SCHEDULING SPARSE SYMMETRIC FAN-BOTH CHOLESKY FACTORIZATION

Mathias Jacquelin
mjacquelin@lbl.gov
Esmond Ng, Kathy Yelick and Yili Zheng
egng|kayelick|yzheng@lbl.gov

May 18 2016

Scalable Solvers Group
Computational Research Department
Lawrence Berkeley National Laboratory

Background and motivation

`Fan-In`, `Fan-Out` and `Fan-Both` factorizations

Parallel distributed memory implementation, a.k.a. `symPACK`

Numerical experiments

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - ...
- Some sparse matrices are symmetric

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - ...
- Some sparse matrices are symmetric

Challenges for current and future platforms:

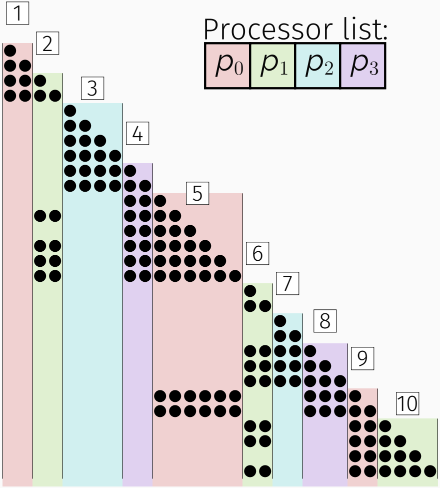
- Higher relative communication costs
- Lower amount of memory per core

Objective:

- Compute sparse $A = LL^T$ factorization
- A is sparse symmetric matrix
- A is positive definite

- Need to exploit symmetry

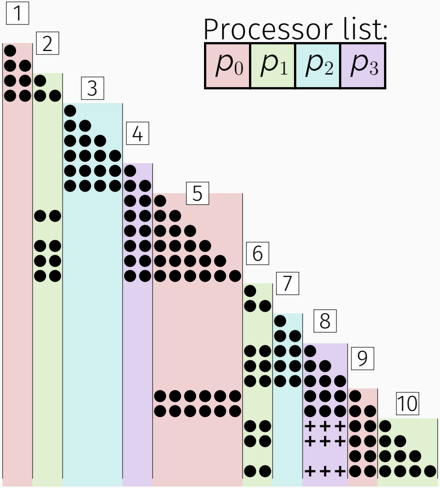
- L is a lower triangular matrix



- Fill in, $\Omega(A) \subseteq \Omega(L)$

$$A = LL^T$$

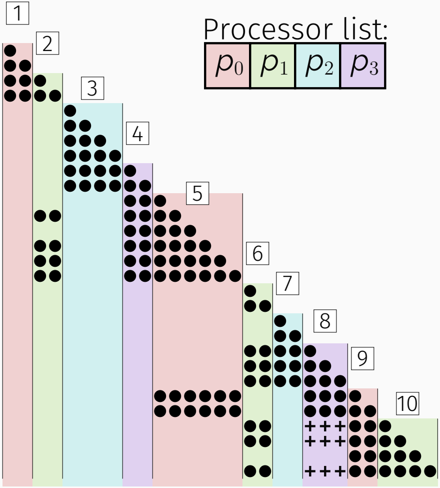
$\Omega(A)$ is the sparsity pattern of A

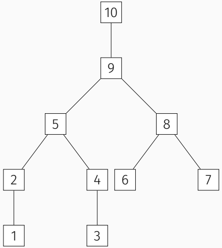


- Fill in, $\Omega(A) \subseteq \Omega(L)$

$$A = LL^T$$

$\Omega(A)$ is the sparsity pattern of A

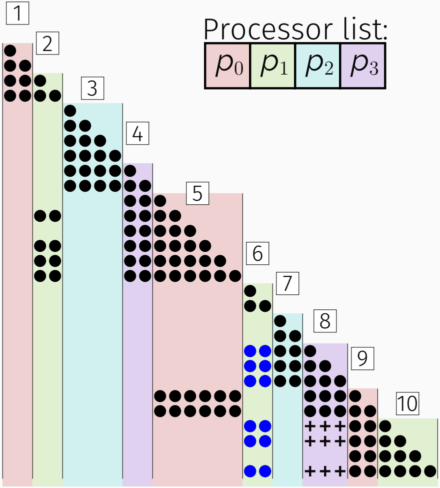


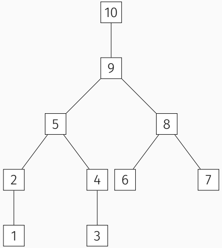


- Elim. tree represents column dependences
- Fill in, $\Omega(A) \subseteq \Omega(L)$

