SYMPACK: A 2D TASK-BASED FACTORIZATION ALGORITHM FOR SPARSE SYMMETRIC MATRICES

Mathias Jacquelin
mjacquelin@lbl.gov

Esmond Ng

February 27 2019

Scalable Solvers Group
Computational Research Department
Lawrence Berkeley National Laboratory

Motivations:

- Sparse matrices arise in many applications:
  - Optimization problems
  - Discretized PDEs
  - Electronic structure theory
  - ...
- Some sparse direct methods require:
  - Sparse factorizations
  - Computing some inverse elements

DEDICATED SOLVERS FOR SPARSE SYMMETRIC MATRICES

- Matrix A is symmetric in many cases
- Symmetric storage: only lower triangular part of A is stored
  - Lower memory consumption
  - Fewer floating point operations
- Many ways to schedule computations, partition data
- Challenging problem: irregular computation load
- **Crucial to remove synchronization points**

- 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**
  $\ell_{j,j} = \sqrt{A_{j,j}}$
  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
end

CHOLESKY FACTORIZATION

- 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})                     ⎫
    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                 ⎫ 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
```

- 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
```

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**

end

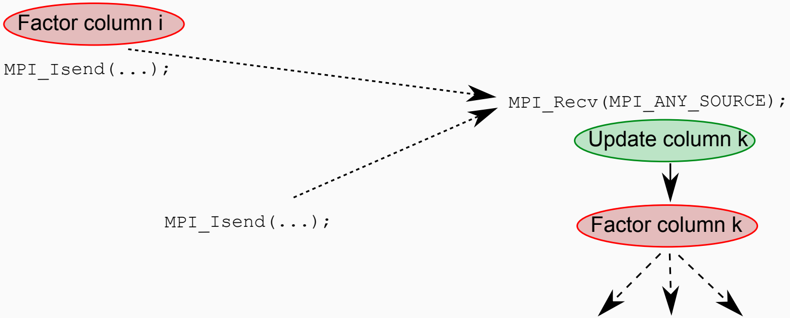
Update next columns and Aggregate updates

for *row* $i = k$ *to* n **do**

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

end

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


Factor column i
MPI_Isend(...);
MPI_Recv(MPI_ANY_SOURCE);
Update column k
MPI_Isend(...);
Factor column k

