Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System

Marc Sergent, David Goudin, Samuel Thibault, Olivier Aumage

Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System

Marc SERGENT, David GOUDIN,
Samuel THIBAULT, Olivier AUMAGE

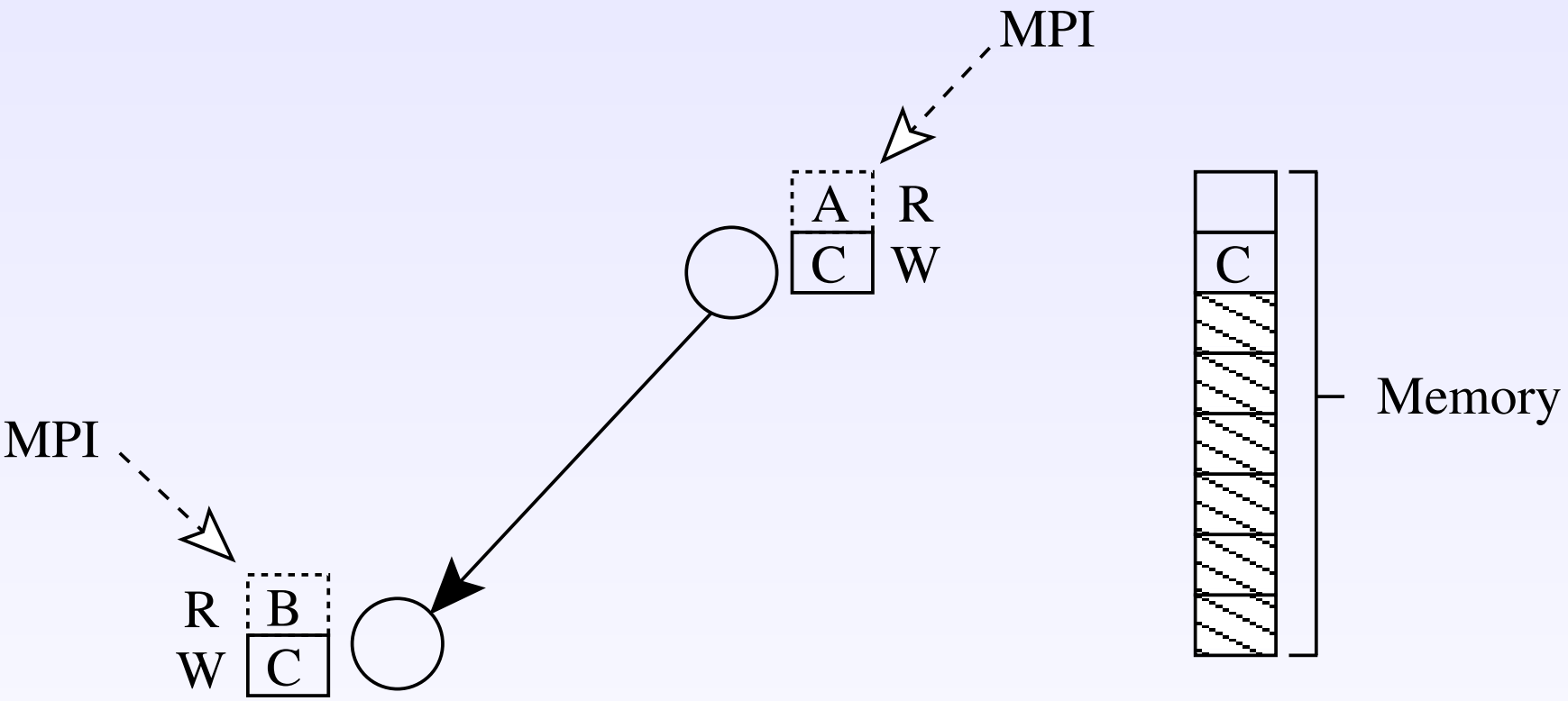
SIAM-PP CONFERENCE,
PARIS, APRIL 12TH, 2016