$$A = LL^T$$

$\Omega(A)$ is the sparsity pattern of A





- Elim. tree represents column dependences
- Fill in, $\Omega(A) \subseteq \Omega(L)$
- Supernode, same structure below diagonal block

$$A = LL^T$$

$\Omega(A)$ is the sparsity pattern of A

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})
    for row i = j + 1 to n do
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}
    end

    for column k = j + 1 to n do
        for row i = k to n do
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}
        end
    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})                 ⎫
    for row i = j + 1 to n do               ⎬ Factor column j
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}         ⎪
    end                                     ⎭

    for column k = j + 1 to n do
        for row i = k to n do
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}
        end
    end
end
```

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

for *column* $j = 1$ *to* n **do**
  $\ell_{j,j} = \sqrt{A_{j,j}}$
  for *row* $i = j + 1$ *to* n **do**
    $\ell_{i,j} = A_{i,j} / \ell_{j,j}$
  end
  } Factor column j

  for *column* $k = j + 1$ *to* n **do**
    for *row* $i = k$ *to* n **do**
      $A_{i,k} = A_{i,k} - \ell_{i,j} \cdot \ell_{k,j}$
    end
  end
  } Update next columns
end

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})                    } Factor column j
    for row i = j + 1 to n do                  }
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}             }
    end                                        }

    for column k = j + 1 to n do               } Update next columns
        for row i = k to n do                  } and Aggregate updates
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j} }
        end                                    }
    end                                        }
end
```

for *column* $j = 1$ *to* n **do**

- $\ell_{j,j} = \sqrt{A_{j,j}}$
- **for** *row* $i = j + 1$ *to* n **do**
  - $\ell_{i,j} = A_{i,j} / \ell_{j,j}$
- **end**

→ Factor column j

- **for** *column* $k = j + 1$ *to* n **do**
  - **for** *row* $i = k$ *to* n **do**
    - $A_{i,k} = A_{i,k} - \ell_{i,j} \cdot \ell_{k,j}$
  - **end**
- **end**

→ Update next columns and Aggregate updates

end

- Only lower triangular part of A is stored
- Basic algorithm:

Algorithm 1: Basic Cholesky algorithm

```
for column j = 1 to n do
    ℓ_{j,j} = sqrt(A_{j,j})                  } Factor column j
    for row i = j + 1 to n do
        ℓ_{i,j} = A_{i,j} / ℓ_{j,j}
    end

    for column k = j + 1 to n do             } Update next columns
        for row i = k to n do                  and Aggregate updates
            A_{i,k} = A_{i,k} − ℓ_{i,j} · ℓ_{k,j}    for row i = k to n do
        end                                        tmp_i = tmp_i + ℓ_{i,j} · ℓ_{k,j}
                                               end
    end                                        A_{*,k} = A_{*,k} − tmp_*
end
```

Factor column j: $\ell_{j,j} = \sqrt{A_{j,j}}$; for row $i = j+1$ to n do $\ell_{i,j} = A_{i,j}/\ell_{j,j}$ end

Update next columns and Aggregate updates:

for row $i = k$ to n do

$tmp_i = tmp_i + \ell_{i,j} \cdot \ell_{k,j}$

end

$A_{*,k} = A_{*,k} - tmp_*$

`fan-both` ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

fan-both ALGORITHM

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

Family determined by type of data exchanged

- Three families [Ashcraft'95]:
  - Fan-In: "fanning-in updates"
    - Reduce **aggregate vectors** (updates)
    - Factorize column
    - Compute all updates from *that column* locally
  - Fan-Out: "fanning-out factors"
    - Factorize column
    - Distribute the Cholesky factor
    - Compute and apply all updates to *my column*.

Family determined by type of data exchanged

Fan-In, Fan-Out $\subset$ Fan-Both

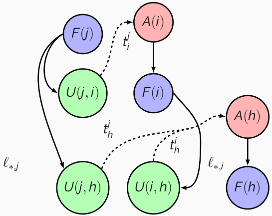
- Three families [Ashcraft'95]: **Fan-In, Fan-Out ⊂ Fan-Both**

- Task based algorithm:
  - A(i): accumulation of **aggregate vectors** (updates) to column *i*

    Reduces the aggregate vectors t_i^*
  - F(j): factorization of col. *j*

    Produces cholesky **factor** $\ell_{*,j}$
  - U(j,i): update of col. *i* with col. *j*

    Put the update in an (temporary) **aggregate vector** t_i^j



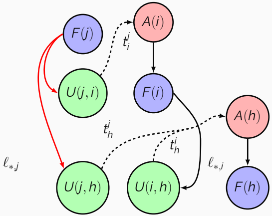
- Three families [Ashcraft'95]: **Fan-In, Fan-Out $\subset$ Fan-Both**

