TASK-BASED SPARSE CHOLESKY SOLVER ON TOP OF RUNTIME SYSTEM

Iain S. Duff, Jonathan D. Hogg and **Florent Lopez**

Sparse Days, 2016

Rutherford Appleton Laboratory
NLAFET Project

Objective

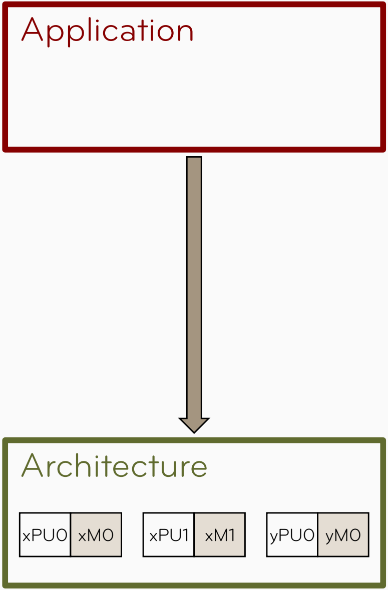
Solve $Ax = b$, where A is large and sparse, on modern architectures.

Using Direct Method: Sparse Cholesky factorization $A = LL^T$

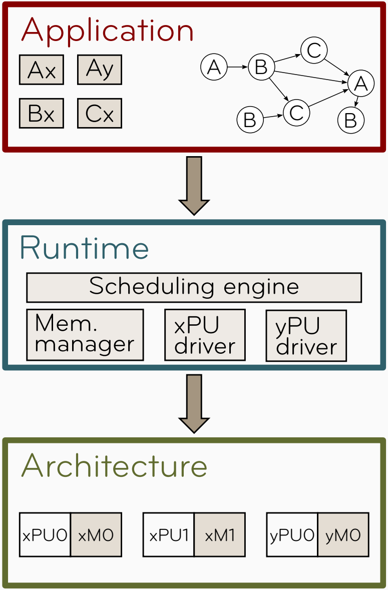
- ▲ Numerically robust and general purpose
- ▼ High memory usage and computational cost

Exploiting modern platforms is challenging:

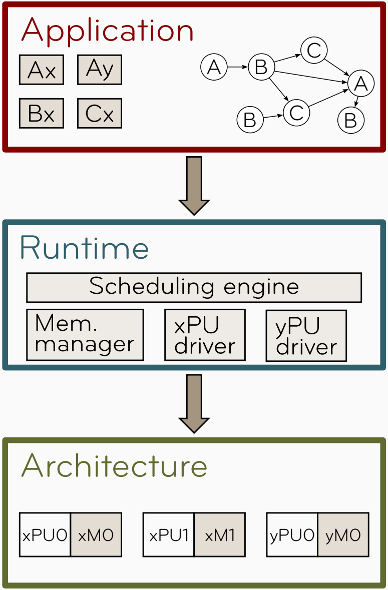
- Multicore processors and deep memory hierarchy.
- Heterogeneous e.g. CPU & GPU or Xeon Phi.
- Distributed-memory systems.



- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.



- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.
- runtimes provide an abstraction layer that hides the architecture details.

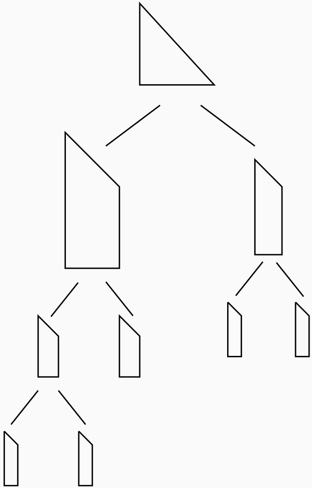


- The classical approach is based on a mixture of technologies (e.g., MPI+OpenMP+CUDA) which.
  - requires a big programming effort.
  - is difficult to maintain and update.
  - is prone to (performance) portability issues.
- runtimes provide an abstraction layer that hides the architecture details.
- the workload is expressed as a DAG of tasks.

In numerical factorization of A the *elimination tree* expresses data dependencies in the factor L. Each node, referred to as *supernode*, is a dense lower trapezoidal submatrix of L.