PITFALL: DEADLOCKS CAN HAPPEN

Asynchronous comm. becomes blocking when out of buffer

PITFALL: DEADLOCKS CAN HAPPEN

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

PITFALL: DEADLOCKS CAN HAPPEN

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

- Deadlock prevention is difficult:
  - Order in operations/messages

PITFALL: DEADLOCKS CAN HAPPEN

Asynchronous comm. becomes blocking when out of buffer

Deadlock issues

- Deadlock prevention is difficult:
  - Order in operations/messages

Potential over-synchronization

PITFALL: DEADLOCKS CAN HAPPEN

Asynchronous comm. becomes blocking when out of buffer

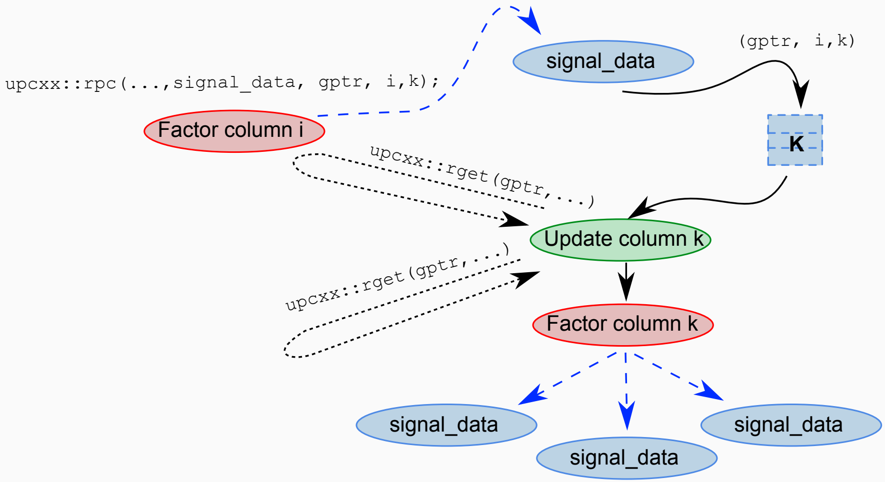
Deadlock issues

- Deadlock prevention is difficult:
  - Order in operations/messages

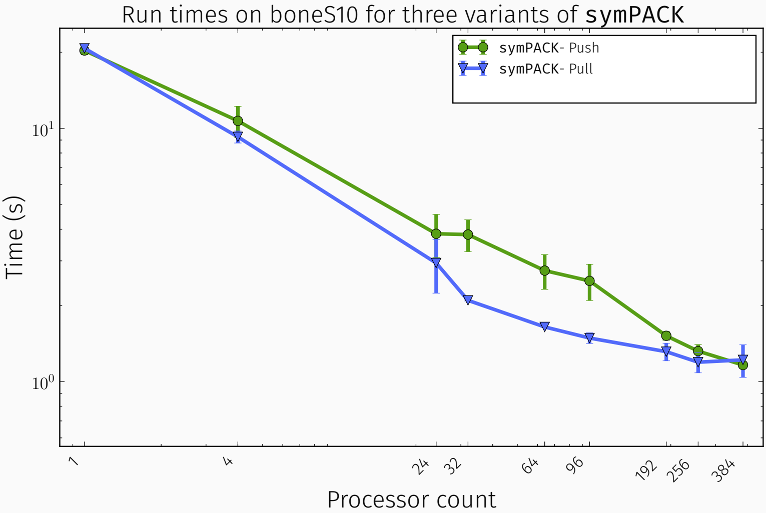
    Potential over-synchronization

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

UPC++ ONE-SIDED PULL STRATEGY

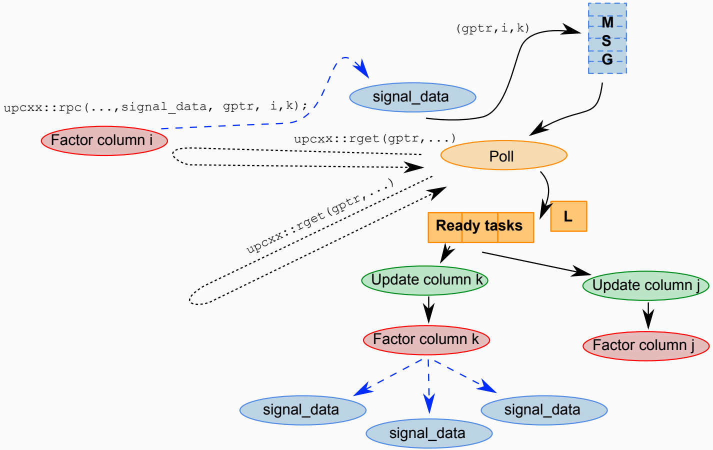


WAS THE SCHEDULE CONSTRAINED ?

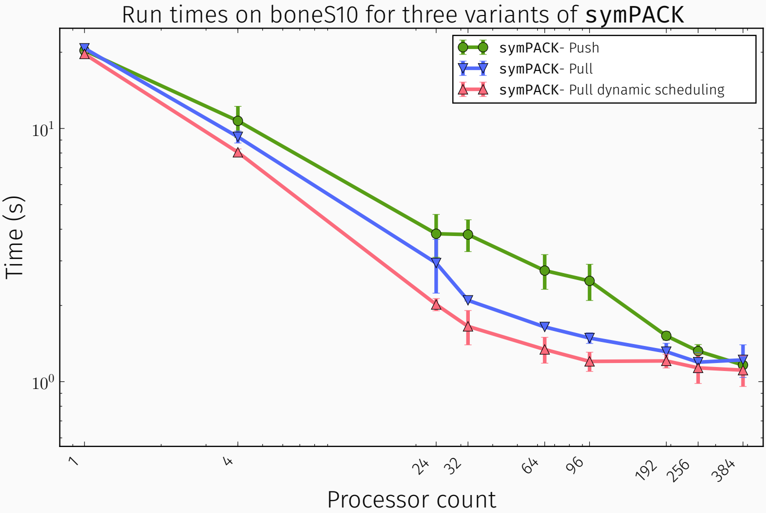


n=914,898 nnz(A)=20,896,803 nnz(L)=318,019,434

- Per-task dependency counts
- Update dependencies as messages are flowing in
- Maintain a list of tasks ready for execution



IMPACT OF COMMUNICATION STRATEGY AND SCHEDULING



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

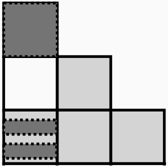
2D DATA DISTRIBUTION

- 2D block cyclic used in many solvers
- Works well in practice for sparse matrices as well
- However, nothing is explicitly balanced

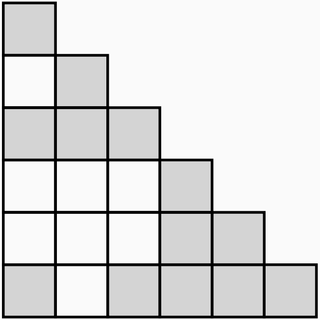
- 2D block cyclic used in many solvers
- Works well in practice for sparse matrices as well
- However, nothing is explicitly balanced

- Can we do better?

