

Harnessing clusters of hybrid nodes with a sequential task-based programming model

Emmanuel AGULLO, Olivier AUMAGE, Mathieu FAVERGE,
Nathalie FURMENTO, Florent PRUVOST, Marc SERGENT,
Samuel THIBAUT

ITER summer school, Aix-En-Provence, France, August 28th, 2014



MORSE



1. Introduction
2. Sequential task-based paradigm on a single node
3. A new programming paradigm for clusters?
4. Distributed Data Management
5. Comparison against state-of-the-art approaches
6. Conclusion and future work

- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: **Cholesky factorization (DPOTRF)**

- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: Cholesky factorization (DPOTRF)

- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: **Cholesky** factorization (DPOTRF)

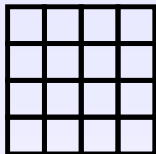
- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: **Cholesky** factorization (DPOTRF)

- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: Dense **Cholesky** factorization (DPOTRF)

- Specialized dense linear algebra libraries
 - ▶ Netlib BLAS & LAPACK sequential references, Intel MKL multithreaded vendor library, ScaLAPACK MPI reference, Nvidia CuBLAS GPU library, HPL Top 500 benchmark, ...
- Runtime systems usually abstract a single node
 - ▶ Plasma/Quark, Flame/SuperMatrix, Morse/StarPU, Dplasma/PaRSEC ...
- How should nodes communicate?
 - ▶ Using explicit MPI user calls
 - ▶ Using a specific paradigm: Dplasma
- Can we keep the same paradigm and let the runtime handle data transfers?
 - ▶ Master-slave model (e.g.: ClusterSs)
 - ▶ Replicated unroll model (e.g. Quarkd)
- Example: Dense **Cholesky** factorization (DPOTRF)

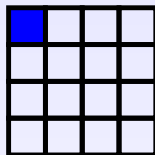
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



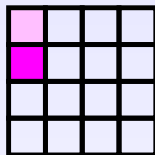
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



Sequential task-based Cholesky on a single node

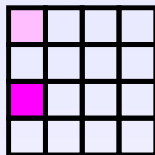
```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



Sequential task-based Cholesky on a single node

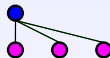
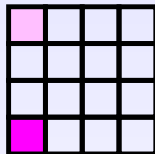
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
```

`task_wait_for_all();`



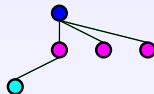
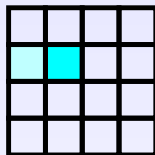
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
  POTRF (RW,A[j][j]);  
  for (i = j+1; i < N; i++)  
    TRSM (RW,A[i][j], R,A[j][j]);  
  for (i = j+1; i < N; i++) {  
    SYRK (RW,A[i][i], R,A[i][j]);  
    for (k = j+1; k < i; k++)  
      GEMM (RW,A[i][k],  
           R,A[i][j], R,A[k][j]);  
  }  
}  
task_wait_for_all();
```



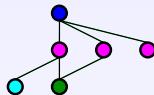
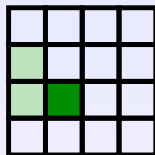
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
  POTRF (RW,A[j][j]);  
  for (i = j+1; i < N; i++)  
    TRSM (RW,A[i][j], R,A[j][j]);  
  for (i = j+1; i < N; i++) {  
    SYRK (RW,A[i][i], R,A[i][j]);  
    for (k = j+1; k < i; k++)  
      GEMM (RW,A[i][k],  
           R,A[i][j], R,A[k][j]);  
  }  
}  
task_wait_for_all();
```