The tree is traversed in a topological order, and each node is factorised using dense Cholesky algorithm.

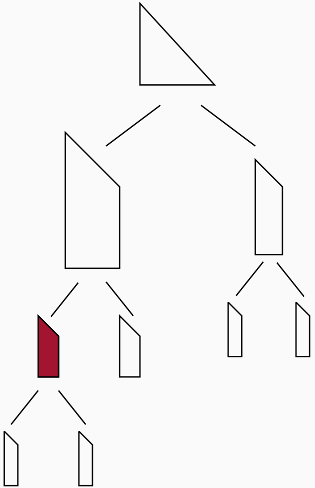
Updates between node are handled using a supernodal scheme i.e. updates are applied directly to the target supernodes.



In numerical factorization of A the *elimination tree* expresses data dependencies in the factor L. Each node, referred to as *supernode*, is a dense lower trapezoidal submatrix of L.

The tree is traversed in a topological order, and each node is factorised using dense Cholesky algorithm.

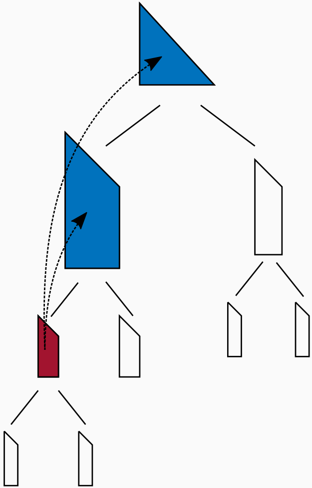
Updates between node are handled using a supernodal scheme i.e. updates are applied directly to the target supernodes.



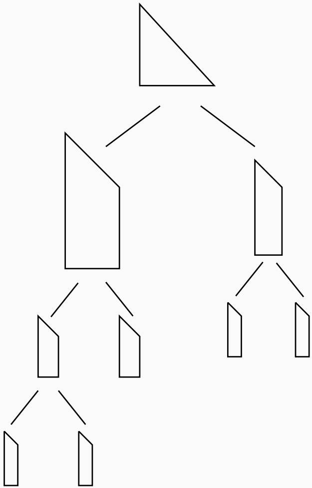
In numerical factorization of *A* the *elimination tree* expresses data dependencies in the factor *L*. Each node, referred to as *supernode*, is a dense lower trapezoidal submatrix of *L*.

The tree is traversed in a topological order, and each node is factorised using dense Cholesky algorithm.

Updates between node are handled using a supernodal scheme i.e. updates are applied directly to the target supernodes.

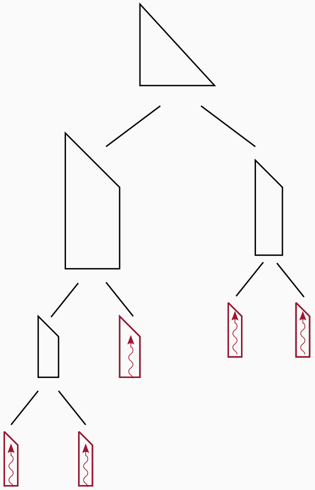


Sources of parallelism in the elimination tree:



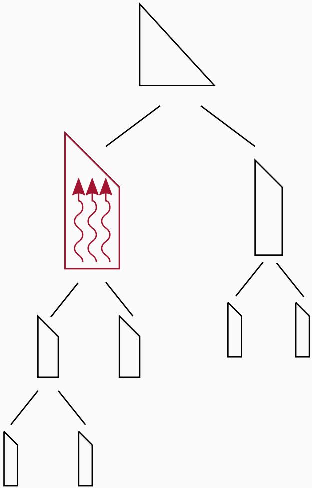
Sources of parallelism in the elimination tree:

- Tree parallelism: Supernode in independent branches can be processed concurrently.