- Task based algorithm:
  - A(i): accumulation of **aggregate vectors** (updates) to column *i*

    Reduces the aggregate vectors t_i^*
  - F(j): factorization of col. *j*

    Produces cholesky **factor** $\ell_{*,j}$
  - U(j,i): update of col. *i* with col. *j*

    Put the update in an (temporary) **aggregate vector** t_i^j



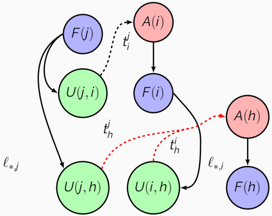
- Three families [Ashcraft'95]: **Fan-In, Fan-Out $\subset$ Fan-Both**

- Task based algorithm:
  - A(i): accumulation of **aggregate vectors** (updates) to column *i*

    Reduces the aggregate vectors t_i^*
  - F(j): factorization of col. *j*

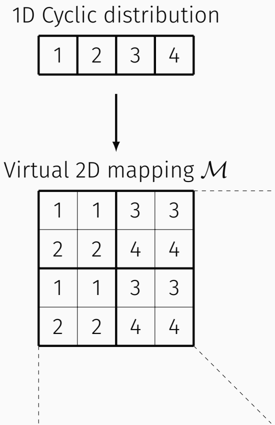
    Produces cholesky **factor** $\ell_{*,j}$
  - U(j,i): update of col. *i* with col. *j*

    Put the update in an (temporary) **aggregate vector** t_i^j



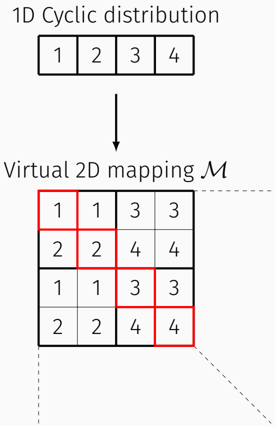
fan-both MAPPINGS

- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size



fan-both MAPPINGS

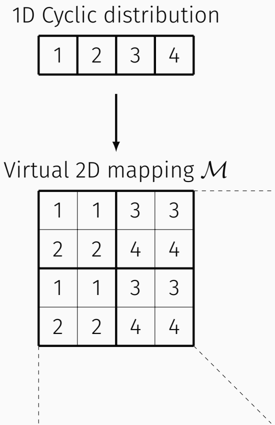
- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal



fan-both MAPPINGS

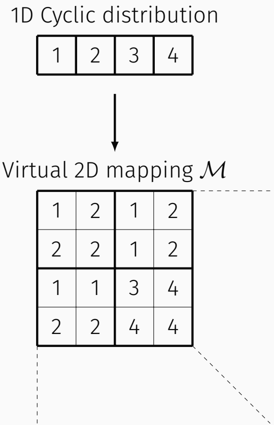
- How do we map tasks ?
  (independently of data)

- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings



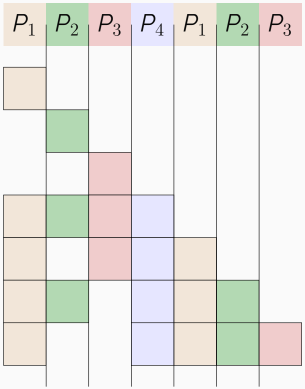
fan-both MAPPINGS

- How do we map tasks ?
  (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings

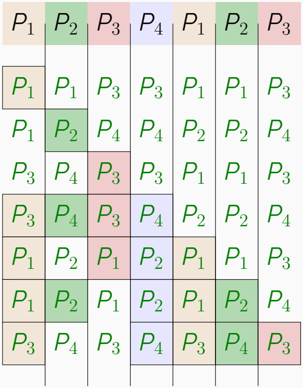


fan-both MAPPINGS

- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid "extends" to matrix size
  - Better if P processors on diagonal
  - Many possible mappings