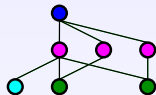
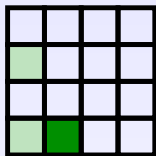
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
  POTRF (RW,A[j][j]);  
  for (i = j+1; i < N; i++)  
    TRSM (RW,A[i][j], R,A[j][j]);  
  for (i = j+1; i < N; i++) {  
    SYRK (RW,A[i][i], R,A[i][j]);  
    for (k = j+1; k < i; k++)  
      GEMM (RW,A[i][k],  
           R,A[i][j], R,A[k][j]);  
  }  
}  
task_wait_for_all();
```



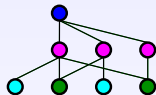
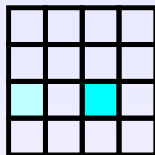
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
           R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



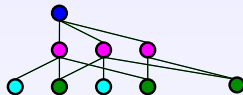
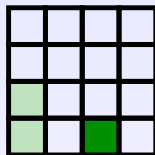
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



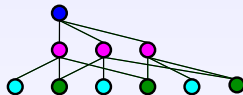
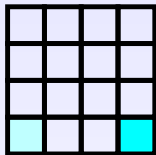
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



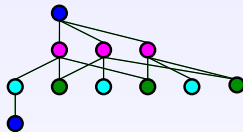
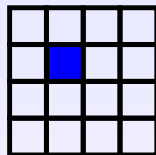
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
  POTRF (RW,A[j][j]);  
  for (i = j+1; i < N; i++)  
    TRSM (RW,A[i][j], R,A[j][j]);  
  for (i = j+1; i < N; i++) {  
    SYRK (RW,A[i][i], R,A[i][j]);  
    for (k = j+1; k < i; k++)  
      GEMM (RW,A[i][k],  
           R,A[i][j], R,A[k][j]);  
  }  
}  
task_wait_for_all();
```



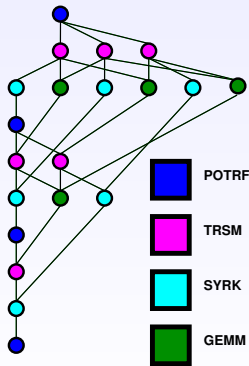
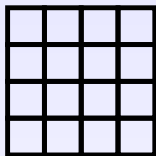
Sequential task-based Cholesky on a single node