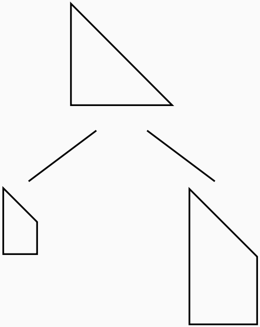


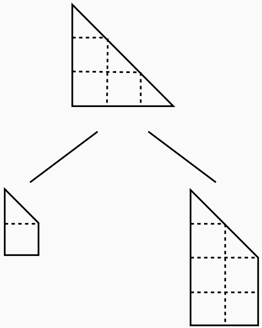
Sources of parallelism in the elimination tree:

- Tree parallelism: Supernode in independent branches can be processed concurrently.
- Node parallelism: When a supernode is large enough, it may be processed in parallel.



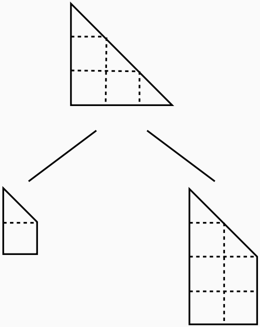
Task-based Sparse Cholesky factorization

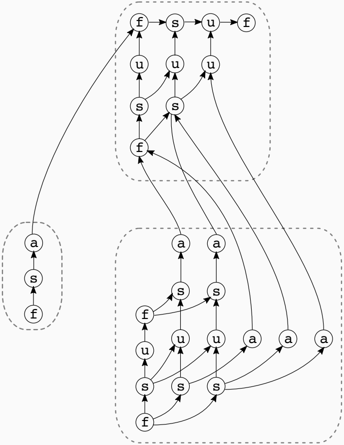




Supernodes are partitioned into square blocks (`nb x nb`) on which operations are applied.

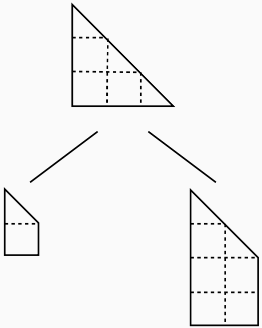
Task-based Sparse Cholesky factorization

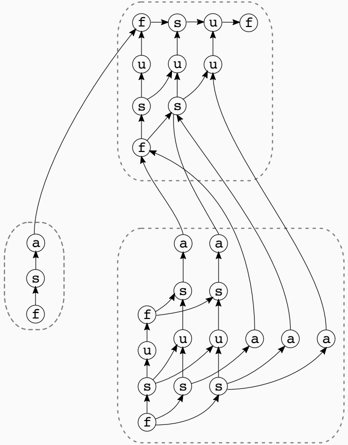




Supernodes are partitioned into square blocks (**nb x nb**) on which operations are applied. The DAG replaces the elimination tree for representing the dependencies.

Task-based Sparse Cholesky factorization





Supernodes are partitioned into square blocks (`nb x nb`) on which operations are applied. The DAG replaces the elimination tree for representing the dependencies.
Implemented in the HSL package MA87.

Task-based Sparse Cholesky factorization

```
forall nodes snode in post-order
   call alloc(snode) ! allocate data structures

   call init(snode) ! initianlize node structure
end do

forall nodes snode in post-order
  ! factorize node
   call factorize(snode) ! factorize block

   ! update ancestor nodes
   forall ancestors(snode) anode
     call update_btw(snode, anode)
   end do

  end do
end do
```

Task-based Sparse Cholesky factorization

```
forall nodes snode in post-order
   call alloc(snode) ! allocate data structures

   call init(snode) ! initianlize node structure
end do

forall nodes snode in post-order
  ! factorize node
  do k=1..n in snode
    call factorize(blk(k,k)) ! factorize block

    do i=k+1..m in snode
        call solve(blk(k,k), blk(i,k)) ! perform solve
    end do

    do j=k+1..n in snode
       do i=k+1..m in snode
          call update(blk(j,k), blk(i,k), blk(i,j))
       end do
    end do

    ! update ancestor nodes
    forall ancestors(snode) anode
      do j=k+1..p(anode) in snode
         do i=k+1..m in snode
            call update_btw(blk(j,k), blk(i,k), a_blk(rmap(i), cmap(j)))
         end do
      end do
    end do

  end do
end do
```

Sequential Task Flow (STF) programming model:

- Tasks are submitted to the runtime system following the sequential algorithm.
- The runtime analyses manipulated data and infers task dependencies in order to ensure the sequential consistency of the parallel code.
- The DAG is executed via a dynamic scheduling of the (ready) tasks on the architectures.
- The runtime may be capable of automatically handling the data transfer across the architecture.
- Superscalar analysis in processors: dependency detection between instructions in order to issue them in parallel.

```
forall nodes snode in post-order
   call alloc(snode) ! allocate data structures

   call init(snode) ! initianlize node structure
end do

forall nodes snode in post-order
  ! factorize node
  do k=1..n in snode
    call factorize(blk(k,k)) ! factorize block

    do i=k+1..m in snode
        call solve(blk(k,k), blk(i,k)) ! perform solve
    end do

    do j=k+1..n in snode
       do i=k+1..m in snode
          call update(blk(j,k), blk(i,k), blk(i,j))
       end do
    end do

    ! update ancestor nodes
    forall ancestors(snode) anode
      do j=k+1..p(anode) in snode
         do i=k+1..m in snode
            call update_btw(blk(j,k), blk(i,k), a_blk(rmap(i), cmap(j)))
         end do
      end do
    end do

  end do
end do
```

STF Sparse Cholesky Factorization

```
forall nodes snode in post-order
   call alloc(snode) ! allocate data structures

   call submit(init, snode:W) ! initianlize node structure
end do

forall nodes snode in post-order
  ! factorize node
  do k=1..n in snode
    call submit(factorize, snode:R, blk(k,k):RW) ! factorize block

    do i=k+1..m in snode
        call submit(solve, blk(k,k):R, blk(i,k):RW) ! perform solve
    end do

    do j=k+1..n in snode
       do i=k+1..m in snode
          call submit(update, blk(j,k):R, blk(i,k):R, blk(i,j):RW)
       end do
    end do

    ! update ancestor nodes
    forall ancestors(snode) anode
      do j=k+1..p(anode) in snode
         do i=k+1..m in snode
            call submit(update_btw, blk(j,k):R, blk(i,k):R, a_blk(rmap(i), cmap(j)):RW)
         end do
      end do
    end do

  end do
end do
call wait_for_all()
```

STF on top of Runtime System

OpenMP 4.0

- `task` construct and `depend` clause (`in`, `out`, `inout`).
- No control on the scheduling policy.
- Shared-memory system only.

StarPU

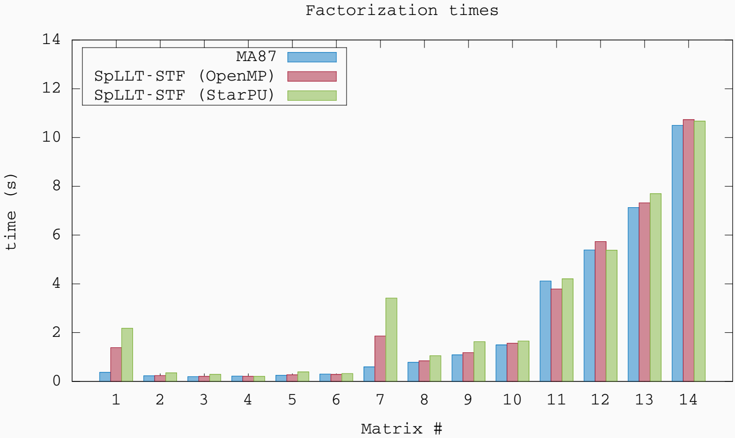
- `starpu_insert_task` and data *handle* with access mode (`R`, `W`, `RW`).
- Full control on schduling policy with possibility to implement new one.
- API for distributed-memory systems.