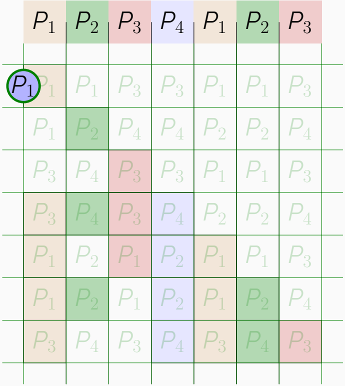


- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings



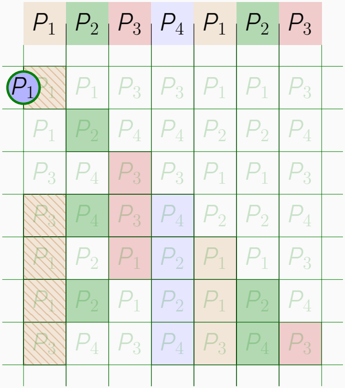
fan-both MAPPINGS

- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$



fan-both MAPPINGS

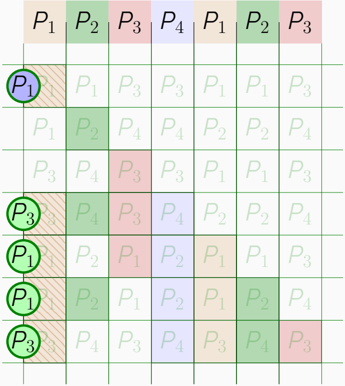
- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$



- How do we map tasks ?
  (independently of data)

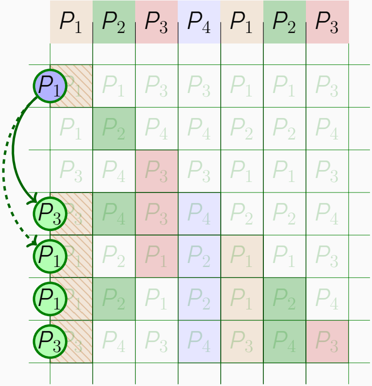
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings

  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$

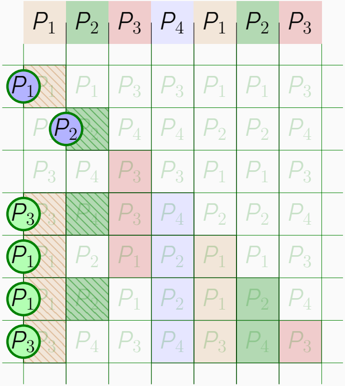


- How do we map tasks ? (independently of data)

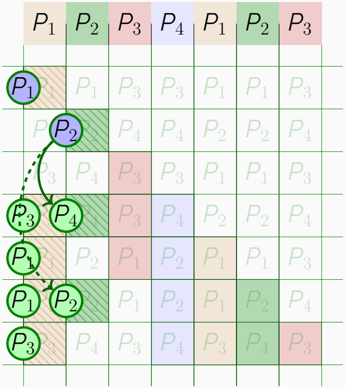
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$



- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$

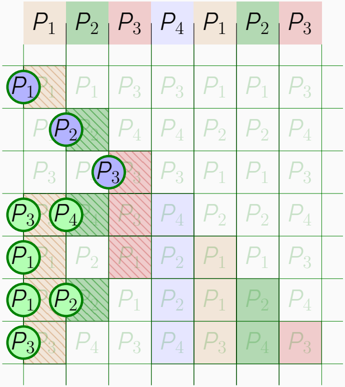


- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid "extends" to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$



fan-both MAPPINGS

- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$

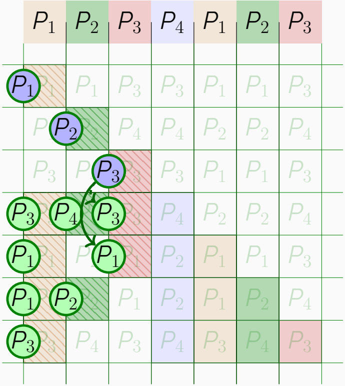


fan-both MAPPINGS

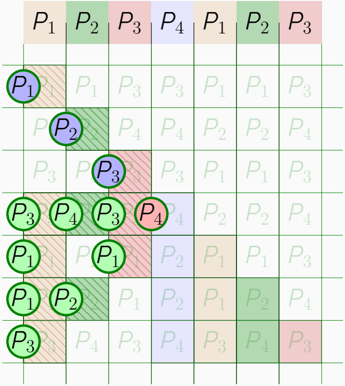
- How do we map tasks ? (independently of data)

- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid "extends" to matrix size
  - Better if P processors on diagonal
  - Many possible mappings

  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$

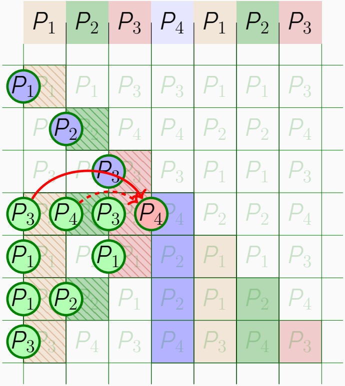


- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$
  - $A(j)$ on $\mathcal{M}(j,j)$

