On the design of parallel linear solvers for large scale problems

Journée problème de Poisson, IHP, Paris

M. Faverge, P. Ramet
Inria HiePACS team

Objectives: Contribute to the design of effective tools for frontier simulations arising from challenging research and industrial multi-scale applications towards exascale computing.

HiePACS: scientific structure
1

Introduction
Guideline

Introduction

Dense linear algebra

Sparse direct solver over runtimes: PaStiX
  On homogeneous shared memory nodes
  On heterogeneous shared memory nodes

Hybrid direct/sparse solver: MaPHyS

Conclusion
What are the modern architectures?

Today software developers face systems with:

- \( \sim 1 \) TFlop/s of compute power per node
- 32 cores or more per nodes, sometimes 100+ hardware threads
- Highly heterogeneous architectures (cores + specialized cores + accelerators/co-processors)
- Deep memory hierarchies
- And also distributed

Consequences of our programming paradigms are systemic noise, load imbalance, overheads (< 70% peak on DLA)
How to harness these devices productively?

- Certainly not the way we did it so far
- We need to shift our focus on:
  - Power (forget GHz think MW)
  - Cost (forget FLOPS think data movement)
  - Concurrency (not linear anymore)
  - Memory Scaling (compute grows twice faster than memory bandwidth)
  - Heterogeneity (it’s not going away)
  - Reliability (it is supposed to fail one day).
A short history of computing paradigms
A short history of computing paradigms
A short history of computing paradigms
A short history of computing paradigms

- MPI
- MPI + X
- MPI + X + Y
- MPI + X + Y + FT
A short history of computing paradigms
**Task-based programming**

- Focus on concepts, data dependencies and flows
- Don’t develop for an architecture but for an impermeable portability layer
- Let the runtime deal with the hardware characteristics
- But provide some flexibility if possible
- StarSS, Swift, Parallax, Quark, XKaapi, StarPU, PARSEC, ...
Task-based programming

▶ Workflows, bag-of-task, grid are similar concepts but at another level of granularity (we’re talking tens of microseconds)

▶ Time constraint: Reduces the opportunity for complex dynamic scheduling strategies, but increases the probability to do the wrong thing

▶ Separation of concerns: each layer with it’s own issues
  ▶ Humans focus on depicted the flow of data between tasks
  ▶ Compiler focus on tasks, problems that fit in one memory level
  ▶ And a runtime orchestrate the flows to maximize the throughput of the architecture
2
Dense linear algebra
FOR k = 0 .. SIZE - 1
    A[k][k], T[k][k] <- GEQRT( A[k][k] )
    FOR m = k+1 .. SIZE - 1
        A[k][k]|Up, A[m][k], T[m][k] <- TSQRT( A[k][k]|Up, A[m][k], T[m][k] )
    FOR n = k+1 .. SIZE - 1
        A[k][n] <- UNMQR( A[k][k]|Low, T[k][k], A[k][n] )
        FOR m = k+1 .. SIZE - 1
            A[k][n], A[m][n] <- TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )
From sequential code to DAG

```c
for (k = 0; k < A.mt; k++) {
    Insert_Task( zgeqrt, A[k][k], INOUT,
                 T[k][k], OUTPUT);
    for (m = k+1; m < A.mt; m++) {
        Insert_Task( ztsqrt, A[k][k], INOUT |
                     REGION_D | REGION_U,
                    A[m][k], INOUT |
                     LOCALITY,
                    T[m][k], OUTPUT);
    }
    for (n = k+1; n < A.nt; n++) {
        Insert_Task( zunmqr, A[k][k], INPUT |
                     REGION_L,
                   T[k][k], INPUT,
                    A[k][m], INOUT);
        for (m = k+1; m < A.mt; m++)
            Insert_Task( ztsmqr, A[k][n], INOUT, |
                         A[m][n], INOUT |
                         LOCALITY,
                        A[m][k], INPUT,
                        T[m][k], INPUT);
    }
}
```

- Sequential C code
- Annotated through some specific syntax
  - `Insert_Task`
  - `INOUT, OUTPUT, INPUT`
  - `REGION_L, REGION_U, REGION_D, ...
  - `LOCALITY`
Dense linear algebra

From sequential code to DAG
Automatic dataflow analysis
Dense linear algebra

From sequential code to DAG

Automatic dataflow analysis

GEQRT(k)
/* Execution space */
k = 0..( MT < NT ) ? MT-1 : NT-1
/* Locality */
: A(k, k)
RW A <- (k == 0) ? A(k, k)