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j]);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j]);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j]);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j]);  
    }  
}  
task_wait_for_all();
```



Sequential task-based Cholesky on a single node

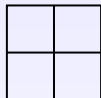
```
for (j = 0; j < N; j++) {  
  POTRF (RW,A[j][j]);  
  for (i = j+1; i < N; i++)  
    TRSM (RW,A[i][j], R,A[j][j]);  
  for (i = j+1; i < N; i++) {  
    SYRK (RW,A[i][i], R,A[i][j]);  
    for (k = j+1; k < i; k++)  
      GEMM (RW,A[i][k],  
           R,A[i][j], R,A[k][j]);  
  }  
}  
task_wait_for_all();
```



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

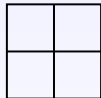
CPU



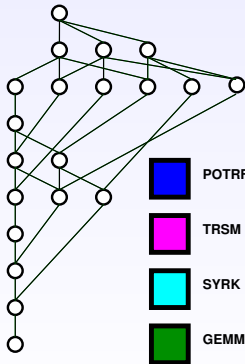
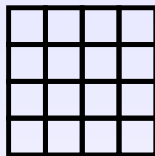
GPU0



CPU



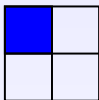
GPU1



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



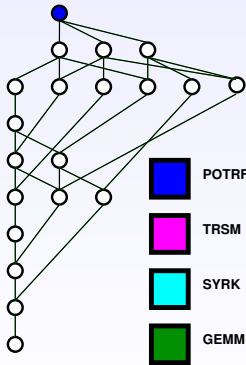
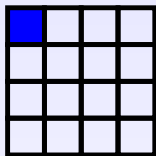
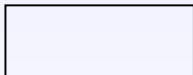
GPU0



CPU



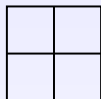
GPU1



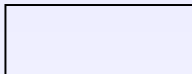
Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

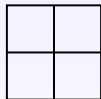
CPU



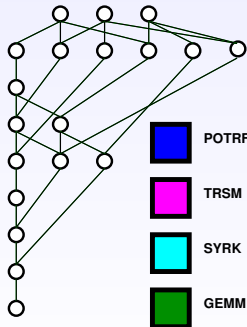
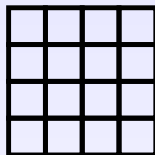
GPU0



CPU



GPU1



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



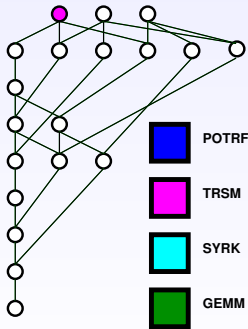
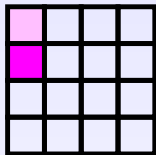
GPU0



CPU



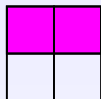
GPU1



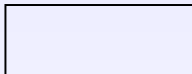
Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

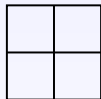
CPU



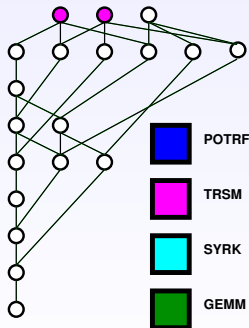
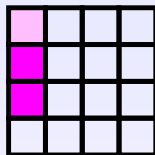
GPU0



CPU



GPU1



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



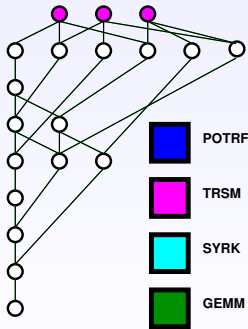
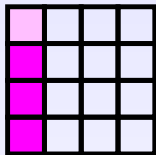
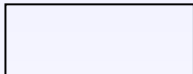
GPU0



CPU



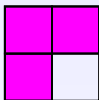
GPU1



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



GPU0



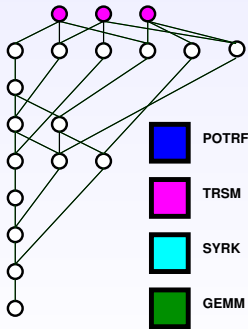
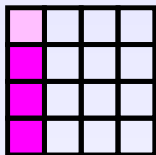
CPU



GPU1



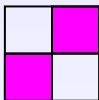
- Handles dependencies



Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



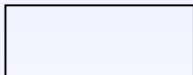
GPU0



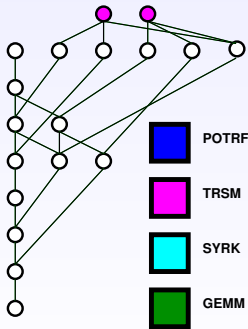
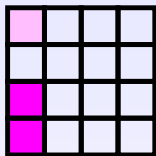
CPU



GPU1



- Handles dependencies



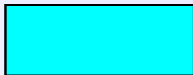
Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



GPU0



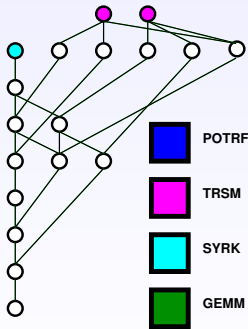
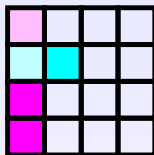
CPU



GPU1



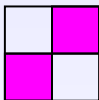
- Handles dependencies



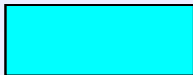
Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

CPU



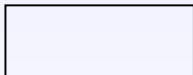
GPU0



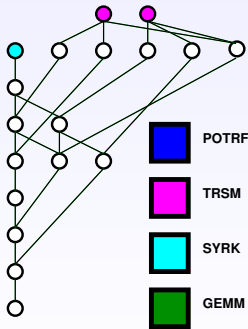
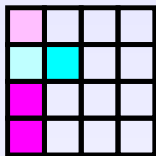
CPU



GPU1

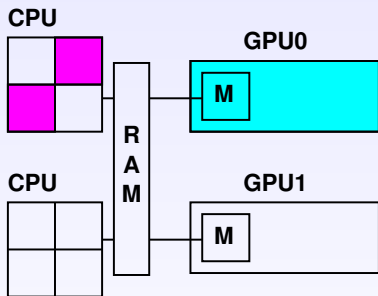


- Handles dependencies
- Handles scheduling (e.g. HEFT)

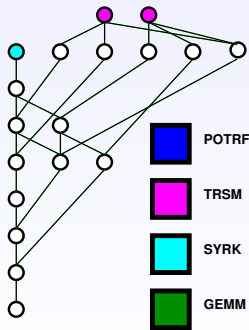
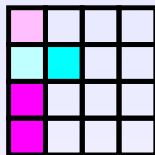


Runtime parallel execution on a heterogeneous node