MORSE

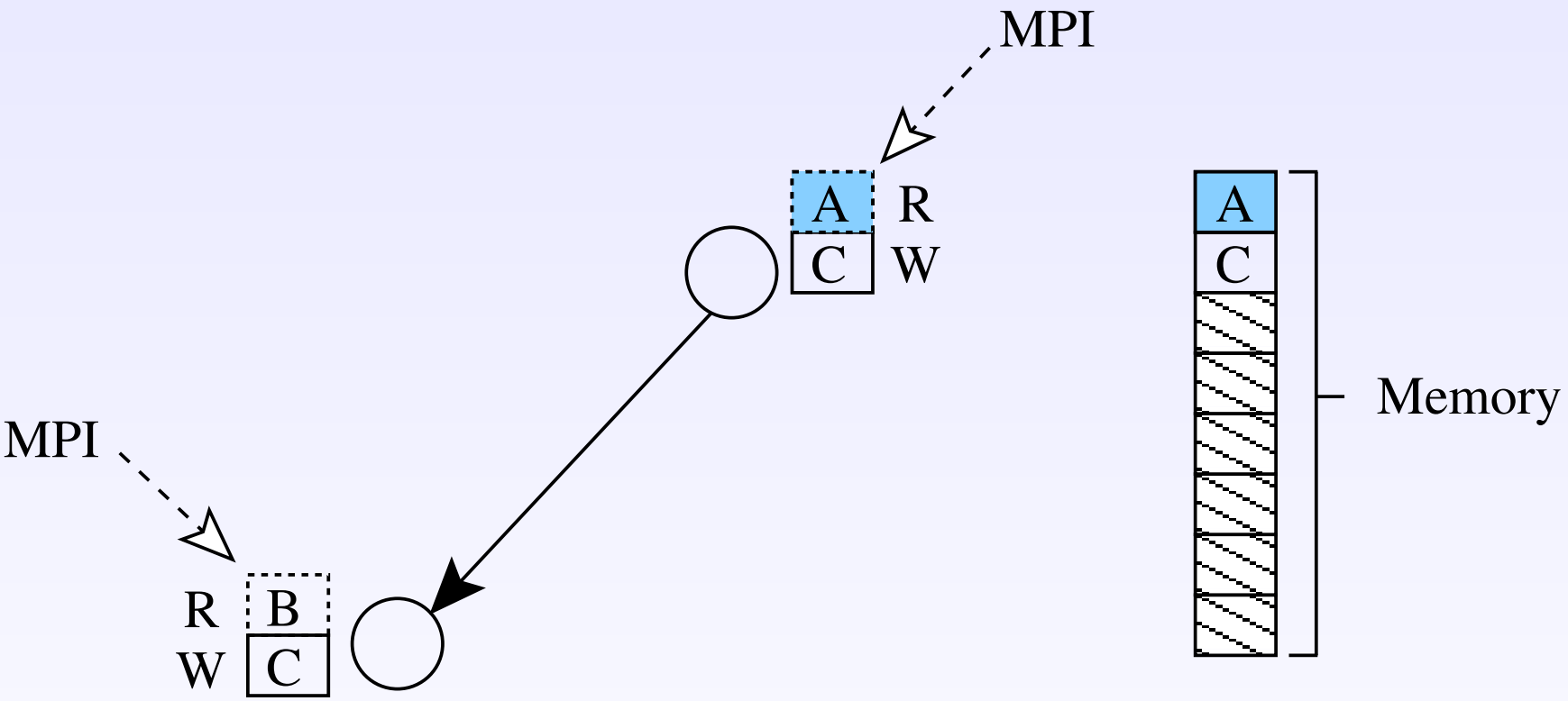
Outline

Outline

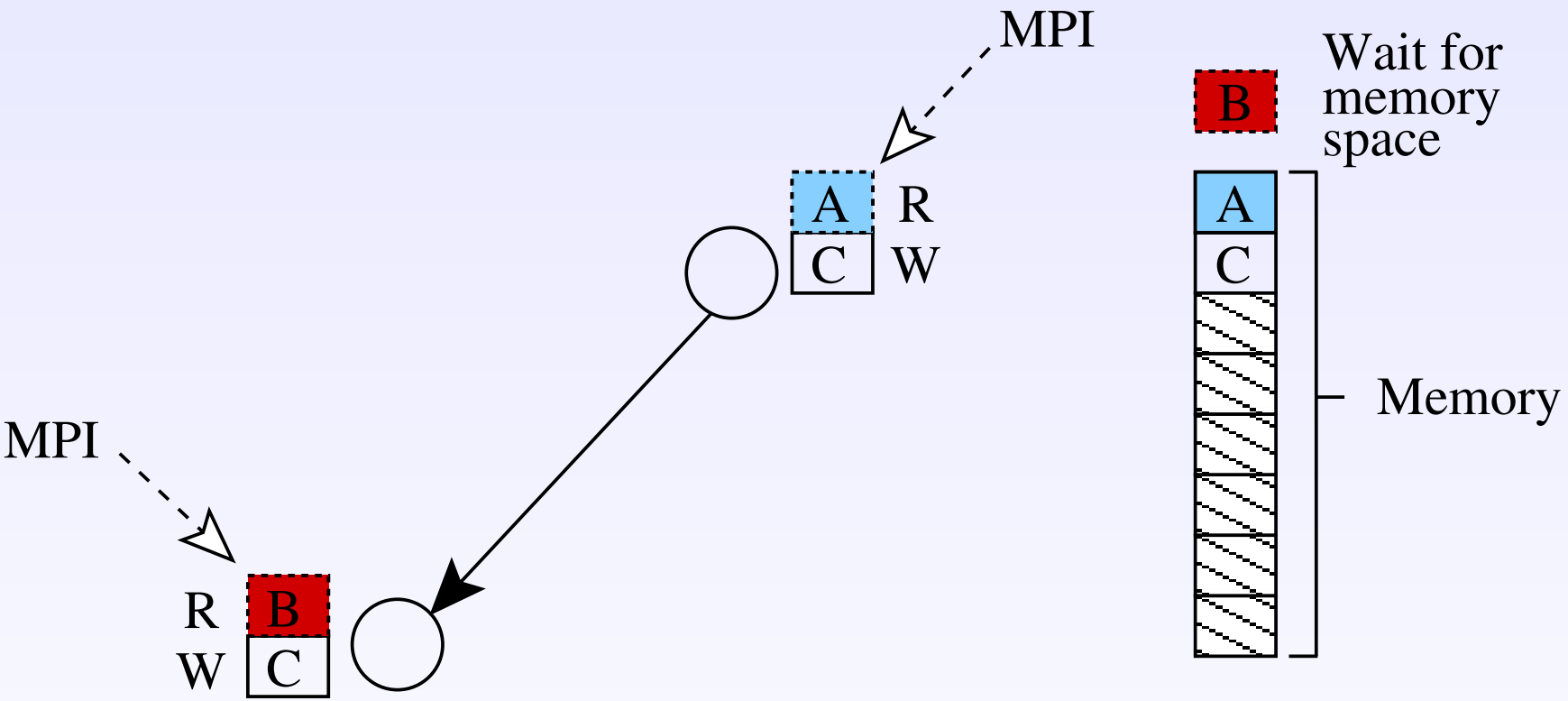
Memory and distributed task-based applications



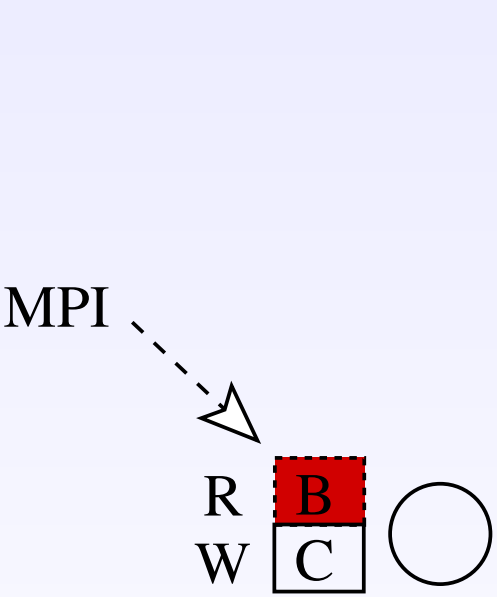
Memory and distributed task-based applications