: A1 TSMQR(k-1, k, k)
-> (k < NT-1) ? A UNMQR(k, k+1 .. NT-1) [type = LOWER]
-> (k < MT-1) ? A1 TSQRT(k, k+1) [type = UPPER]
-> (k == MT-1) ? A(k, k) [type = UPPER]

WRITE T <- T(k, k)
-> T(k, k)
-> (k < NT-1) ? T UNMQR(k, k+1 .. NT-1)
/* Priority */
; (NT-k)*(NT-k)*(NT-k)

BODY
zgeqrt( A, T )
END
Parameterized Task Graph VS Sequential Task Flow

Advantages of PTG

- Local view of the DAG
- No need to unroll the full graph
- Full knowledge of Input/Output for each task
- Strong separation of concerns (Data/task distribution, algorithm)

Advantages of STF

- Knowledge of the full graph for scheduling strategies
- Handle dynamic representation of the algorithm
- Submission process can be postponed to hang on data values (convergence criteria)
- More simple to implement than PTG
Dense linear algebra

QR factorization with Chameleon (StarPU)
Heterogeneous shared memory node

![Graph showing performance of QR factorization with different configurations](image_url)
QR factorization with Chameleon (StarPU)

Heterogeneous shared memory node

+ 200 GFlop/s but 12 cores = 150 GFlop/s
Dense linear algebra

QR factorization with DPLASMA (PaRSEC)

Distributed homogeneous memory nodes (60 × 8 cores)
Dense linear algebra

Libraries for solving large dense linear systems

Chameleon (http://project.inria.fr/chameleon)

- Inria HiePACS/Runtime
- StarPU Runtime
- STF programing model
- Smart/Complex scheduling

DPLASMA (http://icl.cs.utk.edu/dplasma)

- ICL, University of Tennessee
- PaRSEC Runtime
- PTG model
- Simple scheduling based on data locality
Libraries for solving large dense linear systems

Chameleon (http://project.inria.fr/chameleon)

- Inria HiePACS/Runtime
- StarPU Runtime
- STF programing model
- Smart/Complex scheduling

DPLASMA (http://icl.cs.utk.edu/dplasma)

- ICL, University of Tennessee
- PaRSEC Runtime
- PTG model
- Simple scheduling based on data locality

What about irregular problems as sparse solvers?
3
Sparse direct solver over runtimes: PaStiX
3.1 Sparse direct solver over runtimes: PaStiX
On homogeneous shared memory nodes
Sparse direct solver over runtimes: PaStiX
On homogeneous shared memory nodes

Tasks structure

(a) Dense tile task decomposition
(b) Decomposition of the task applied while processing one panel
Sparse direct solver over runtimes: PaStiX
On homogeneous shared memory nodes

DAG representation

(c) Dense DAG

(d) Sparse DAG representation of a sparse $LDL^T$ factorization
Supernodal sequential algorithm

```plaintext
forall the Supernode $S_1$ do
    panel ($S_1$);
    /* update of the panel */
    forall the extra diagonal block $B_i$ of $S_1$ do
        $S_2$ ← supernode_in_front_of ($B_i$);
        gemm ($S_1$, $S_2$);
        /* sparse GEMM $B_{k,k \geq i} \times B_i^T$ substracted from $S_2$ */
    end
end
```
**StarPU Tasks submission**

```plaintext
forall the Supernode $S_1$ do
    submit_panel ($S_1$);
    /* update of the panel */
    forall the extra diagonal block $B_i$ of $S_1$ do
        $S_2$ ← supernode_in_front_of ($B_i$);
        submit_gemm ($S_1$, $S_2$);
        /* sparse GEMM $B_{k,k\geq i} \times B_i^T$ subtracted from $S_2$ */
    end
end
wait_for_all_tasks ();
```
PARSEC’s parameterized task graph

Panel Factorization in JDF Format

Panel Factorization in JDF Format

Sparse direct solver over runtimes: PaStiX  On homogeneous shared memory nodes
Matrices and Machines