```
task_wait_for_all();
```

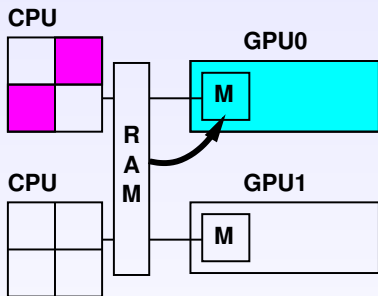


- Handles dependencies
- Handles scheduling (e.g. HEFT)

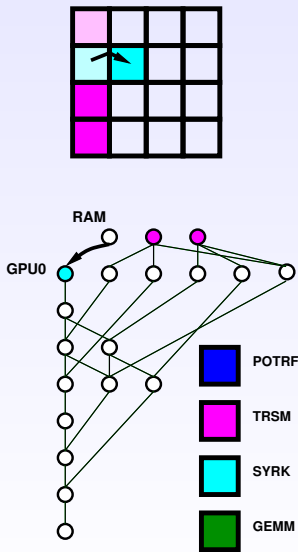


Runtime parallel execution on a heterogeneous node

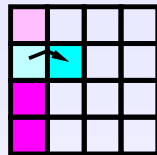
```
task_wait_for_all();
```



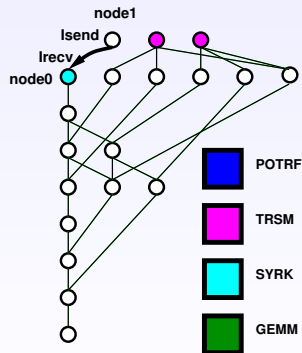
- Handles dependencies
- Handles scheduling (e.g. HEFT)
- Handles data consistency (MSI protocol)



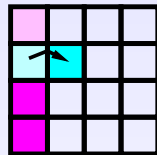
A new programming paradigm for clusters?



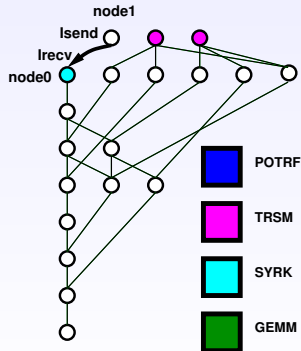
- Inferring communications from the task graph



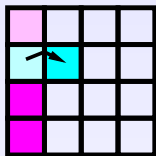
A new programming paradigm for clusters?



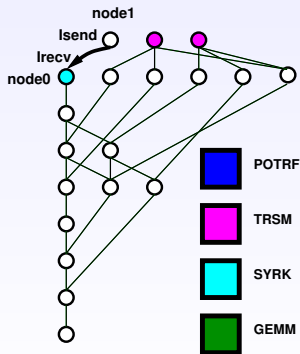
- Inferring communications from the task graph
- How to establish the mapping?



A new programming paradigm for clusters?

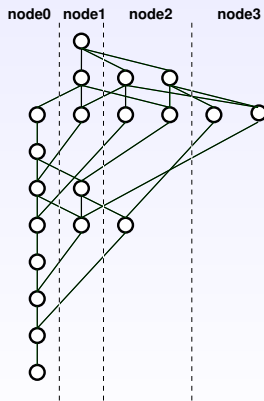


- Inferring communications from the task graph
- How to establish the mapping?
- How to initiate communications?



Mapping: Which node executes which tasks?

- The application provides the mapping



Data transfers between nodes

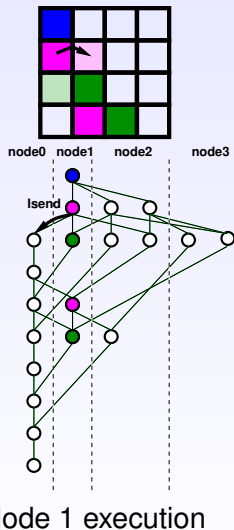
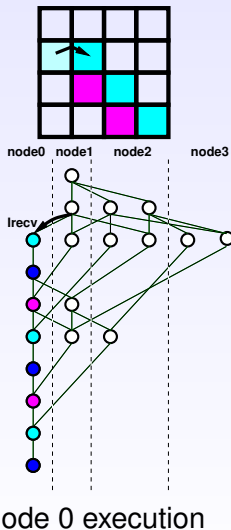
All nodes unroll the whole task graph

They determine tasks they will execute

They can infer required communications

No negotiation between nodes
(not master-slave)

Unrolling can be pruned



Same paradigm for clusters (vs single node)

same code

