B[i])*0.25;
}
```

Cilk

```
for (int step = 0; step < 2; ++step){
  cilk_for (int i = 0; i < count; ++i)
    A[i] = (B[i+1] + B[i-1] + 2.0*B[i]) * 0.25;
}
```

Kaapi

```
for (int step = 0; step < 2; ++step){
#pragma kaapi parallel loop
  for (int i = 0; i < count; ++i)
    A[i] = (B[i+1] + B[i-1] + 2.0*B[i]) * 0.25;
}
```

Fork join model

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs
- Naive canonical example: recursive Fibonacci computation

OMP

```
void fibonacci(long* result, long n) {
  if (n < 2)
    *result = n;
  else {
    long x,y;
#pragma omp task
    fibonacci( &x, n-1 );
    fibonacci( &y, n-2 );
#pragma omp taskwait
    *result = x + y;
  }
}
```

Fork join model

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs
- Naive canonical example: recursive Fibonacci computation

Cilk+

```
long fibonacci(long n) {
  if (n < 2)
    return (n);
  else {
    long x, y;
    x = cilk_spawn fibonacci(n - 1);
    y = fibonacci(n - 2);
    cilk_sync;
    return (x + y);
  }
}
```

Fork join model

- Task based program: **spawn** + **sync**
- Especially suited for recursive programs
- Naive canonical example: recursive Fibonacci computation

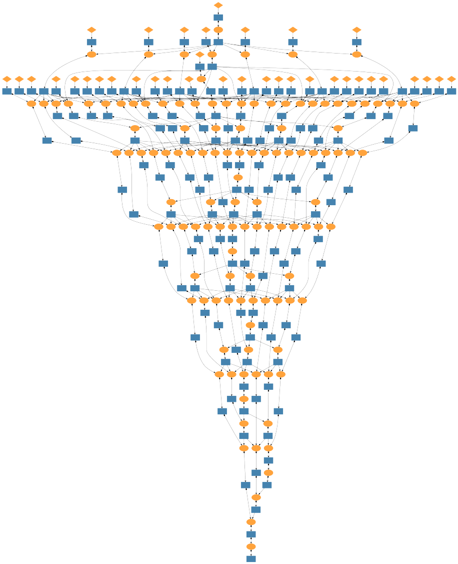
Kaapi

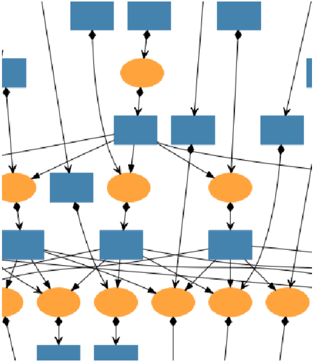
```
void fibonacci(long* result, long n) {
  if (n<2)
    *result = n;
  else {
    long x,y;
#pragma kaapi task
    fibonacci( &x, n-1 );
    fibonacci( &y, n-2 );
#pragma kaapi sync
    *result = x + y;
  }
}
```

Data flow task model

- Task based model
- Basic definition:
  - A task is ready for execution when all its inputs variables are ready
  - A variable is ready when it was written (...)
- Old languages: ID, SISAL...
- New languages/libraries: Athapascan [96], Kaapi [06], StarSs [07], StarPU [08], Quark [10]...

Data flow graph: Cholesky factorization





SmpSS

```
#pragma smpss task write(array)
extern void compute( double* array, int count);
#pragma smpss task read(array)
extern void print( double* array, int count);
int main() {
#pragma smpss start
    compute( array, count);
    print( array, count);     // Read after write dependency
#pragma smpss sync
#pragma smpss finish
}
```

Kaapi