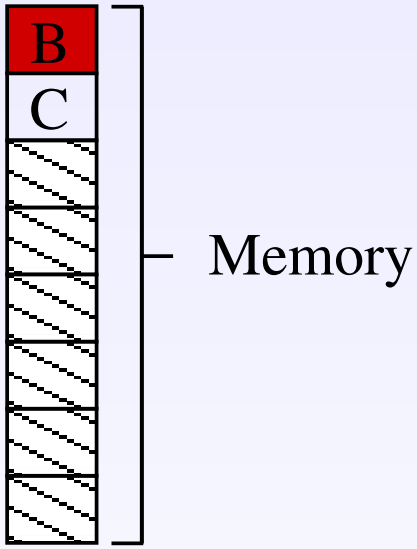


Memory and distributed task-based applications



Memory and distributed task-based applications

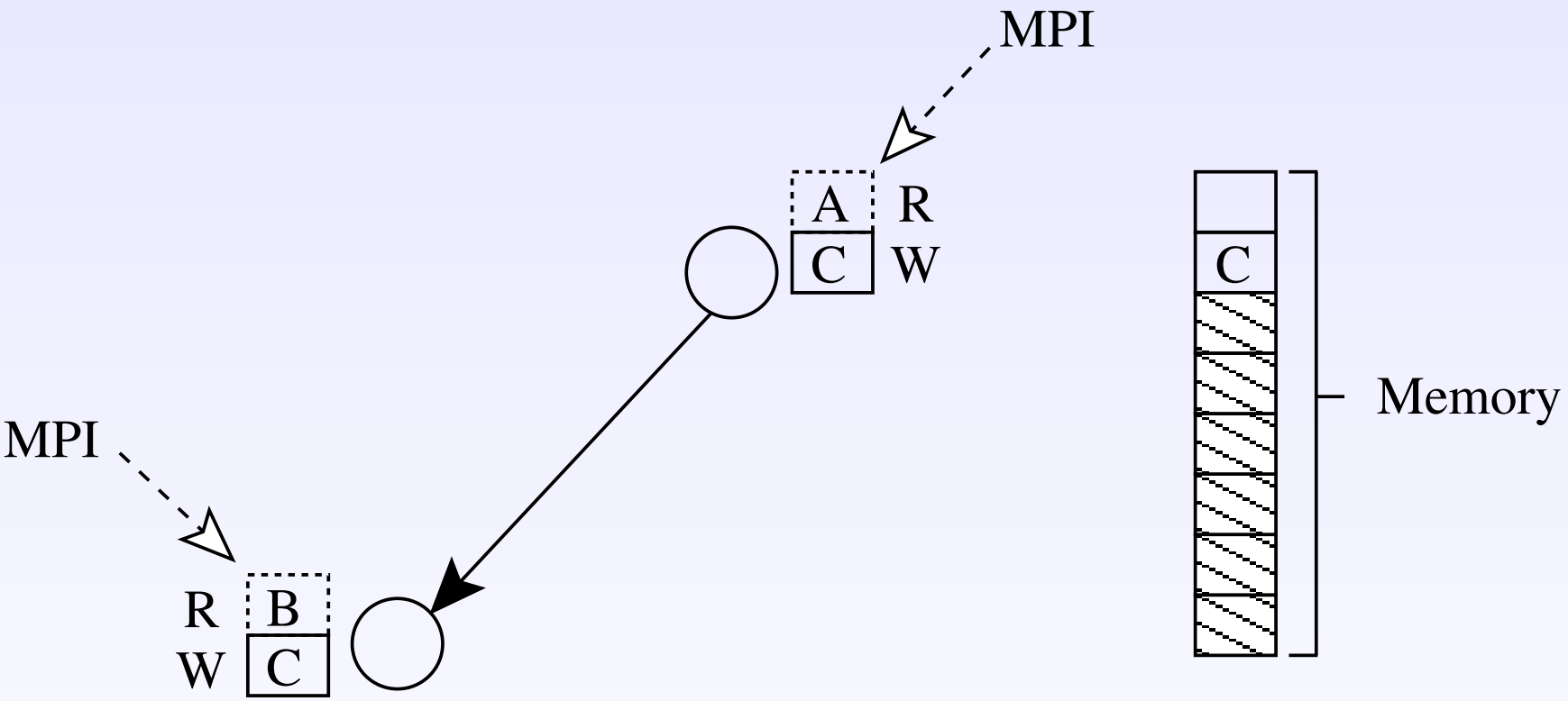




Memory and distributed task-based applications

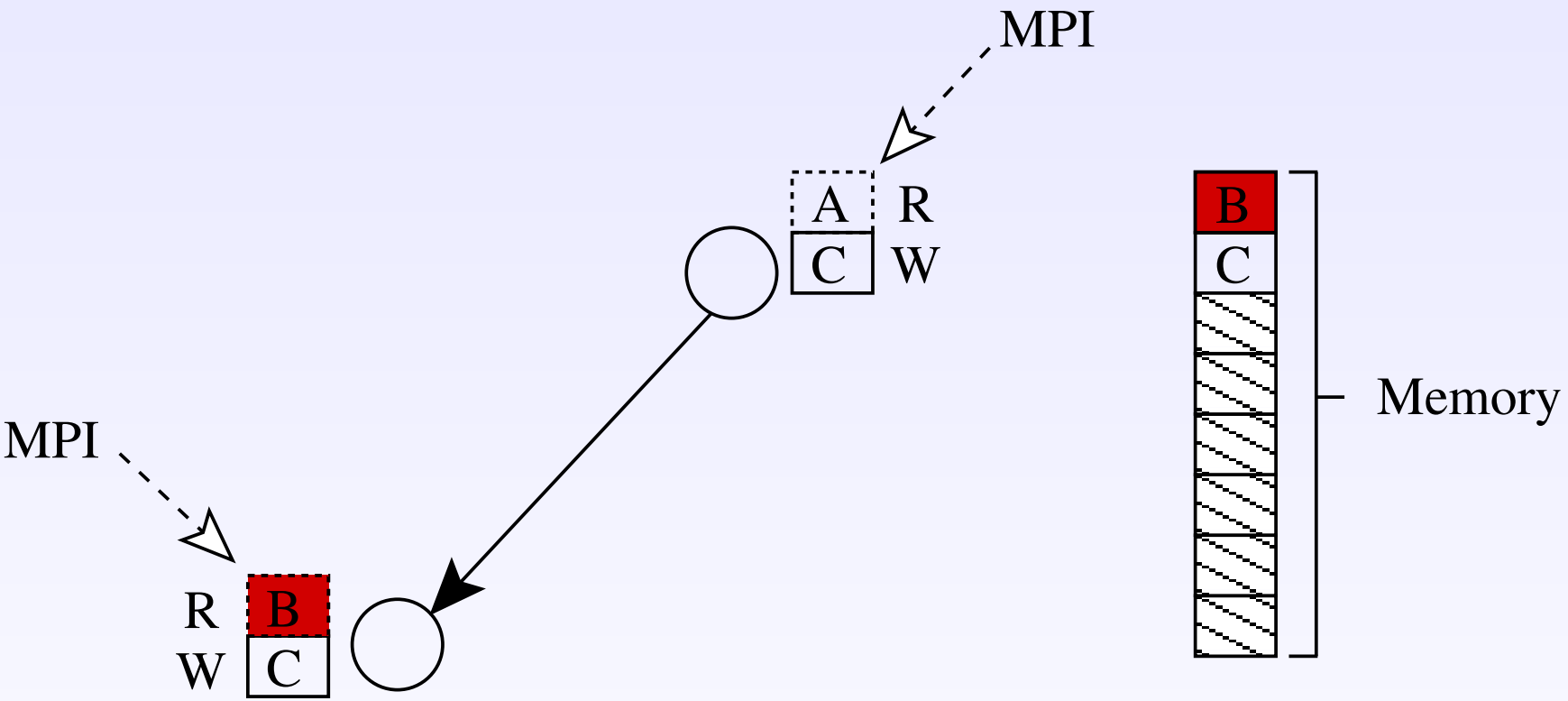
- Data A received before data B : works fine