Experiments

#	Matrix	Flops (10^9)	Application/description
1	Schmid/thermal2	18.6	Unstructured thermal FEM
2	Rothberg/gearbox	22.8	Aircraft flap actuator
3	DNVS/m_t1	23.4	Tubular joint
4	DNVS/thread	35.7	Threaded connector
5	DNVS/shipsec1	40.5	Ship section
6	GHS_psdef/crankseg_2	48.8	Linear static analysis
7	AMD/G3_circuit	67.3	Circuit simulation
8	Koutsovasilis/F1	228	AUDI engine crankshaft
9	Oberwolfach/boneS10	297	Bone micro-FEM
10	ND/nd12k	514	3D mesh problem
11	JGD Trefethen/Trefethen_20000	669	Integer matrix
12	ND/nd24k	2080	3D mesh problem
13	Oberwolfach/bone010	3910	Bone micro-FEM
14	GHS_psdef/audikw_1	5840	Automotive crankshaft

- Symmetric positive-definite matrices.
- Metis nested disection ordering.
- Machine: 2 *x* 14 cores E5-2695 v3 (Haswell) @ 2.30GHz.

Experiments



- SpLLT and MA87 obtain similar performance most problems.
- In two cases (Matrices #1 and #7) the difference with MA87 is relatively big.

STF model: limitations

#	SpLLT				
	OpenMP		StarPU		MA87
	build (s)	facto (s)	build (s)	facto (s)	facto (s)
1	1.238	1.801	1.677	2.123	0.376
2	0.152	0.220	0.281	0.318	0.252
3	0.155	0.205	0.200	0.262	0.194
4	0.125	0.203	0.152	0.240	0.213
5	0.215	0.247	0.271	0.363	0.259
6	0.178	0.267	0.283	0.310	0.257
7	1.712	2.631	2.737	3.345	0.586
8	0.600	0.812	0.763	0.920	0.786
9	0.812	1.186	1.299	1.599	1.111
10	0.770	1.478	0.763	1.405	1.498
11	0.749	3.692	1.586	2.406	3.829
12	2.887	5.379	2.778	5.076	5.498
13	3.063	7.416	2.280	7.392	7.195
14	3.383	10.650	3.141	10.680	10.642

- In the STF model, depending on DAG size and granularity of tasks, the time spent for building the DAG might be important compared to the factorization time.

#	SpLLT				
	OpenMP		StarPU		MA87
	build (s)	facto (s)	build (s)	facto (s)	facto (s)
1	1.238	1.801	1.677	2.123	0.376
2	0.152	0.220	0.281	0.318	0.252
3	0.155	0.205	0.200	0.262	0.194
4	0.125	0.203	0.152	0.240	0.213
5	0.215	0.247	0.271	0.363	0.259
6	0.178	0.267	0.283	0.310	0.257
7	1.712	2.631	2.737	3.345	0.586
8	0.600	0.812	0.763	0.920	0.786
9	0.812	1.186	1.299	1.599	1.111
10	0.770	1.478	0.763	1.405	1.498
11	0.749	3.692	1.586	2.406	3.829
12	2.887	5.379	2.778	5.076	5.498
13	3.063	7.416	2.280	7.392	7.195
14	3.383	10.650	3.141	10.680	10.642

- In the STF model, depending on DAG size and granularity of tasks, the time spent for building the DAG might be important compared to the factorization time.

Parametrized Task Graph (PTG) programming model:

- Uses a compact representation of the DAG (problem size independent).
- The dataflow between tasks is explicitly encoded (i.e. task dependencies are explicitly given to the runtime system).
- The runtime handles the communications implicitly using the dataflow representation.

PTG vs. STF

▲ In the PTG model, the DAG is progressively unrolled during the execution following the execution of tasks in a distributed way.

▼ Data-flow programming is much less intuitive than STF programming.

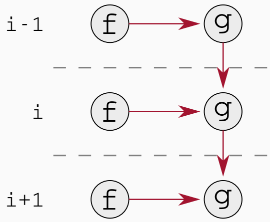
PTG: example

```
for (i = 1; i <= N; i++) {
  x[i] = f(x[i]);
  if (i > 1)
     y[i] = g(x[i], y[i-1]);
}
```

Simple squential code

```
for (i = 1; i <= N; i++) {
  x[i] = f(x[i]);
  if (i > 1)
     y[i] = g(x[i], y[i-1]);
}
```