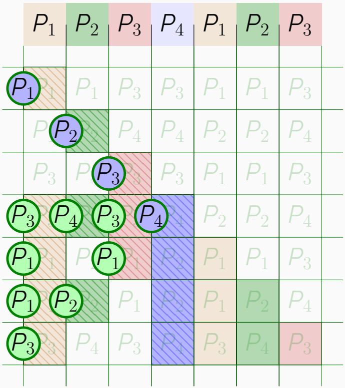


fan-both MAPPINGS

- How do we map tasks ?
  (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$
  - $A(j)$ on $\mathcal{M}(j,j)$



- How do we map tasks ? (independently of data)
- Use of 2D computation mapping grid $\mathcal{M}$
  - Mapping grid “extends” to matrix size
  - Better if P processors on diagonal
  - Many possible mappings
  - $F(i)$ on proc. $\mathcal{M}(i,i)$
  - $U(j,i)$ on $\mathcal{M}(j,i)$
  - $A(j)$ on $\mathcal{M}(j,j)$



0	1	2	3	0	1
0	1	2	3	0	1
0	1	2	3	0	1
0	1	2	3	0	1
0	1	2	3	0	1
0	1	2	3	0	1

`Fan-In`

$\mathcal{M}_{i,j} = mod(i, P)$

0	0	0	0	0	0
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3
0	0	0	0	0	0
1	1	1	1	1	1

`Fan-Out`

$\mathcal{M}_{i,j} = mod(j, P)$

0	0	2	2	0	0
1	1	3	3	1	1
0	0	2	2	0	0
1	1	3	3	1	1
0	0	2	2	0	0
1	1	3	3	1	1

`Fan-Both`

$$\mathcal{M}_{i,j} = \begin{array}{l} mod(\min(i,j), P) + \\ P\lfloor mod(\max(i,j), P)/P \rfloor \end{array}$$

Three different computation maps, corresponding to `Fan-In`, `Fan-Out` and `Fan-Both`

DESIGN PRINCIPLES

- Remove synchronization points
  - Asynchronous point to point send
  - Group communication:
    (MPI) Collectives probably not the way to go
    - Requires too many communicators
    - Efficient non blocking collectives needed
    - Collective nature

DESIGN PRINCIPLES

- Remove synchronization points
  - Asynchronous point to point send
  - Group communication:
    (MPI) Collectives probably not the way to go
    - Requires too many communicators
    - Efficient non blocking collectives needed
    - Collective nature

    - Asynchronous tree-based group communications
    - Non-collectives = full asynchronicity

DESIGN PRINCIPLES

- Remove synchronization points
  - Asynchronous point to point send
  - Group communication:
    (MPI) Collectives probably not the way to go
    - Requires too many communicators
    - Efficient non blocking collectives needed
    - Collective nature

    - Asynchronous tree-based group communications
    - Non-collectives = full asynchronicity

- Minimize memory operations
  - Row-major layout

- Remove synchronization points
  - Asynchronous point to point send
  - Group communication:
    (MPI) Collectives probably not the way to go
    - Requires too many communicators
    - Efficient non blocking collectives needed
    - Collective nature

    - Asynchronous tree-based group communications
    - Non-collectives = full asynchronicity
- Minimize memory operations
  - Row-major layout
  - Avoid making extra copies when sending data

DEADLOCK PREVENTION

- All operations described by task $T_{src \to tgt}$
- Message $Msg_{src \to tgt}$
- "Push" strategy natural with MPI

- All operations described by task $T_{src \to tgt}$
- Message $Msg_{src \to tgt}$
- “Push” strategy natural with MPI

Asynchronous comm. becomes blocking when out of buffer

DEADLOCK PREVENTION

- All operations described by task $T_{src \to tgt}$
- Message $Msg_{src \to tgt}$
- "Push" strategy natural with MPI

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

DEADLOCK PREVENTION

- All operations described by task $T_{src \to tgt}$
- Message $Msg_{src \to tgt}$
- "Push" strategy natural with MPI

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

- Deadlock prevention is difficult:
  - Total order in operations/messages

    (Also observed by Amestoy et al.)
  - Order by non decreasing *tgt*, then *src*:

    $\Rightarrow$ Use of priority queue for tasks/messages

DEADLOCK PREVENTION

- All operations described by task $T_{src \rightarrow tgt}$
- Message $Msg_{src \rightarrow tgt}$
- "Push" strategy natural with MPI

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

- Deadlock prevention is difficult:
  - Total order in operations/messages

    (Also observed by Amestoy et al.)
  - Order by non decreasing *tgt*, then *src*:

    $\Rightarrow$ Use of priority queue for tasks/messages

    Potential over-synchronization

DEADLOCK PREVENTION

- All operations described by task $T_{src \to tgt}$
- Message $Msg_{src \to tgt}$
- "Push" strategy natural with MPI

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

- Deadlock prevention is difficult:
  - Total order in operations/messages

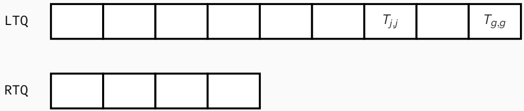
    (Also observed by Amestoy et al.)
  - Order by non decreasing *tgt*, then *src*:

    $\Rightarrow$ Use of priority queue for tasks/messages

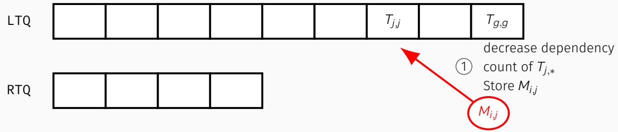
    Potential over-synchronization
- "Pull" strategy (one sided communications)
  - Signal data when available
  - Receiver gets data when ready

TASK SCHEDULING IN `sympack`

- Tasks $T_{src \rightarrow tgt}$
- Tasks currently mapped statically
- Processor manages local task queue `LTQ`

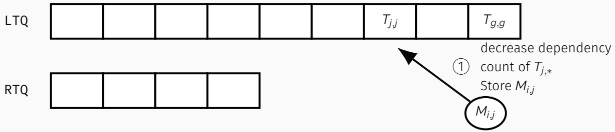


- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue `LTQ`
    - Dependency count

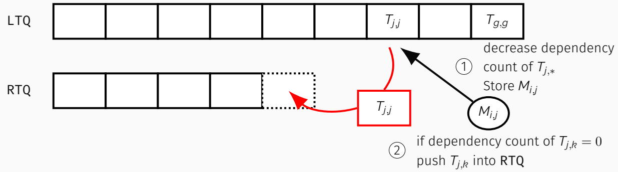


TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue `LTQ`
  - Dependency count

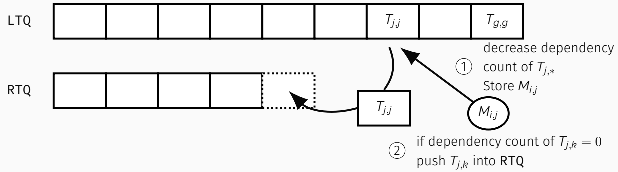


- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
  - Dependency count
  - Ready tasks are placed in RTQ



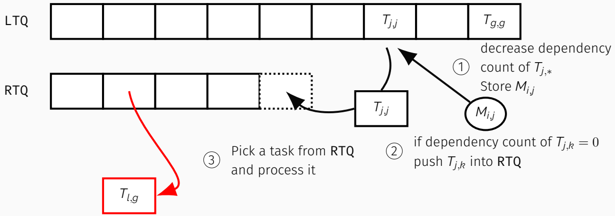
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
  - Dependency count
  - Ready tasks are placed in RTQ



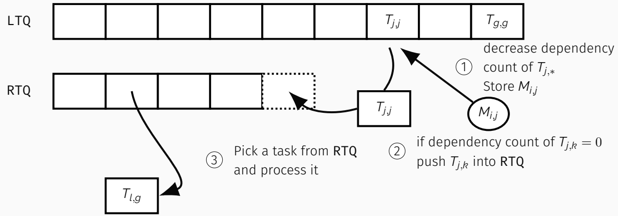
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
  - Dependency count
  - Ready tasks are placed in RTQ



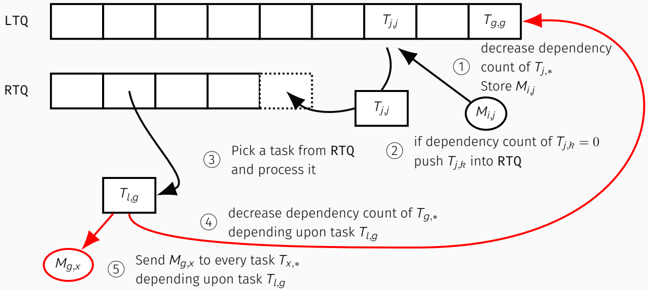
TASK SCHEDULING IN sympack

- Tasks $T_{src \rightarrow tgt}$
- Tasks currently mapped statically
- Processor manages local task queue `LTQ`
  - Dependency count
  - Ready tasks are placed in `RTQ`