Memory and distributed task-based applications



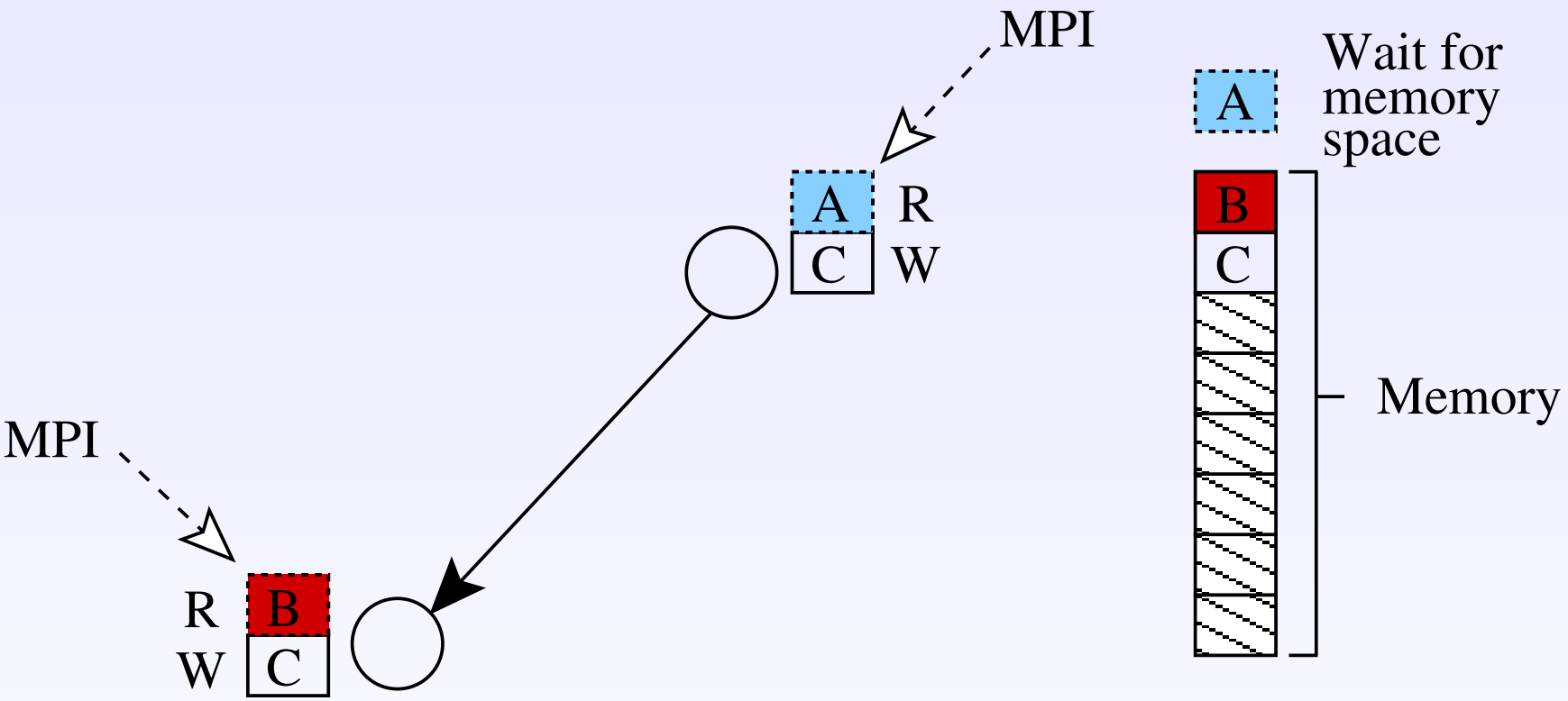
- Data A received before data B : works fine

Memory and distributed task-based applications



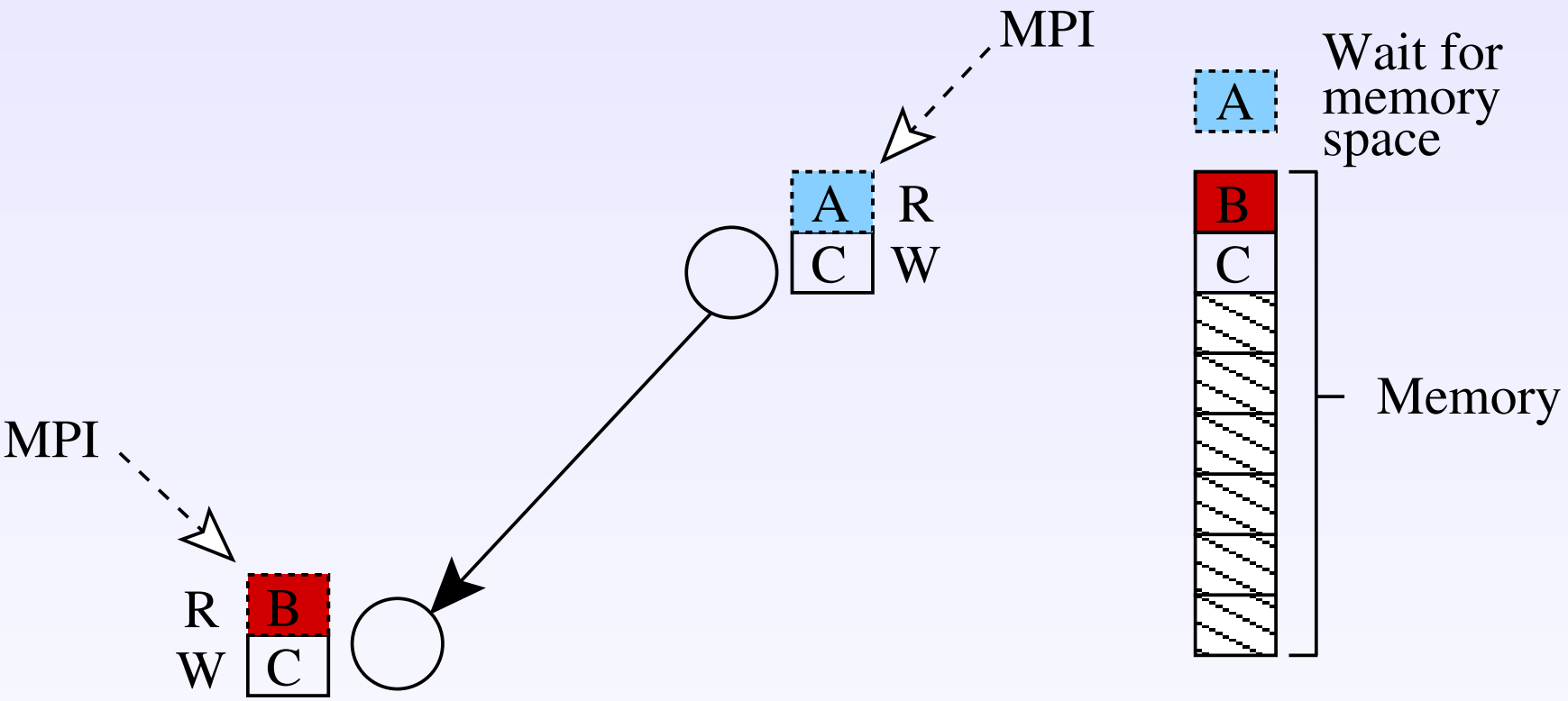
- Data A received before data B : works fine