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Prec</th>
<th>Method</th>
<th>Size</th>
<th>nnz_A</th>
<th>nnz_L</th>
<th>TFlop</th>
</tr>
</thead>
<tbody>
<tr>
<td>FilterV2</td>
<td>Z</td>
<td>LU</td>
<td>0.6e+6</td>
<td>12e+6</td>
<td>536e+6</td>
<td>3.6</td>
</tr>
<tr>
<td>Flan</td>
<td>D</td>
<td>LL^T</td>
<td>1.6e+6</td>
<td>59e+6</td>
<td>1712e+6</td>
<td>5.3</td>
</tr>
<tr>
<td>Audi</td>
<td>D</td>
<td>LL^T</td>
<td>0.9e+6</td>
<td>39e+6</td>
<td>1325e+6</td>
<td>6.5</td>
</tr>
<tr>
<td>MHD</td>
<td>D</td>
<td>LU</td>
<td>0.5e+6</td>
<td>24e+6</td>
<td>1133e+6</td>
<td>6.6</td>
</tr>
<tr>
<td>Geo1438</td>
<td>D</td>
<td>LL^T</td>
<td>1.4e+6</td>
<td>32e+6</td>
<td>2768e+6</td>
<td>23</td>
</tr>
<tr>
<td>Pmldf</td>
<td>Z</td>
<td>LDL^T</td>
<td>1.0e+6</td>
<td>8e+6</td>
<td>1105e+6</td>
<td>28</td>
</tr>
<tr>
<td>Hook</td>
<td>D</td>
<td>LU</td>
<td>1.5e+6</td>
<td>31e+6</td>
<td>4168e+6</td>
<td>35</td>
</tr>
<tr>
<td>Serena</td>
<td>D</td>
<td>LDL^T</td>
<td>1.4e+6</td>
<td>32e+6</td>
<td>3365e+6</td>
<td>47</td>
</tr>
</tbody>
</table>

Table: Matrix description (Z: double complex, D: double).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processors</th>
<th>Frequency</th>
<th>GPUs</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mirage</td>
<td>Westmere Intel Xeon X5650 (2 × 6)</td>
<td>2.67 GHz</td>
<td>Tesla M2070 (×3)</td>
<td>36 GB</td>
</tr>
</tbody>
</table>
CPU scaling study: GFlop/s during numerical factorization

![Performance Chart]

- **native**
- **STARPU**
- **PARSEC**

<table>
<thead>
<tr>
<th></th>
<th>native</th>
<th>STARPU</th>
<th>PARSEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 core</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 cores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 cores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 cores</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 cores</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Graph Details**
- Performance measured in GFlop/s.
- Comparison across different numbers of cores for various applications.
- Applications include FilterV2(Z, LU), Flan(D, LL^T), audi(D, LL^T), MHD(D, LU), Geo1438(D, LL^T), pmlDF(Z, LDL^T), HOOK(D, LU), Serena(D, LDL^T).
Sparse direct solver over runtimes: PaStiX on homogeneous shared memory nodes

CPU scaling study: GFlop/s during numerical factorization

![Performance graph showing CPU scaling study for different problems and algorithms.](image)

- FilterV2(Z, LU)
- Flan(D, LLᵀ)
- audi(D, LLᵀ)
- MHD(D, LU)
- Geo1438(D, LLᵀ)
- pmIDF(Z, LDLᵀ)
- HOOK(D, LU)
- Serena(D, LDLᵀ)

Performance (GFlop/s)

native  STARPU  PARSEC
1 core  1 core  1 core
3 cores 3 cores 3 cores
6 cores 6 cores 6 cores
9 cores 9 cores 9 cores
12 cores 12 cores 12 cores
CPU scaling study: GFlop/s during numerical factorization

![Chart showing performance comparison between different solvers and configurations]
CPU scaling study: GFlop/s during numerical factorization

- FilterV2(Z, LU)
- Flan(D, LL^T)
- audi(D, LL^T)
- MHD(D, LU)
- Geo1438(D, LL^T)
- pmlDF(Z, LDL^T)
- HOOK(D, LU)
- Serena(D, LDL^T)
CPU scaling study: GFlop/s during numerical factorization

![Graph showing CPU scaling study for various applications and toolboxes.](image-url)
3.2 Sparse direct solver over runtimes: PaStiX

On heterogeneous shared memory nodes
Why using GPUs on sparse problems?