```
int main() {
#pragma kaapi parallel
  {
#  pragma kaapi task write(array[0..count])
    compute( array, count);
#  pragma kaapi task read(array[0..count])
    print( array, count);      // Read after write dependency
  } // implicit barrier at the end of Kaapi parallel region
}
```

Existing solutions

	// prog model	Architecture	Target app.
Cilk[96]	Fork-join	Multi-CPUs	Divide&Conquer
OMP 1.0 [97]	Parallel loop	Multi-CPUs	ForEach
+ 3.0 [08]	+ Fork-join	Multi-CPUs	+ Divide&Conquer
Athapascan[98]	Rec. Data flow	Clusters+multi-CPU	D&C, LinAlg
TBB[06]	Parallel loop	Multi-CPU	D&C, Linalg
	Fork-join		
Kaapi[06-12]	Rec. Data flow	Multi-CPUs & GPUs	D&Q, LinAlg
	Parallel loop		ForEach,
StarSs [07]	Flat data flow	multi-CPUs (SMPSs)	LinAlg
	Flat data flow	multi-CPUs (SMPSs)	LinAlg
	Flat data flow	Cell (CellSs)	LinAlg
	Flat data flow	Grid (GridSs)	LinAlg
StarPU [09]	Flat data flow	multi-CPUs&GPUs	LinAlg
Quark[10]	Flat data flow	Multi-CPUs	LinAlg

Outline

Parallel programming models

- Parallel loop model
- Fork-join model
- Data flow Tasks model
- Existing solutions

Comparison Fork-Join vs Data flow

- Overhead of task management

Comparison Fork-Join vs Data flow

Fork-Join: OpenMP-3.0

Data flow: Kaapi

Goal: how excessive synchronizations affect performances

By studying

- impact on performances on Cholesky/LU matrix factorization
- cost of task creation and scheduling (micro benchmark: Fibonacci)

Fork-Join vs Data flow

Strong synchronizations in Fork-Join model:

- if task T_1 depend on task T_0 e.g. task T_0 produces value for task T_1
- spawn T0; sync; spawn T1; spawn T2; ...
- synchronization point at “sync”: barrier that waits for all previous spawned tasks, even if concurrency exists with some tasks after the barrier

Fork-Join vs Data flow

Strong synchronizations in Fork-Join model:

- if task T_1 depend on task T_0 e.g. task T_0 produces value for task T_1
- spawn T0; sync; spawn T1; spawn T2; ...
- synchronization point at “sync”: barrier that waits for all previous spawned tasks, even if concurrency exists with some tasks after the barrier

Data Flow model:

data flow tasks to express such dependencies

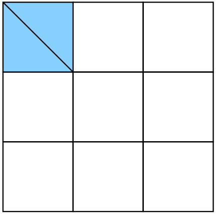
- program : creates tasks
- runtime : schedule tasks according to the real dependencies

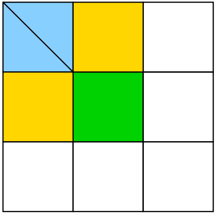
Illustration: Cholesky factorization

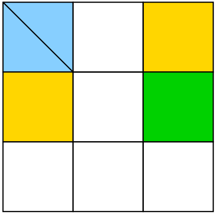
```
void Cholesky( double* A, int N, size_t NB ) {