```
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j]);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j]);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j]);
    }
}
task_wait_for_all();
```

Same paradigm for clusters (vs single node)

Almost same code

- MPI communicator

```
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j], WORLD);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j], WORLD);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j], WORLD);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j], WORLD);
    }
}
task_wait_for_all();
```


Same paradigm for clusters (vs single node)

Almost same code

- MPI communicator
- Mapping function

```
int getnode(int i, int j) { return((i%p)*q + j%q); }

for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j], WORLD, getnode(j,j));
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j], WORLD, getnode(i,j));
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j], WORLD, getnode(i,i));
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j], WORLD, getnode(i,k));
    }
}
task_wait_for_all();
```

Same paradigm for clusters (vs single node)

Almost same code

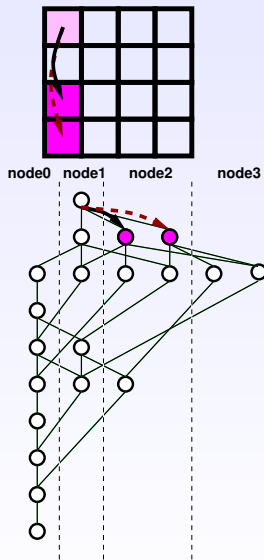
- MPI communicator
- Mapping function

```
int getnode(int i, int j) { return((i%p)*q + j%q); }  
set_rank(A, getnode);
```

```
for (j = 0; j < N; j++) {  
    POTRF (RW,A[j][j], WORLD);  
    for (i = j+1; i < N; i++)  
        TRSM (RW,A[i][j], R,A[j][j], WORLD);  
    for (i = j+1; i < N; i++) {  
        SYRK (RW,A[i][i], R,A[i][j], WORLD);  
        for (k = j+1; k < i; k++)  
            GEMM (RW,A[i][k],  
                R,A[i][j], R,A[k][j], WORLD);  
    }  
}  
task_wait_for_all();
```

Data transfers and cache system

- Duplicate data transfers.
- Let us avoid them
- It means caching the received data
- And drop it later
 - ▶ When written to
 - ▶ As advised by application



Cache flush

```
int getnode(int i, int j) { return((i%p)*q + j%q); }
set_rank(A, getnode);
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j], WORLD);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j], WORLD);

    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j], WORLD);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j], WORLD);
    }
}

task_wait_for_all();
```

Cache flush

```
int getnode(int i, int j) { return((i%p)*q + j%q); }
set_rank(A, getnode);
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j], WORLD);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j], WORLD);
    flush(A[j][j]);
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j], WORLD);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j], WORLD);
        flush(A[i][j]);
    }
}