- updates are compute intensive
- small updates
- non continuous data
- rectangular shape

Compacted in memory
GPU kernel performance

Figure: Performance comparison on the DGEMM kernel for three implementations: cuBLAS library, ASTRA framework, and the sparse adaptation of the ASTRA framework.
**GPU kernel performance - 2 Streams**

![Graph showing performance comparison on the DGEMM kernel for three implementations: cuBLAS library, ASTRA framework, and the sparse adaptation of the ASTRA framework.](image)

**Figure:** Performance comparison on the DGEMM kernel for three implementations: cuBLAS library, ASTRA framework, and the sparse adaptation of the ASTRA framework.
GPU kernel performance - 3 Streams

Figure: Performance comparison on the DGEMM kernel for three implementations: cuBLAS library, ASTRA framework, and the sparse adaptation of the ASTRA framework.
GPU scaling study: GFlop/s for numerical factorization

![Graph showing performance (GFlop/s) for various combinations of CPU and GPU usage. The graph includes data for Native, StarPU, and PARSEC with 1 and 3 streams. Each category has bars for different solvers (FilterV2, Flan, audi, MHD, Geo1438, pmlDF, HOOK, Serena) and indicates performance for 1, 2, and 3 GPUs.](image-url)
GPU scaling study: GFlop/s for numerical factorization

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Performance (GFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native: CPU only</td>
<td></td>
</tr>
<tr>
<td>STARPU: CPU only</td>
<td></td>
</tr>
<tr>
<td>PARSEC 1 stream:</td>
<td></td>
</tr>
<tr>
<td>PARSEC 3 streams:</td>
<td></td>
</tr>
</tbody>
</table>

FilterV2(Z, LU) | 100  | 150  | 200  |
Flan(D, LLᵀ)    |      |      |      |
audi(D, LLᵀ)    | 50   | 100  | 150  |
MHD(D, LU)      | 25   | 50   | 75   |
Geo1438(D, LLᵀ) |      |      |      |
pmDF(Z, LDLᵀ)   | 75   | 150  | 225  |
HOOK(D, LU)     |      |      |      |
Serena(D, LDLᵀ) | 50   | 100  | 150  |
GPU scaling study: GFlop/s for numerical factorization

Performance (GFlop/s)

Native: CPU only
STARPU: CPU only 1 GPU 2 GPU 3 GPU
PARSEC 1 stream: CPU only 1 GPU 2 GPU 3 GPU
PARSEC 3 streams: 1 GPU 2 GPU 3 GPU

FilterV2(Z, LU)
Flan(D, LL^T)
audi(D, LL^T)
MHD(D, LU)
Geo1438(D, LL^T)
pmlDF(Z, LDL^T)
HOOK(D, LU)
Serena(D, LDL^T)

M. Faverge - Journée problème de Poisson

January 26, 2015 - 32
GPU scaling study: GFlop/s for numerical factorization

Performance (GFlop/s)

- Native: CPU only
- StarPU: CPU only
- PARSEC 1 stream: 1 GPU, 2 GPUs, 3 GPUs
- PARSEC 3 streams: 1 GPU, 2 GPUs, 3 GPUs

Sparse direct solver over runtimes: PaStiX
On heterogeneous shared memory nodes

M. Faverge - Journée problème de Poisson
Sparse direct solver over runtimes: PaStiX
On heterogeneous shared memory nodes

GPU scaling study: GFlop/s for numerical factorization

<table>
<thead>
<tr>
<th></th>
<th>Native:</th>
<th>STARPU:</th>
<th>PARSEC 1 stream:</th>
<th>PARSEC 3 streams:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU only</td>
<td>CPU only</td>
<td>1 GPU</td>
<td>2 GPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 GPU</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
<td>2 GPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 GPU</td>
<td></td>
</tr>
<tr>
<td>FilterV2(Z, LU)</td>
<td>CPU only</td>
<td>CPU only</td>
<td>1 GPU</td>
<td>2 GPU</td>
</tr>
<tr>
<td>Flan(D, LLT)</td>
<td></td>
<td></td>
<td>3 GPU</td>
<td></td>
</tr>
<tr>
<td>audi(D, LLT)</td>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
</tr>
<tr>
<td>MHD(D, LU)</td>
<td></td>
<td></td>
<td>2 GPU</td>
<td></td>
</tr>
<tr>
<td>Geo1438(D, LLT)</td>
<td></td>
<td></td>
<td>3 GPU</td>
<td></td>
</tr>
<tr>
<td>pm1DF(Z, LDLT)</td>
<td></td>
<td></td>
<td></td>
<td>1 GPU</td>
</tr>
<tr>
<td>HOOK(D, LU)</td>
<td></td>
<td></td>
<td>2 GPU</td>
<td></td>
</tr>
<tr>
<td>Serena(D, LDLT)</td>
<td></td>
<td></td>
<td>3 GPU</td>
<td></td>
</tr>
</tbody>
</table>