Memory and distributed task-based applications



- Data A received before data B : works fine

Memory and distributed task-based applications



- Data A received before data B : works fine
- Data B received before data A : deadlock

Memory and distributed task-based applications

Issue

- Bad memory allocation order for remote data can lead to deadlocks
  - Need to constraint the order of memory allocations

Problematic

- To what extent should we sequentialize memory allocations for recieving remote data ?
  - Without hindering the communication/computation overlapping
  - Without introducing other deadlocks

Memory and distributed task-based applications

Proposed solution

- Decouple the task submission and task execution steps
- Introduce a sequential ordering of tasks **at submission time**, not at execution time

$\Rightarrow$ Sequential Task Flow (STF) model

- Implemented by the StarPU runtime system (Inria STORM team)

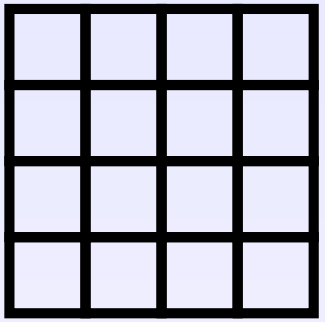
Working principle

- Memory allocations performed at submission time
- When not enough memory available, blocks task submission
- When enough memory have been freed, unblocks task submission

Outline

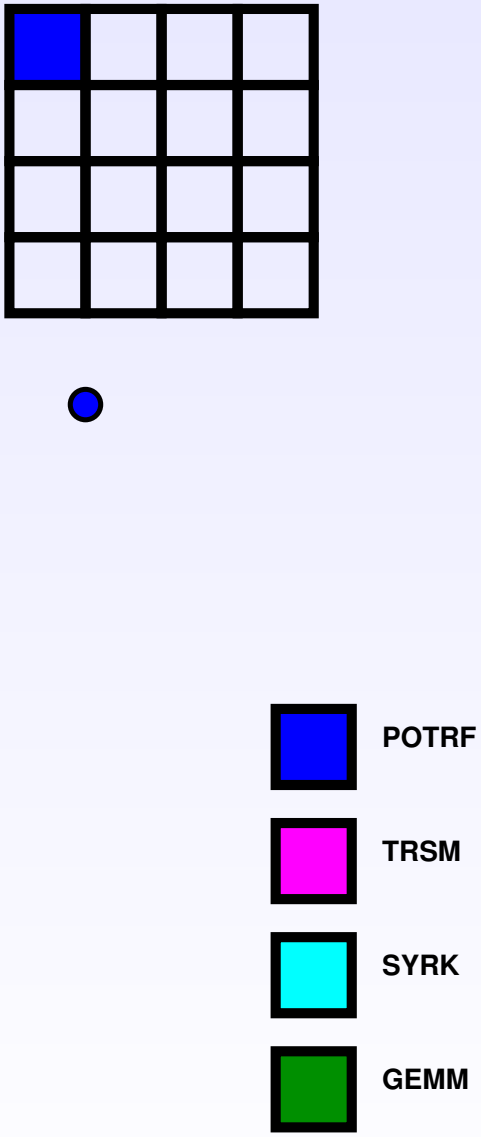
Sequential task-based Cholesky on a single node

```
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();
```



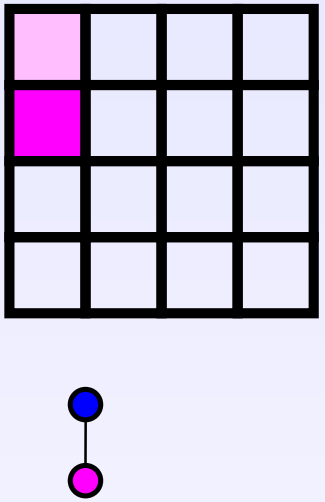
Sequential task-based Cholesky on a single node

```
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();
```



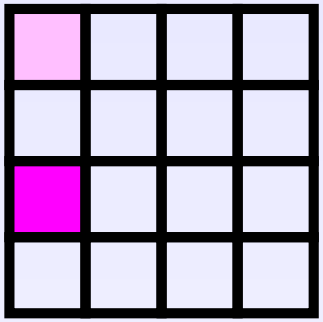
Sequential task-based Cholesky on a single node

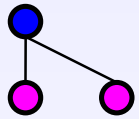
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```



Sequential task-based Cholesky on a single node

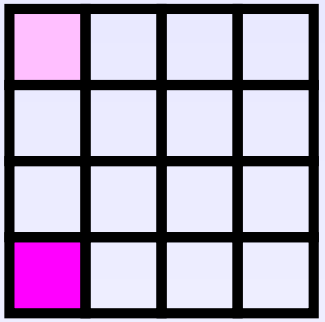
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