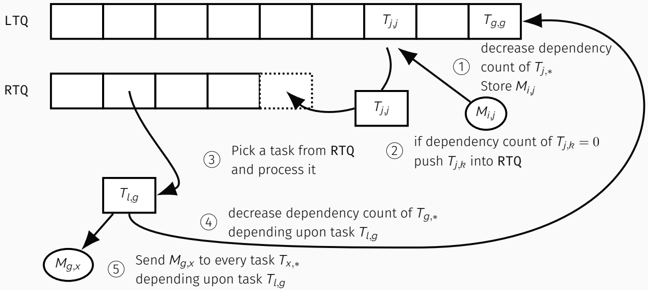
TASK SCHEDULING IN sympack

- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
    - Dependency count
    - Ready tasks are placed in RTQ

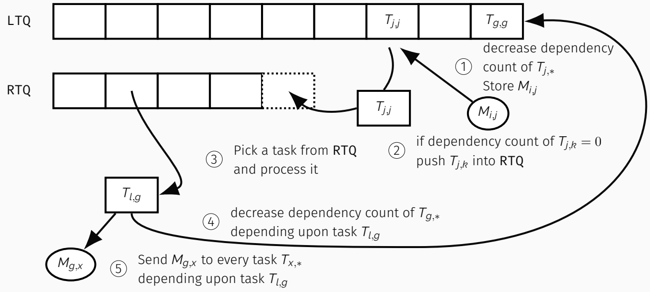


TASK SCHEDULING IN sympack

- Tasks $T_{src \rightarrow tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
    - Dependency count
    - Ready tasks are placed in RTQ

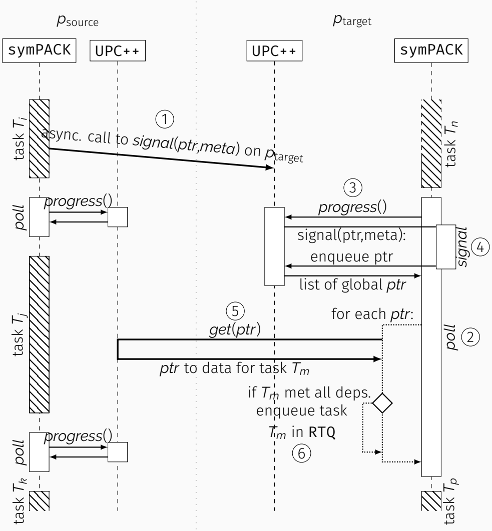


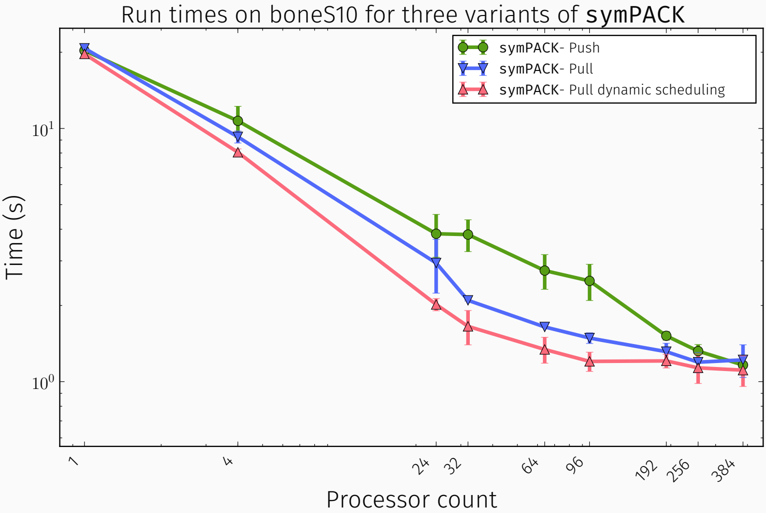
- Tasks $T_{src \to tgt}$
- Tasks currently mapped statically
- Processor manages local task queue LTQ
  - Dependency count
  - Ready tasks are placed in RTQ



Scheduling policy ? FIFO, close to diagonal, etc.

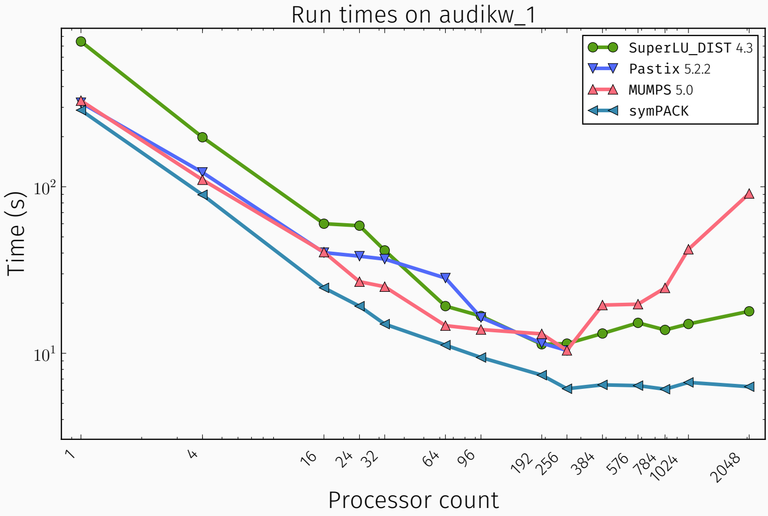
- UPC++ and GASNet for communications
- global pointer to remote memory
- one-sided communications
- asynchronous remote functions calls