Performance (GFlop/s)
4

Hybrid direct/sparse solver: MaPHyS
Sparse linear solvers

Goal: solving $Ax = b$, where $A$ is sparse

Hybrid direct/sparse solver: MaPHyS

Usual trades off

Direct
- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

Iterative
- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Sparse linear solvers

Goal: solving $Ax = b$, where $A$ is sparse

Usual trades off

Direct
- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

Iterative
- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Sparse linear solvers

Goal: solving $Ax = b$, where $A$ is sparse

Usual trades off

**Direct**
- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

**Iterative**
- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Sparse linear solvers

Goal: solving $A x = b$, where $A$ is sparse

Usual trades off

**Direct**
- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

**Iterative**
- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Sparse linear solvers

Goal: solving $Ax = b$, where $A$ is sparse

Usual trades off

**Direct**
- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

**Iterative**
- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Sparse linear solvers

Goal: solving $A x = b$, where $A$ is sparse

Usual trades off

Direct

- Robust/accurate for general problems
- BLAS-3 based implementations
- Memory/CPU prohibitive for large 3D problems
- Limited weak scalability

Iterative

- Problem dependent efficiency / accuracy
- Sparse computational kernels
- Less memory requirements and possibly faster
- Possible high weak scalability
Hybrid Linear Solvers

Develop robust scalable parallel hybrid direct/iterative linear solvers

▶ Exploit the efficiency and robustness of the sparse direct solvers
▶ Develop robust parallel preconditioners for iterative solvers
▶ Take advantage of the natural scalable parallel implementation of iterative solvers

Domain Decomposition (DD)

▶ Natural approach for PDE’s
▶ Extend to general sparse matrices
▶ Partition the problem into subdomains, subgraphs
▶ Use a direct solver on the subdomains
▶ Robust preconditioned iterative solver
MaPHyS

Global algebraic view

- Global hybrid decomposition:
  \[ A = \begin{pmatrix} A_{II} & A_{II} \\ A_{II} & A_{GG} \end{pmatrix} \]

- Global Schur complement:
  \[ S = A_{GG} - A_{II} A_{II}^{-1} A_{II} \]
MaPHyS

Local algebraic view

- Local hybrid decomposition:

\[ A^{(i)} = \begin{pmatrix} A_{I_i I_i} & A_{I_i \Gamma_i} \\ A_{\Gamma_i I_i} & A^{(i)}_{\Gamma\Gamma} \end{pmatrix} \]

- Local Schur Complement:

\[ S^{(i)} = A_{\Gamma\Gamma}^{(i)} - A_{I_i I_i} A_{I_i I_i}^{-1} A_{I_i \Gamma_i} \]

- Algebraic Additive Schwarz Preconditioner

\[ \mathcal{M} = \sum_{i=1}^{N} R_{\Gamma_i}^T (\bar{S}^{(i)})^{-1} R_{\Gamma_i} \]
Parallel implementation

- Each subdomain $A^{(i)}$ is handled by one CPU core
  $$A^{(i)} = \begin{pmatrix} A_{I_i I_i} & A_{I_i \Gamma_i} \\ A_{I_i \Gamma_i} & A^{(i)}_{\Gamma \Gamma} \end{pmatrix}$$

- Concurrent partial factorizations are performed on each CPU core to form the so called “local Schur complement”
  $$S^{(i)} = A^{(i)}_{\Gamma \Gamma} - A_{I_i I_i}A^{-1}_{I_i I_i}A_{I_i \Gamma_i}$$

- The reduced system $Sx_\Gamma = f$ is solved using a distributed Krylov solver
  - One matrix vector product per iteration each CPU core computes
    $$S^{(i)}(x_\Gamma^{(i)})^k = (y^{(i)})^k$$
  - One local preconditioner apply $(M^{(i)})(z^{(i)})^k = (r^{(i)})^k$
  - Local neighbor-neighbor communication per iteration
  - Global reduction (dot products)

- Compute simultaneously the solution for the interior unknowns
  $$A_{I_i I_i}x_{I_i} = b_{I_i} - A_{I_i \Gamma_i}x_{\Gamma_i}$$
Modular implementation