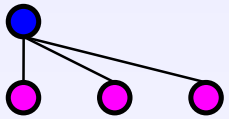




Sequential task-based Cholesky on a single node

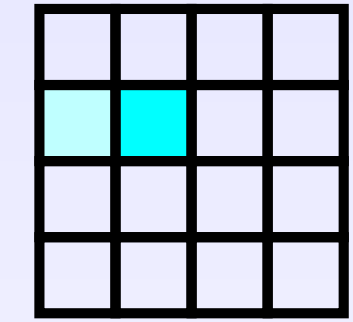
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

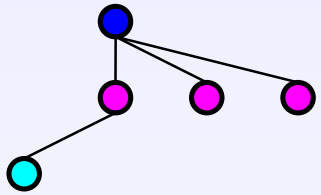




Sequential task-based Cholesky on a single node

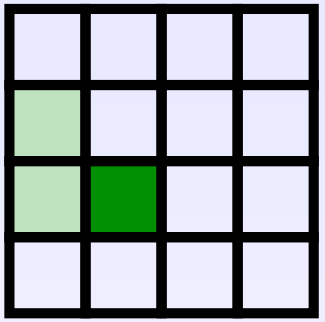
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

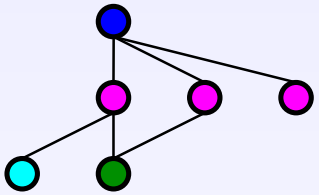




Sequential task-based Cholesky on a single node

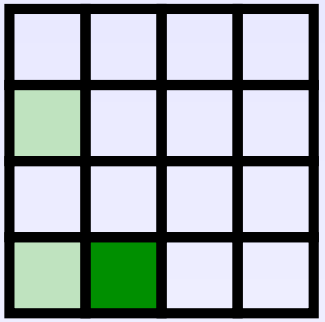
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

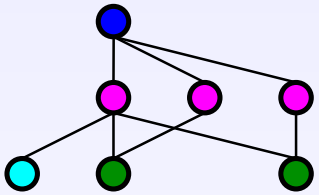




Sequential task-based Cholesky on a single node

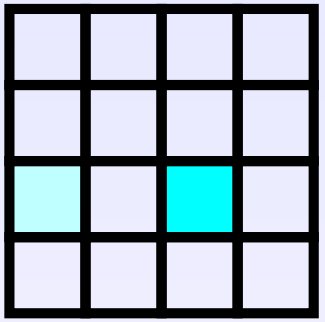
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

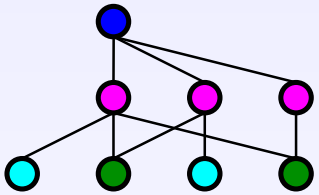




Sequential task-based Cholesky on a single node

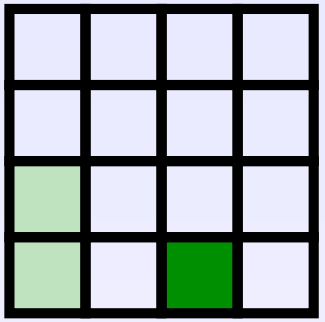
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

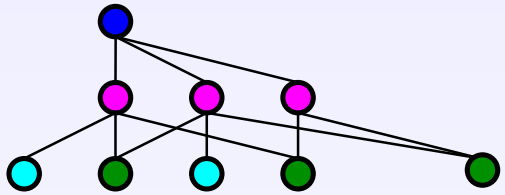




Sequential task-based Cholesky on a single node

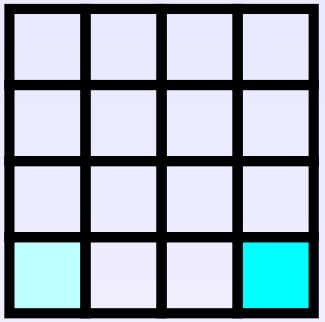
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

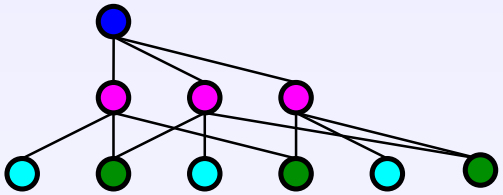




Sequential task-based Cholesky on a single node

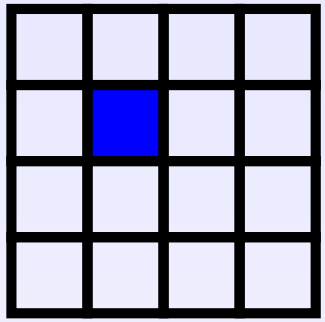
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```