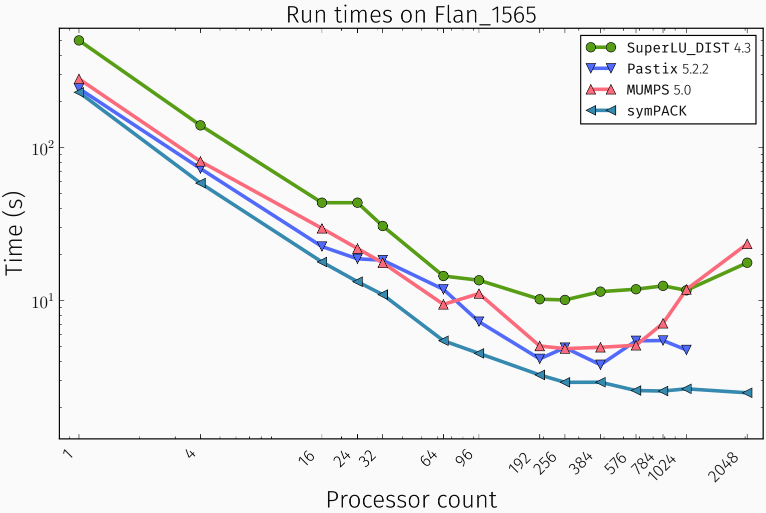
n=914,898 nnz(A)=20,896,803 nnz(L)=318,019,434

STRONG SCALING VS. STATE-OF-THE-ART



n=943,695 nnz(A)=39,297,771 nnz(L)=1,221,674,796

STRONG SCALING VS. STATE-OF-THE-ART



n=1,564,794   nnz(A)=57,865,083   nnz(L)=1,574,541,576

	Speedup vs. sym.			Speedup vs. best		
Problem	min	max	**avg.**	min	max	**avg.**
G3_circuit	0.24	5.70	**1.07**	0.24	5.70	**1.07**
Flan_1565	1.06	9.40	**2.11**	1.06	7.07	**1.94**
af_shell7	0.89	10.61	**3.61**	0.89	7.77	**3.21**
audikw_1	1.11	14.46	**3.14**	1.11	2.84	**1.77**
boneS10	–	–	–	0.86	4.73	**1.75**
bone010	1.06	16.83	**3.34**	1.06	2.03	**1.47**

CONCLUSIONS

- Reduces communication cost in theory [Ashcraft'95]
- Increases parallelism during updates

CONCLUSIONS

- Reduces communication cost in theory [Ashcraft'95]
- Increases parallelism during updates
- Avoiding deadlocks is challenging (Similar to observation by Larkar et al.)
- New symmetric solver `symPACK`
  - implements `Fan-Both`
  - Task based Cholesky requires fine / dynamic scheduling
  - One sided approach using UPC++
  - Asynchronous task execution model
  - dynamic scheduling

ONGOING AND FUTURE WORK

- 2D wrap mapping performance
- Conflict with load balancing (proportional mapping) ?

- Tree-based group communications
- Hybrid parallelism (OpenMP)

- Data distribution (2D, block based ?)
- Scheduling strategies
- New task mapping policies

- Parallel ordering becomes a bottleneck

ONGOING AND FUTURE WORK

- 2D wrap mapping performance
- Conflict with load balancing (proportional mapping) ?

- Tree-based group communications
- Hybrid parallelism (OpenMP)

- Data distribution (2D, block based ?)
- Scheduling strategies
- New task mapping policies

- Parallel ordering becomes a bottleneck

Async. model important for scalability and to tolerate variability

ONGOING AND FUTURE WORK

- 2D wrap mapping performance
- Conflict with load balancing (proportional mapping) ?

- Tree-based group communications
- Hybrid parallelism (OpenMP)

- Data distribution (2D, block based ?)
- Scheduling strategies
- New task mapping policies

- Parallel ordering becomes a bottleneck

Async. model important for scalability and to tolerate variability

`www.sympack.org`