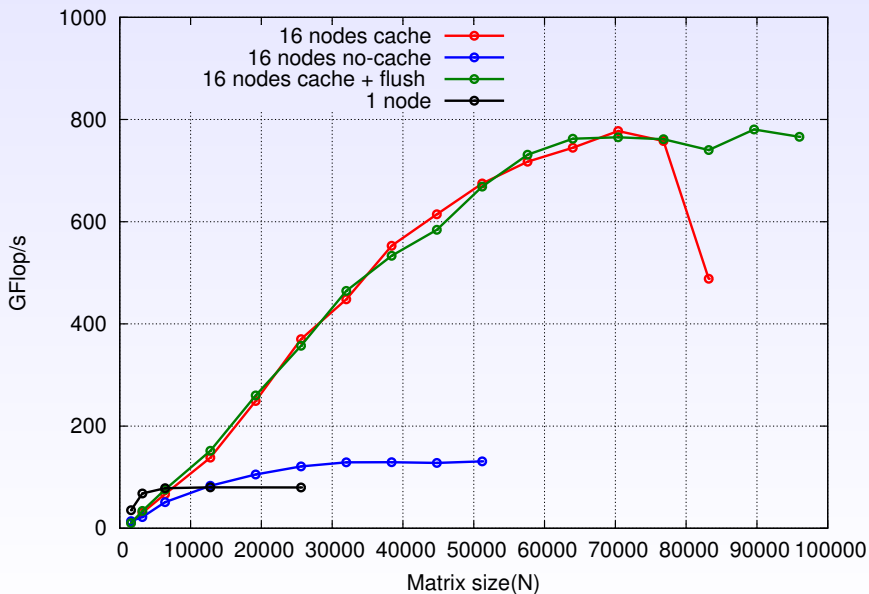
task_wait_for_all();
```

Cache flush

```
int getnode(int i, int j) { return((i%p)*q + j%q); }
set_rank(A, getnode);
for (j = 0; j < N; j++) {
    POTRF (RW,A[j][j], WORLD);
    for (i = j+1; i < N; i++)
        TRSM (RW,A[i][j], R,A[j][j], WORLD);