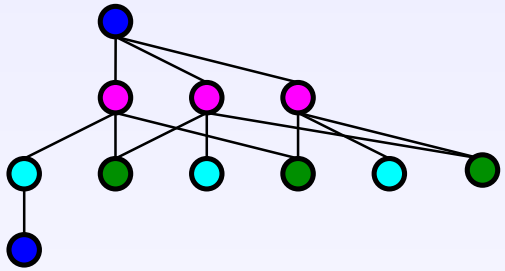




Sequential task-based Cholesky on a single node

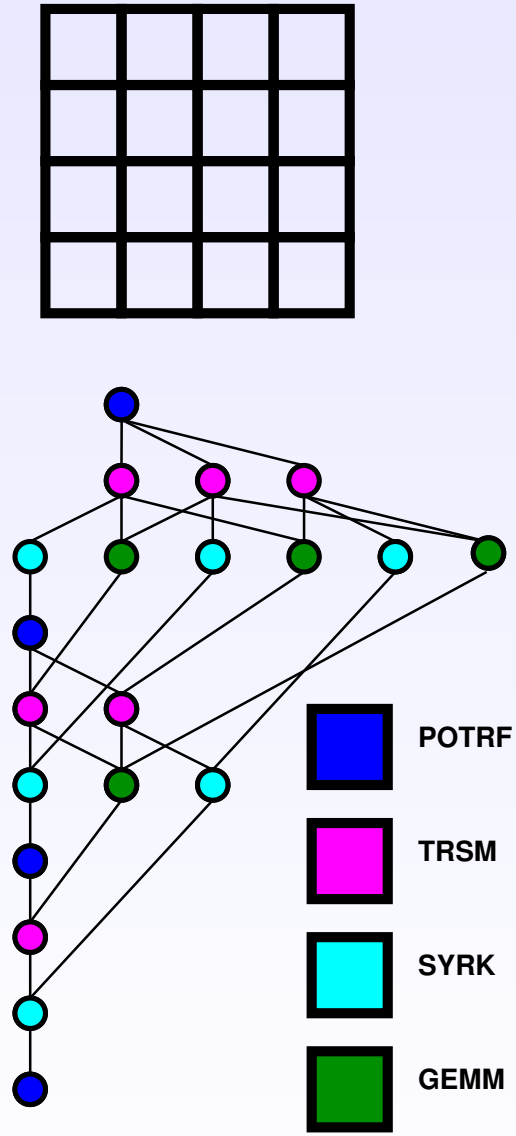
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
    TRSM (RW,A[i][j], R,A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
  }
}
task_wait_for_all();
```





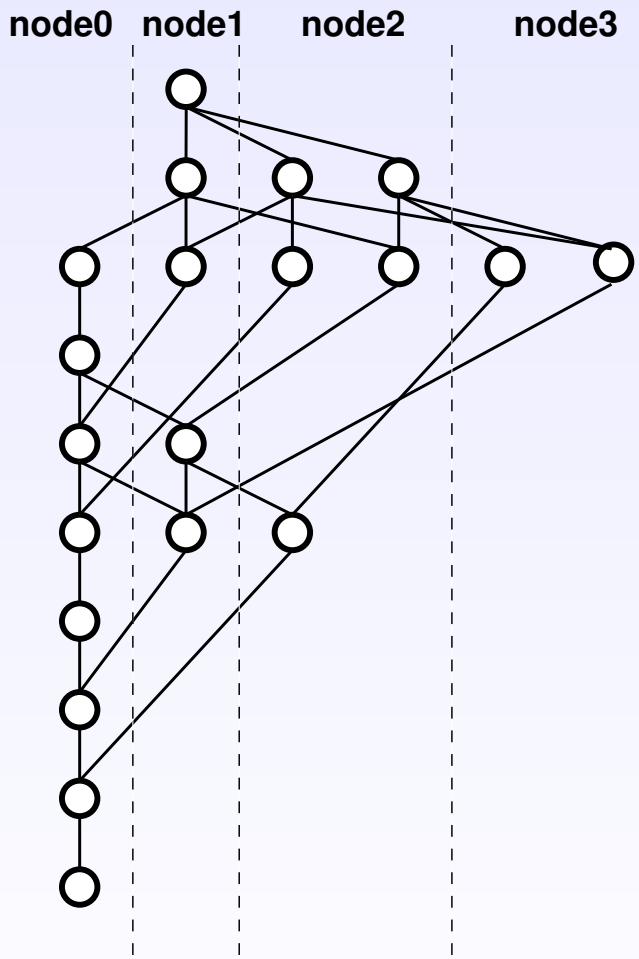
Sequential task-based Cholesky on a single node