Sequential partitioners

▶ **SCOTCH** [Pellegrini et al.]
▶ **METIS** [G. Karypis and V. Kumar]

Sparse direct solvers - one-level parallelism

▶ **MUMPS** [P. Amestoy et al.] (with Schur option)
▶ **PaSTiX** [P. Ramet et al.] (with Schur option)

Iterative Solvers

▶ **CG/GMRES/FGMRES** [V. Fraysse and L. Giraud]
Modular implementation and **bottlenecks**

**Sequential partitioners**

- **SCOTCH** [Pellegrini et al.]
- **METIS** [G. Karypis and V. Kumar]

**Sparse direct solvers - one-level parallelism**

- **MUMPS** [P. Amestoy et al.] (with Schur option)
- **PaSTiX** [P. Ramet et al.] (with Schur option)

**Iterative Solvers**

- **CG/GMRES/FGMRES** [V.Fraysse and L.Giraud]
Modular implementation and bottlenecks

Sequential partitioners

- **Scotch** [Pellegrini et al.]
- **Metis** [G. Karypis and V. Kumar]

Sparse direct solvers - one-level parallelism

- **MUMPS** [P. Amestoy et al.] (with Schur option)
- **PaStiX** [P. Ramet et al.] (with Schur option)

Iterative Solvers

- **CG/GMRES/FGMRES** [V. Fraysse and L. Giraud]
Modular implementation and bottlenecks

Sequential partitioners

- **Scotch** [Pellegrini et al.]
- **Metis** [G. Karypis and V. Kumar]

Sparse direct solvers - one-level parallelism

- **Mumps** [P. Amestoy et al.] (with Schur option)
- **PaStiX** [P. Ramet et al.] (with Schur option)

Iterative Solvers

- **CG/GMRES/FGMRES** [V. Fraysse and L. Giraud]
MPI-thread on Hopper@Nersc with Nachos4M - Time

![Graph showing the time (s) versus the number of cores for different thread per process (t/p) configurations: 3 t/p, 6 t/p, 12 t/p, 24 t/p. The graph plots time on a logarithmic scale.]
MPI-thread on Hopper@Nersc with Nachos4M - Memory
5 Conclusion
Sparse direct solver: **PaStiX**

**Main Features**

- LLt, LDLt, LU: supernodal implementation (BLAS3)
- Static pivoting + Refinement: CG/GMRES/BiCGstab
- Column-block or Block mapping
- Simple/Double precision + Real/Complex arithmetic
- Support external ordering libraries ((PT-)Scotch or METIS ...)
- **MPI/Threads (Cluster/Multicore/SMP/NUMA)**
- Multiple GPUs using DAG runtimes
- X. Lacoste PhD defense on February 18th, 2015
- Multiple RHS (direct factorization)
- Incomplete factorization with ILU(k) preconditionner
- Schur complement computation
- C++/Fortran/Python/PETSc/Trilinos/FreeFem/...
Sparse direct solver: \texttt{PaStiX}

What next?

- Exploiting DAG runtimes:
  - distributed memory architectures
  - other accelerators: Intel Xeon Phi
- Ordering methods for the supernodes:
  Reduce the number of extra diagonal blocks
- H-matrix arithmetic (A. Buttari’s talk):
  In collaboration with Univ. of Stanford
Sparse hybrid solver: \textbf{MaPHYs}  
Current status and future

- **Sequential version:**  
  → Original \textit{MaPhyS} version

- **Multi-threaded implementation**  
  → Integrated during S. Nakov PhD

- **Hybrid MPI / Multi-threaded implementation**  
  → Is it worth it?

- **Task-flow programming model**  
  - L. Poirel PhD started in October 2014
  - Non blocking algorithms (W. Vanroose’s talk)

- **Exploiting $\mathcal{H}$ – matrix for the Shur complement**
Softwares

Graph/Mesh partitioner and ordering:

http://scotch.gforge.inria.fr

Sparse linear system solvers:

http://pastix.gforge.inria.fr

MaPHyS

https://wiki.bordeaux.inria.fr/maphys/doku.php
Softwares

Chameleon:

MORSE

http://icl.cs.utk.edu/projectsdev/morse
http://project.inria.fr/chameleon/

DPlasma:

http://icl.cs.utk.edu/dplasma/index.html
Thanks !