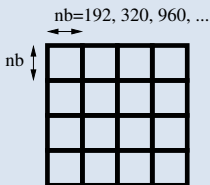
    for (i = j+1; i < N; i++) {
        SYRK (RW,A[i][i], R,A[i][j], WORLD);
        for (k = j+1; k < i; k++)
            GEMM (RW,A[i][k],
                R,A[i][j], R,A[k][j], WORLD);
    }
}
flush_all();
task_wait_for_all();
```

Cache effect, Cholesky Factorization, tile size 320

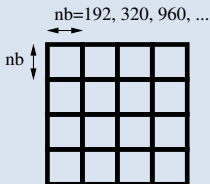


- StarPU-MPI is actually a separate library on top of StarPU
- MPI communications are technically very much like tasks running on CPU
 - ▶ StarPU automatically fetches data from GPU as needed
 - ▶ StarPU automatically pushes data to GPU as needed

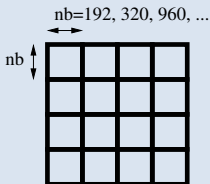
- Double-precision **Cholesky**
 - ▶ Scalapack
 - ▶ Dplasma/PaRSEC
 - ▶ **Magma-morse/StarPU**
- 64 nodes



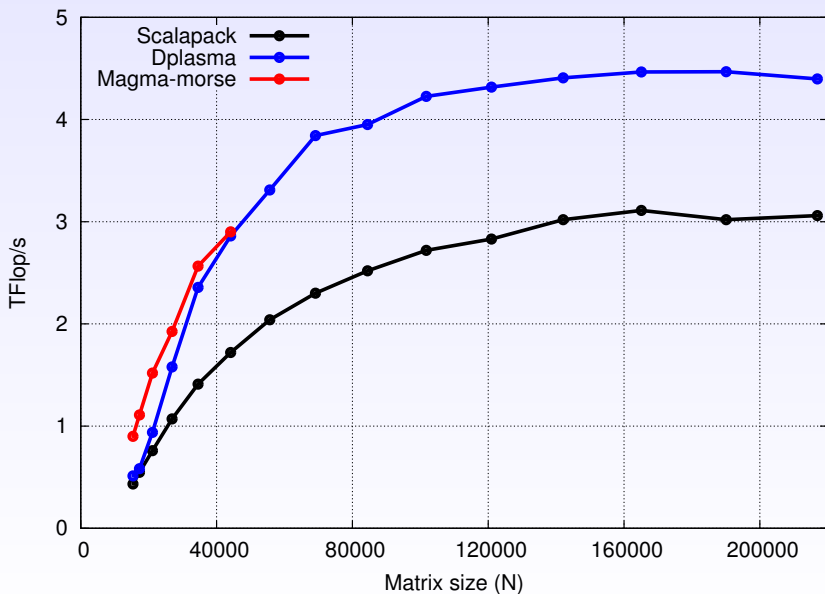
- Double-precision **Cholesky**
 - ▶ Scalapack
 - ▶ Dplasma/PaRSEC
 - ▶ **Magma-morse/StarPU**
- 64 nodes
 - ▶ 2 Intel Westmere @ 2.66 GHz (8 cores per node)
- Homogeneous tile size: 192x192



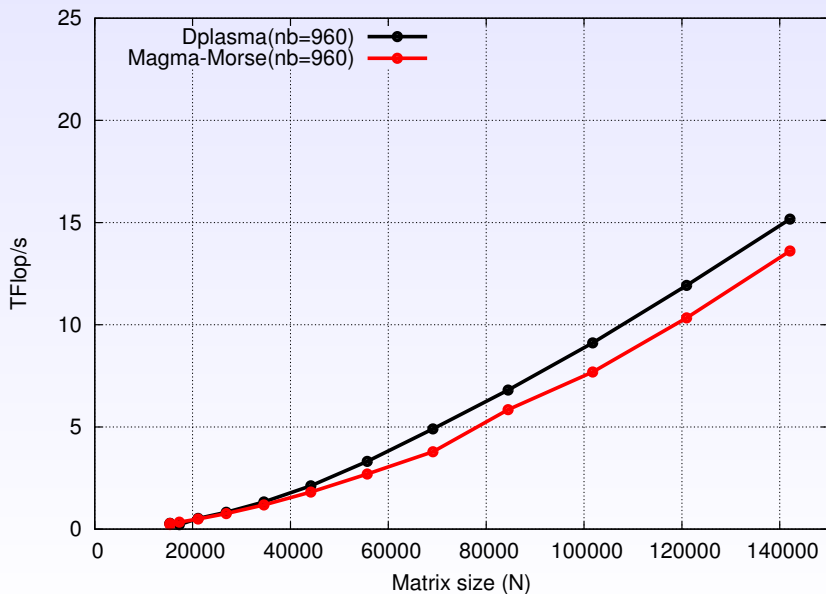
- Double-precision **Cholesky**
 - ▶ Scalapack
 - ▶ Dplasma/PaRSEC