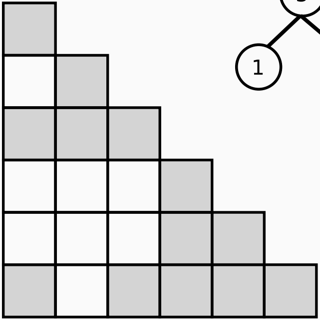
- 2D block cyclic used in many solvers
- Works well in practice for sparse matrices as well
- However, nothing is explicitly balanced
- Can we do better?
- Can we store mapping information?
  - Cell: block “delimited” by supernode partition
  - Block: set of contiguous rows in a given supernode
  - A cell can hold multiple blocks

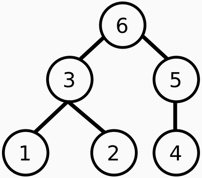


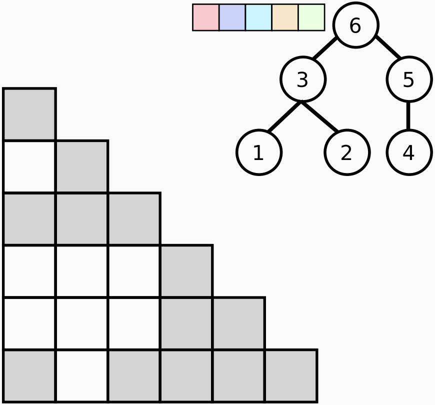
A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH



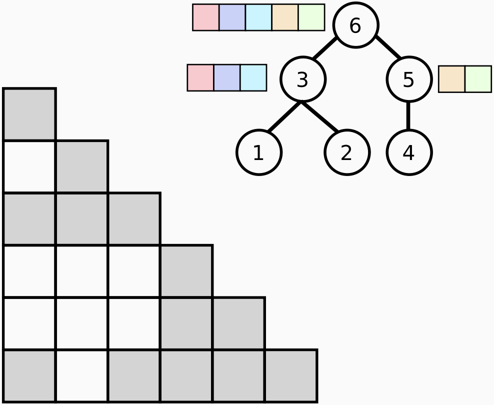
A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH



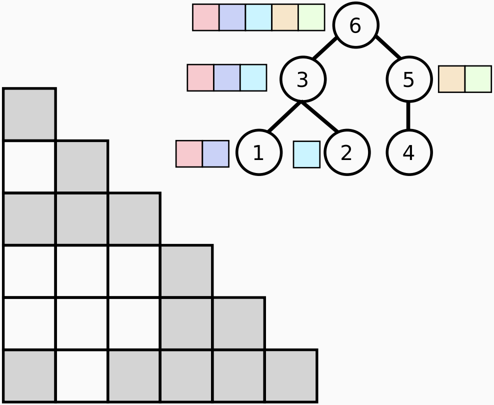




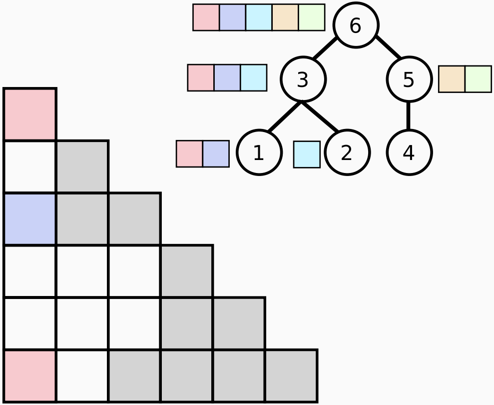
- Subtree-to-subcube mapping



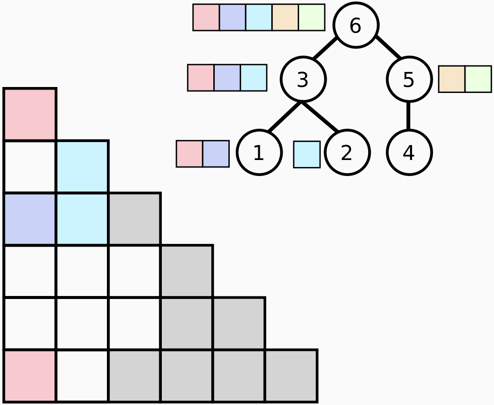
- Subtree-to-subcube mapping



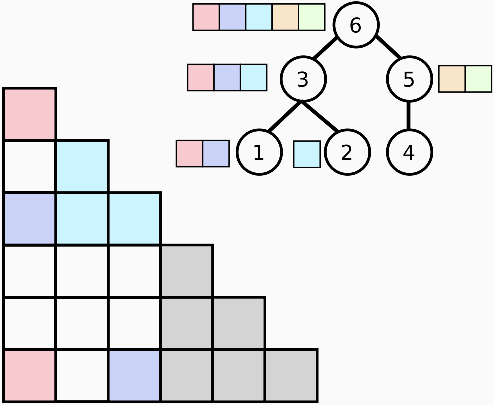
- Subtree-to-subcube mapping



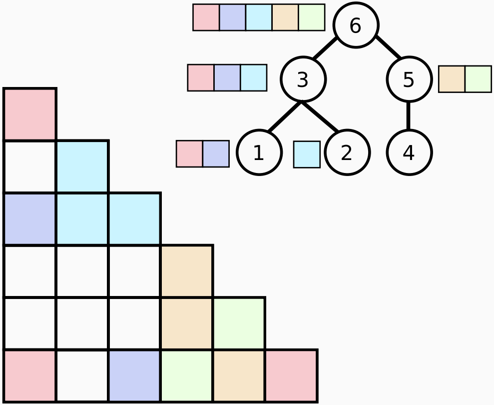
- Subtree-to-subcube mapping
- Non-empty cells distributed within groups



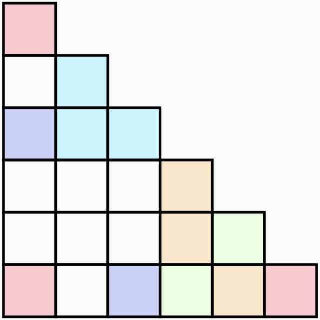
- Subtree-to-subcube mapping
- Non-empty cells distributed within groups



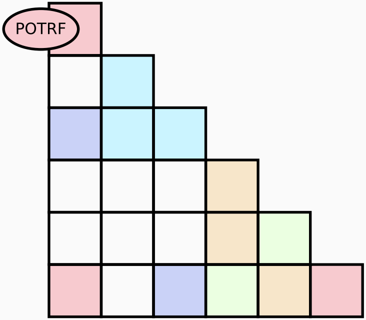
- Subtree-to-subcube mapping
- Non-empty cells distributed within groups



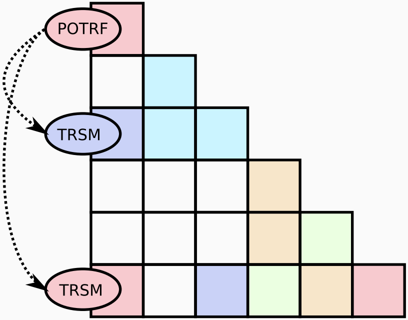
- Subtree-to-subcube mapping
- Non-empty cells distributed within groups

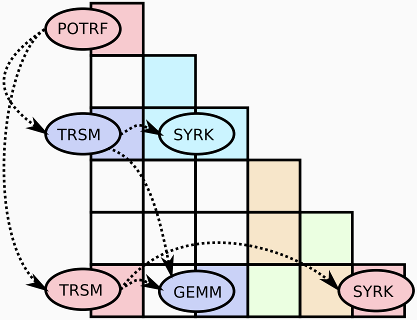


A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH

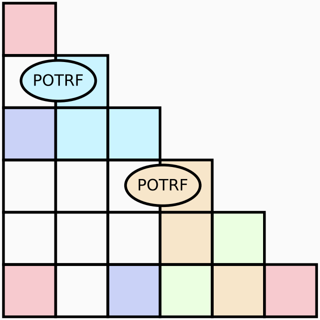


A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH

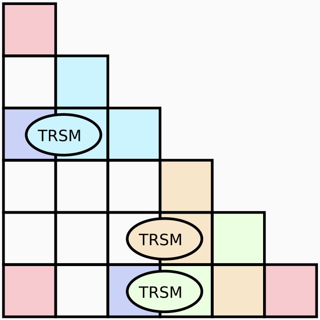



POTRF
TRSM
SYRK
TRSM
GEMM
SYRK

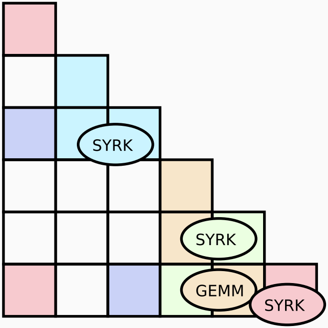
A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH



A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH

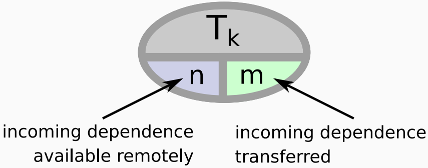


A NEW 2D DATA DISTRIBUTION, A NEW TASK GRAPH



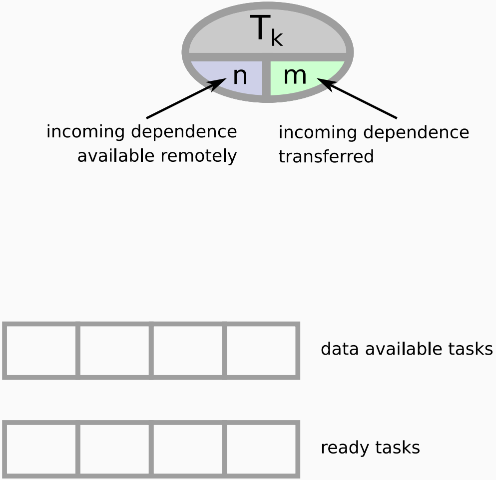
- A **Future** is a synchronization object for asynchronous operations:
  - When the operation is complete, future becomes *ready*
  - A *callback* can be attached to a future
- A **Promise** can be thought as a counter:
  - Associated with a future
  - Future is ready when the count reaches 0.

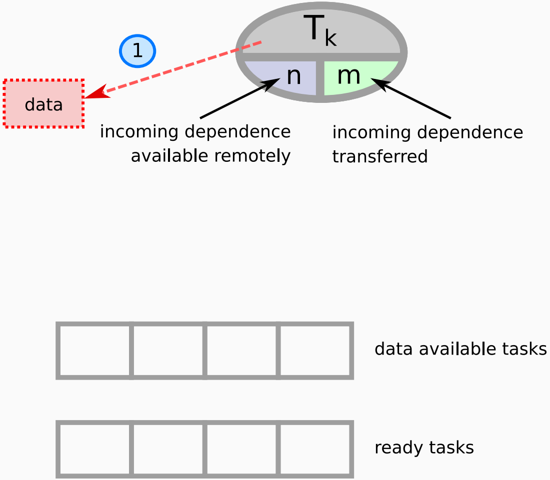
2D TASK GRAPH SCHEDULING: GETTING RID OF THE POLLING FUNCTION