  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 );
      }
    }

  }
}
```

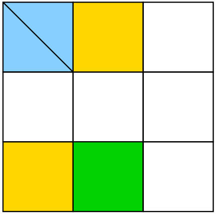
Illustration: Cholesky factorization

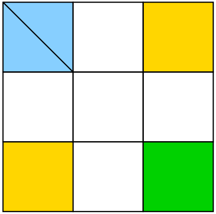
```
void Cholesky( double* A, int N, size_t NB ) {
#pragma omp parallel
#pragma omp single nowait
  for (size_t k=0; k < N; k += NB)
  {
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma omp task firstprivate(k, m) shared(A)
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }
#pragma omp taskwait // Barrier: no concurrency with next tasks
    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma omp task firstprivate(k, m) 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 firstprivate(k, m) 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 // Barrier: no concurrency with tasks at iteration k+1
  }
}
```









SYNC.

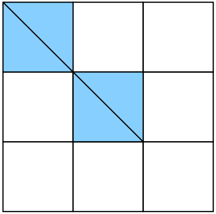
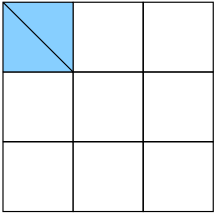
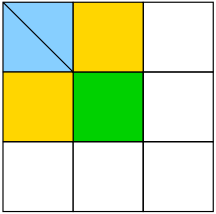
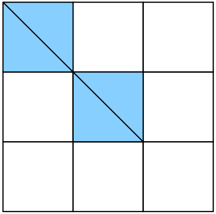


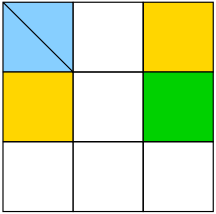
Illustration: Cholesky factorization

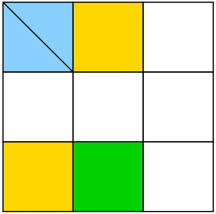
```
void Cholesky( double* A, int N, size_t NB ){
#pragma kaapi parallel
  for (size_t k=0; k < N; k += NB)
  {
#pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    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 );
      }
    }
  }
  // Implicit barrier only at the end of Kaapi parallel region
}
```

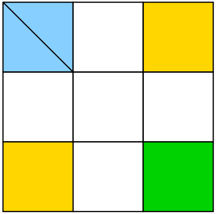













Benchmarks

Sparse version of the above: Kaapi vs OMP codes.

Benchmarks

Sparse version of the above: Kaapi vs OMP codes.

Also confirmed by other versions of data-flow tasks:

- PLASMA [Dongarra& Al.]
- SMPSs [Badia & Al.]

Challenges proper to exact linear algebra

Slicing dimensions

- Uniform block slicing leads to unbalanced load
- Varying block sizes set statically
- Dynamically adapted block sizes (work-stealing)

Rank deficient matrices

- block sizes revealed during execution

Overhead of task management

Algorithm: naive recursive Fibonacci computation

Fork-join model:

- OpenMP : gcc-4.6.2
- Cilk+ / Intel : icc-12.1.2
- TBB 4.0

Data flow model: Kaapi-1.0.2

AMD Opteron 4×12 cores

OpenMP

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

Kaapi

```
void fibonacci(long* result,const long n){
  if (n<2) *result = n;
  else
  {
    long x,y;
#pragma kaapi task write(x)
    fibonacci( &x, n-1 );
    fibonacci( &y, n-2 );
#pragma kaapi sync
    *result = x + y;
  }
}
```

Cilk +

```
long fibonacci(long n){
  if (n < 2) return (n);
  else {
  long x, y;
  x = cilk_spawn fibonacci(n - 1);
  y = fibonacci(n - 2);
  cilk_sync;
  return (x + y);
  }
}
```

Intel 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;}
};
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 N
        } else {
            FibContinuation& c = *new(allocate_conti
            FibTask& b = *new( c.allocate_child() )
            recycle_as_child_of(c);
            n -= 2;
            sum = &c.x;
            c.set_ref_count(2);
            c.spawn( b );
```

Results

Sequential	Cilk+	TBB-4.0	OpenMP	Kaapi
0.0904s	1.063s	2.356s	2.429s	0.728s
Slowdown ($\frac{T_1}{\text{Sequential}}$)	$\times 11.7$	$\times 26$	$\times 27$	$\times 8$

# cores	Cilk+	TBB-4.0	Kaapi	OpenMP
1	1.063	2.356	0.728	2.43
8	0.127	0.293	0.094	51.06
16	0.065	0.146	0.047	104.14
32	0.035	0.072	0.024	No time
48	0.028	0.049	0.017	No time

Conclusion

Difficult choice of the parallel programming language:

- POSIX threads: set the scheduling at programming time
- OpenMP:
  - Parallel loops
  - Fork-join Tasks
  - But still no data flow capabilities
- Cilk, TBB, Kaapi:
  - Parallel loop
  - Data flow tasks model (recursive or flat)
  - annotation, library, or proper compiler

Conclusion

Difficult choice of the parallel programming language:

- POSIX threads: set the scheduling at programming time
- OpenMP:
  - Parallel loops
  - Fork-join Tasks
  - But still no data flow capabilities
- Cilk, TBB, Kaapi:
  - Parallel loop
  - Data flow tasks model (recursive or flat)
  - annotation, library, or proper compiler

Towards fully adaptive parallelism

- Work-stealing but in a fixed set of tasks (created at start-up time)
- Aim at *on-the-fly tasks creations* (extraction of parallelism from sequential code)