 - ▶ **Magma-morse/StarPU**
- 64 nodes
 - ▶ 2 Intel Westmere @ 2.66 GHz (8 cores per node)
 - ▶ 2 Nvidia Tesla M2090 (2 GPUs per node)
- Homogeneous tile size: 192x192
- Heterogeneous tile sizes: 320x320 / 960x960



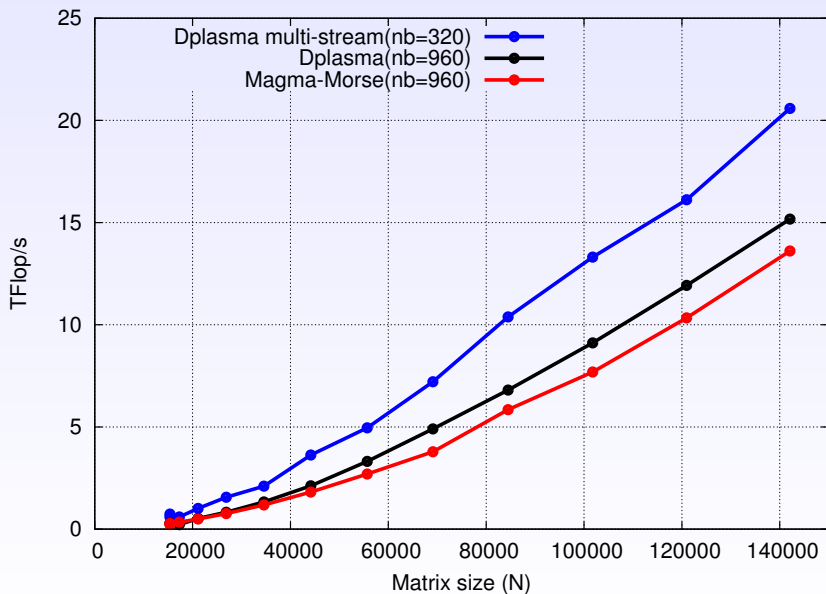
64 homogeneous nodes (8 cores per node)



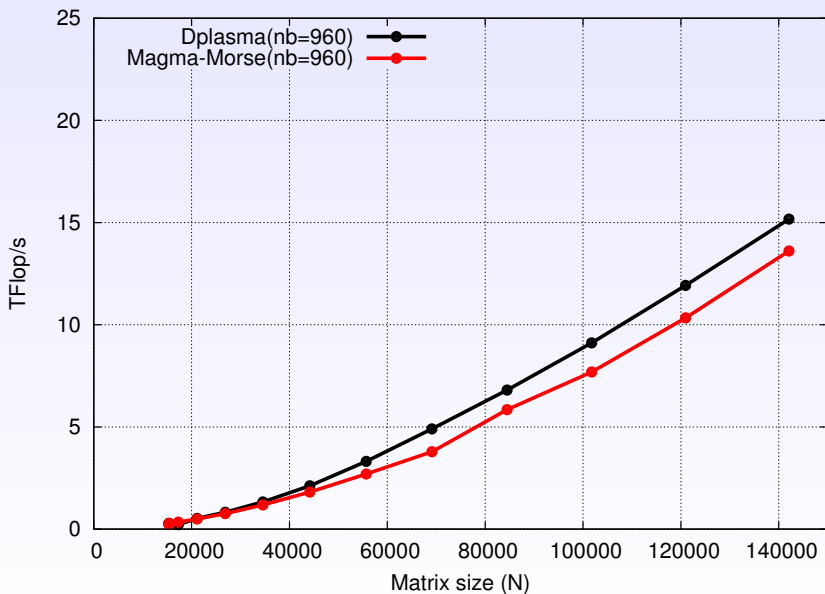
64 heterogeneous nodes (8 cores + 2 GPUs per node)



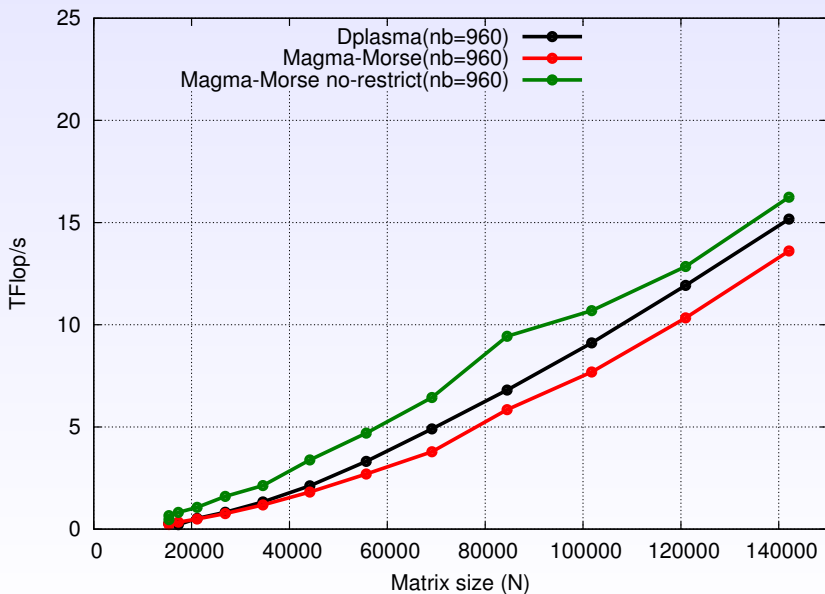
64 heterogeneous nodes (8 cores + 2 GPUs per node)



64 heterogeneous nodes (8 cores + 2 GPUs per node)



64 heterogeneous nodes (8 cores + 2 GPUs per node)



Contribution

- **Almost no code changes vs single node**
- **Competitive performance**

Future work

- Dynamic inter-node load balancing
- Automatic pruning

Related work

- Sparse direct solvers (see Pierre's talk)
- Krylov methods, hybrid (direct+iterative) solvers, FMM, H-matrix
- Explicit cases
- See OpenMP 3.1 & 4.0 standards

MORSE: <http://icl.cs.utk.edu/morse/>

StarPU: <http://runtime.bordeaux.inria.fr/StarPU/>

Open postdoctoral positions @ INRIA Bordeaux

Pruning the task graph traversal