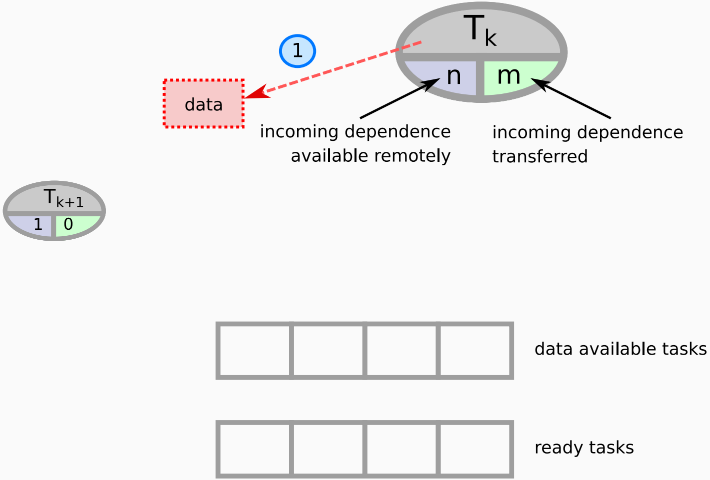


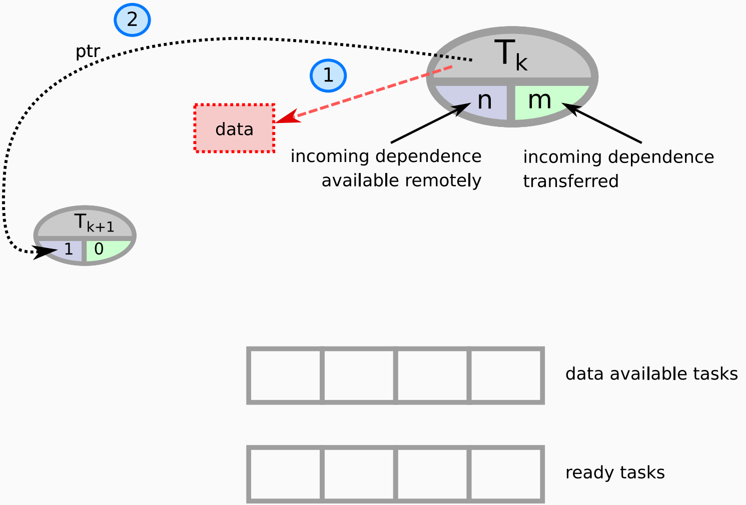
- A **Future** is a synchronization object for asynchronous operations:
  - When the operation is complete, future becomes *ready*
  - A *callback* can be attached to a future
- A **Promise** can be thought as a counter:
  - Associated with a future
  - Future is ready when the count reaches 0.
- Each task has two **Promise**s (a counter)

2D TASK GRAPH SCHEDULING: GETTING RID OF THE POLLING FUNCTION

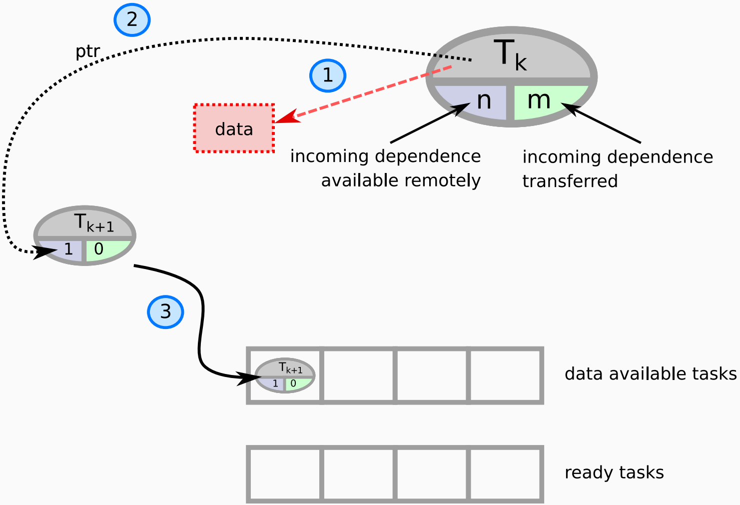


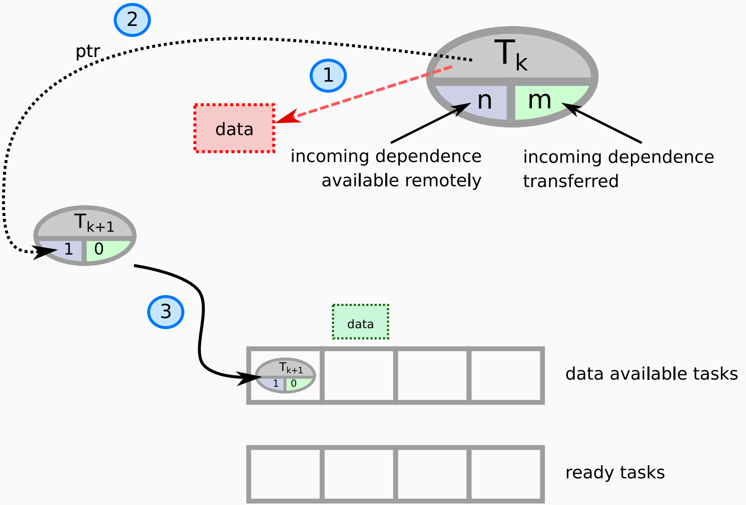

1
data
T_k
n
m
incoming dependence available remotely
incoming dependence transferred
data available tasks
ready tasks


1
data
T_k
n
m
incoming dependence
available remotely
incoming dependence
transferred
T_{k+1}
1
0
data available tasks
ready tasks