Simple squential code

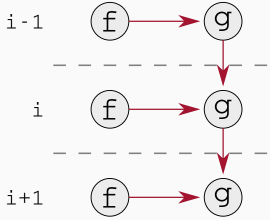


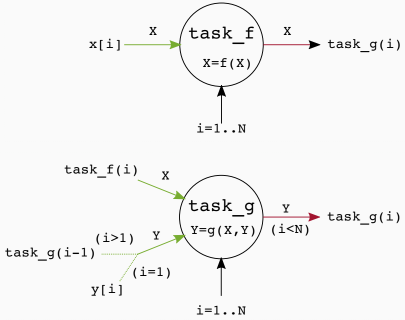
DAG

PTG: EXAMPLE

```
for (i = 1; i <= N; i++) {
  x[i] = f(x[i]);
  if (i > 1)
     y[i] = g(x[i], y[i-1]);
}
```

Simple squential code





DAG

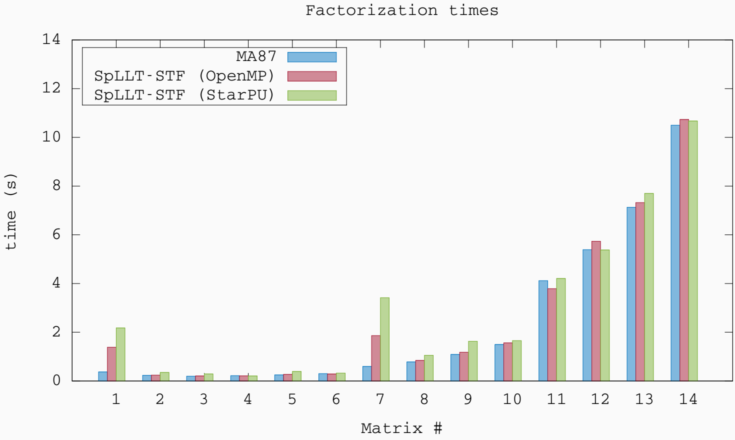
PTG representation

PTG Sparse Cholesky Factorization

We implemented a PTG-based version of SpLLT using PaRSEC which is one of the few runtime system supporting this model:

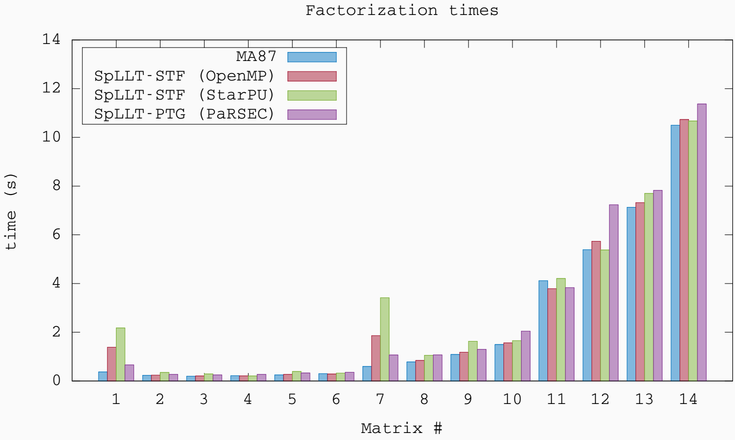
- In PaRSEC, The PTG code is written using a dedicated language: Job Data Flow (JDF).
- In a distributed-memory context, The runtime system is capable of handling iter-node communications implicitly.

Experiments



- Competitive performance compared to MA87 and OpenMP/StarPU codes.
- Better performance on matrices # 1 and # 7 compared to STF-based implementations but still not as good as MA87.

Experiments



- Competitive performance compared to MA87 and OpenMP/StarPU codes.
- Better performance on matrices # 1 and # 7 compared to STF-based implementations but still not as good as MA87.

- The runtime-based solver SpLLT gives competitive results compared to the hand-tuned HSL code MA87.
- Both OpenMP and StarPU versions offer good performance but we have seen some limitations of the STF model.
- The PTG version also offer good performance, it doesn't suffer from the same limitations as the STF-based codes but the code seems less efficient than the other version (runtime overhead ?).

Ongoing and Future work

- Run on distributed-memory systems: require to provide a data distribution to the runtime system.
- Run on GPU and Xeon Phi devices.
- Handle indefinite systems using pivoting techniques.

Thanks!

Questions?