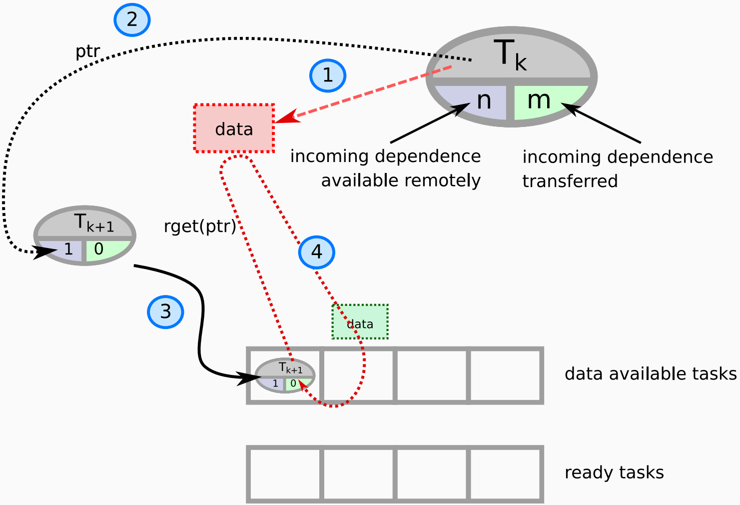

2
ptr
T_k
1
n
m
data
incoming dependence available remotely
incoming dependence transferred
T_{k+1}
1
0
data available tasks
ready tasks

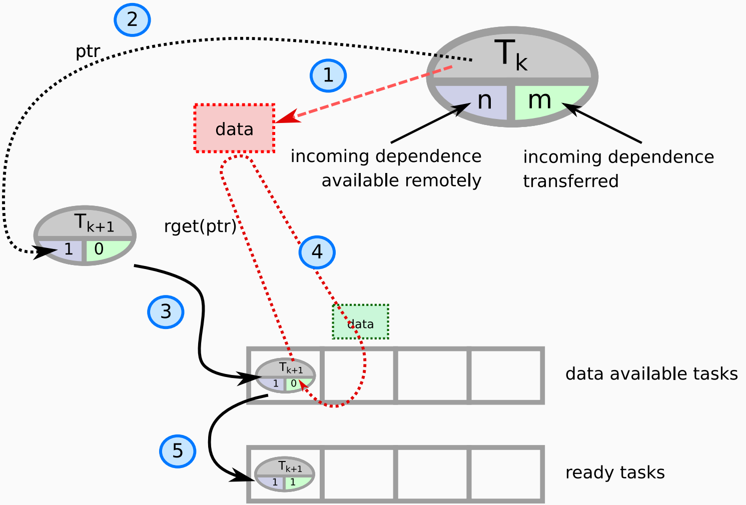
2D TASK GRAPH SCHEDULING: GETTING RID OF THE POLLING FUNCTION



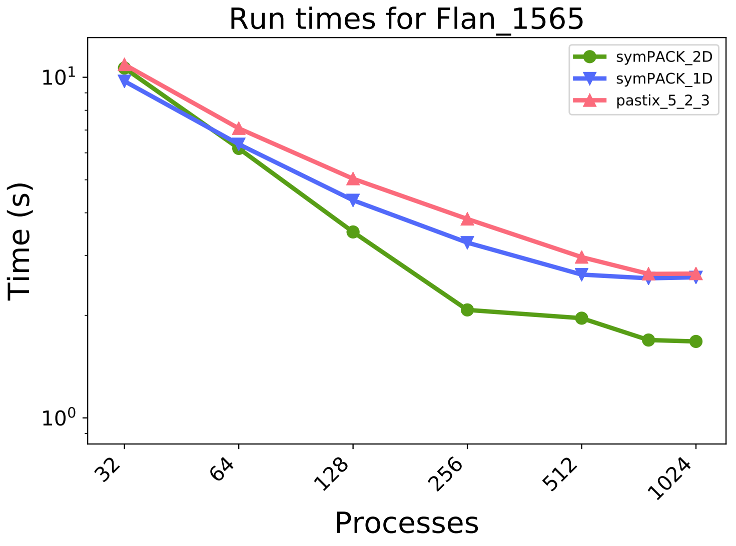

2
ptr
1
Tk
n
m
data
incoming dependence available remotely
incoming dependence transferred
Tk+1
1
0
3
data
Tk+1
1
0
data available tasks
ready tasks

2D TASK GRAPH SCHEDULING: GETTING RID OF THE POLLING FUNCTION

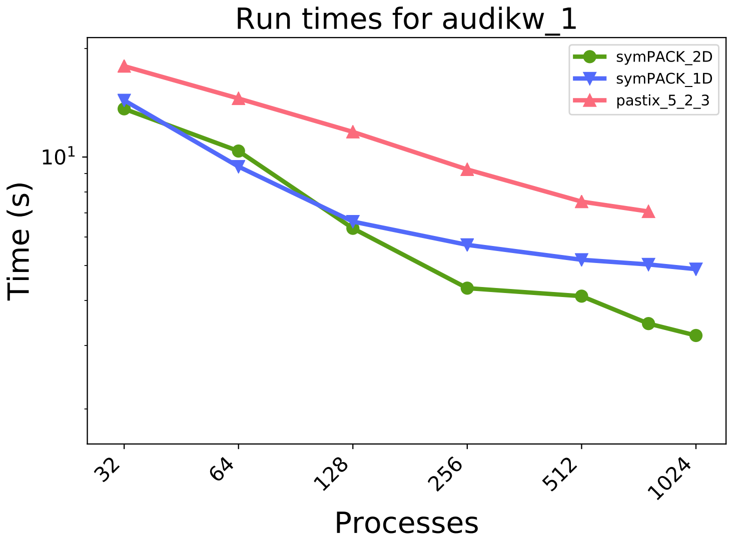



2
ptr
1
T_k
n
m
data
incoming dependence
available remotely
incoming dependence
transferred
T_{k+1}
1
0
rget(ptr)
4
3
data
data available tasks
5
ready tasks

STRONG SCALING VS. STATE-OF-THE-ART (NERSC CORI HASWELL)

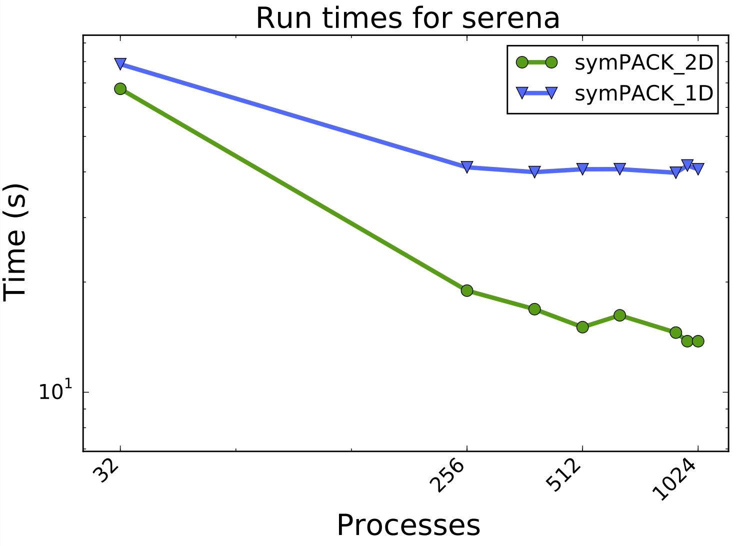


n=1,564,794 nnz(L)=1,574,541,576



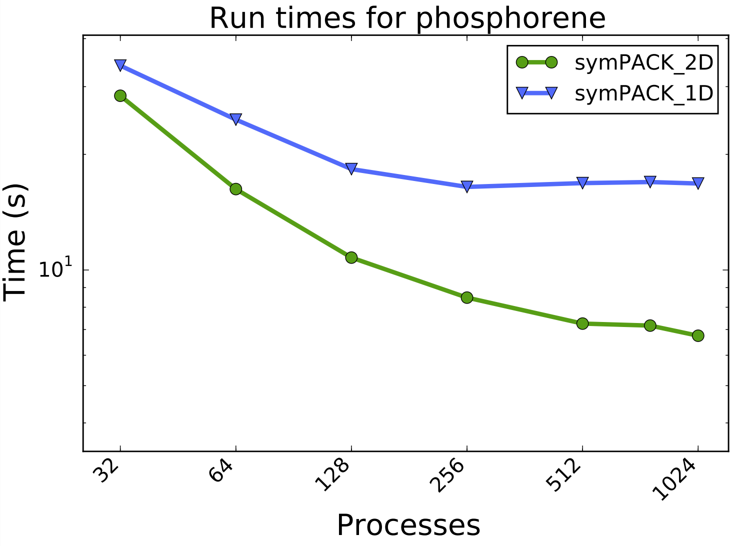
n=943,695 nnz(L)=1,261,342,196

STRONG SCALING VS. STATE-OF-THE-ART (NERSC CORI HASWELL)



n=1,391,349 nnz(L)=2,821,178,652

STRONG SCALING VS. STATE-OF-THE-ART (NERSC CORI HASWELL)



n=512,000 nnz(L)=1,697,433,600

THE FUTURE<T>: UPDATES TO COME

- Aggregate updates using a tree pattern
- 1D data distribution at leaves
- Use tasks to implement 3D type of layout at higher levels (multiple tasks on the same cell)
- Accelerator / GPU support
  - Upcoming UPC++ with seamless local/remote host/device memory accesses
  - Batched BLAS
- Acknowledgments:
  - DOE SciDAC FASTMath, CompCat, ComPASS4
  - ECP Pagoda