```
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();
```



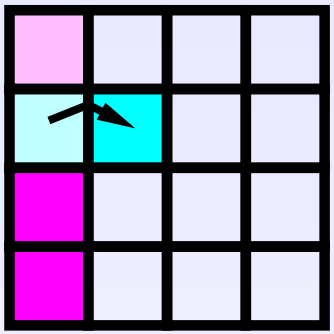
Distributed case : which node executes which tasks?

- Two mappings needed for distributed computing
  - Where is the data ?
    - Data mapping
  - Where are the computations ?
    - Task mapping
- The application must provide the initial data mapping
- Two approaches for task mapping :
  - Automatically inferred
    - From the data mapping
    - By default, where data is written to
  - Specified by the application

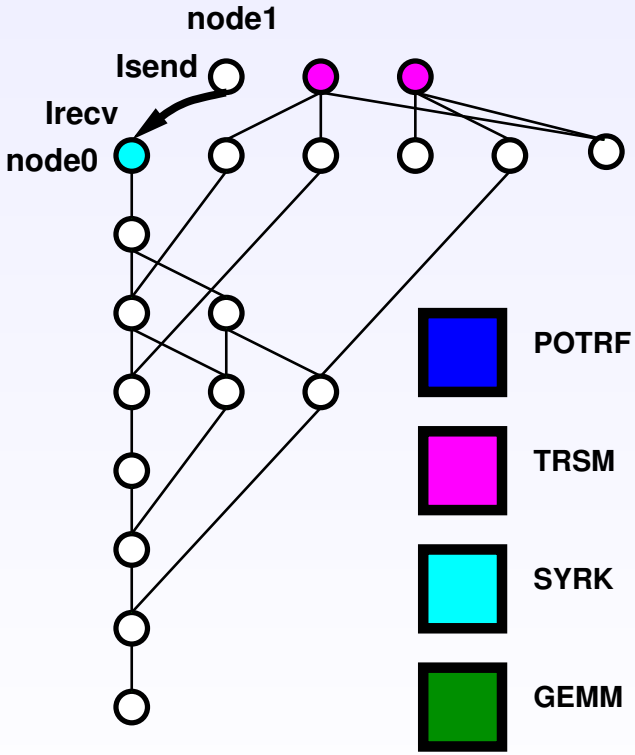


Distributed programming paradigm

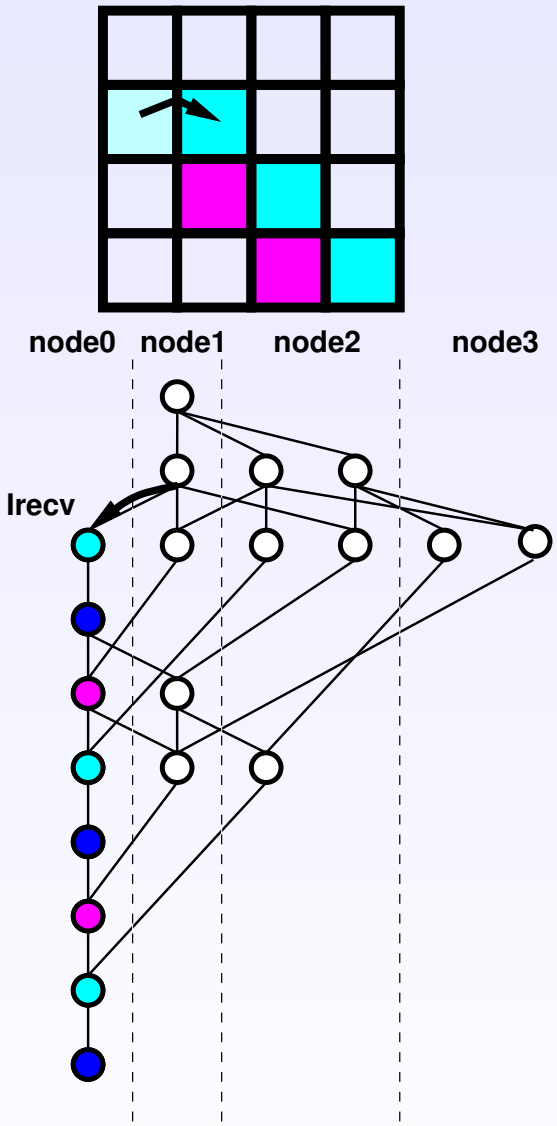
- Data communications inferred from the task graph
  - Edges = data communications
  - If tasks are executed on different processes, infer an MPI communication



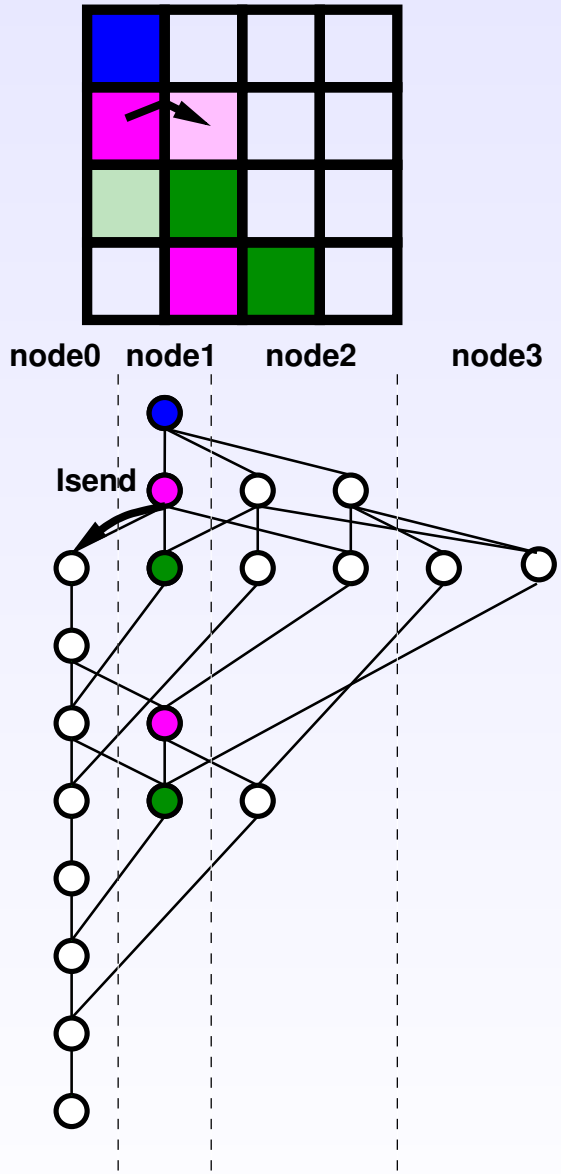
- Automatically inferred by the runtime system at submission time
  - Replicated unrolling of the task graph
    - Same task submission order on all processes
    - Communications inferred in the same order on all processes



Data transfers between nodes



Node 0 execution

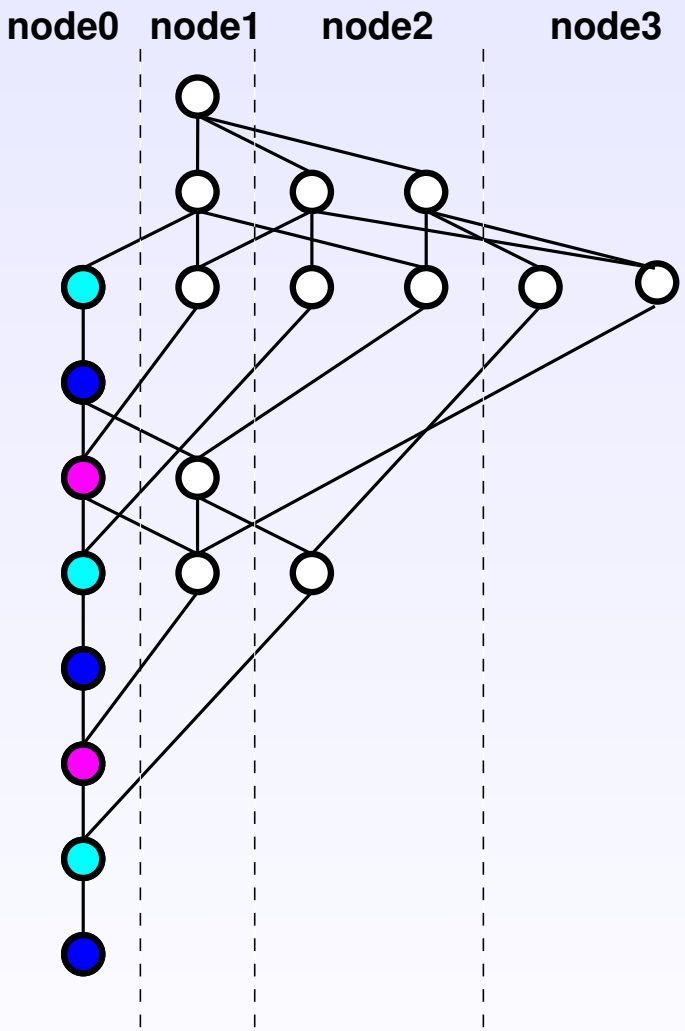


Node 1 execution

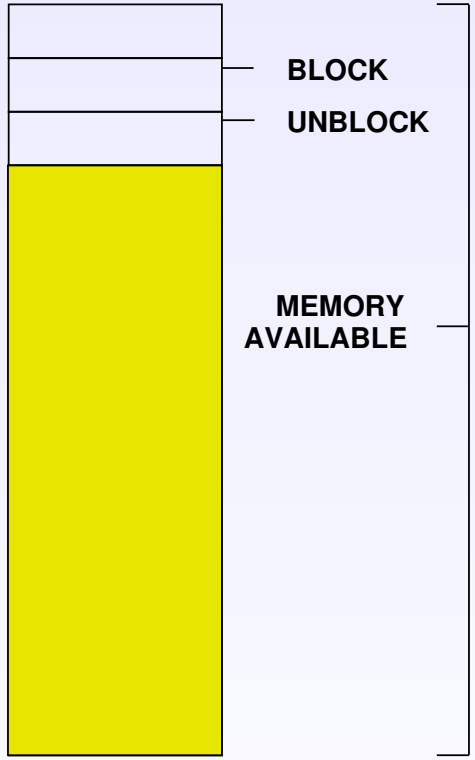
Outline

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

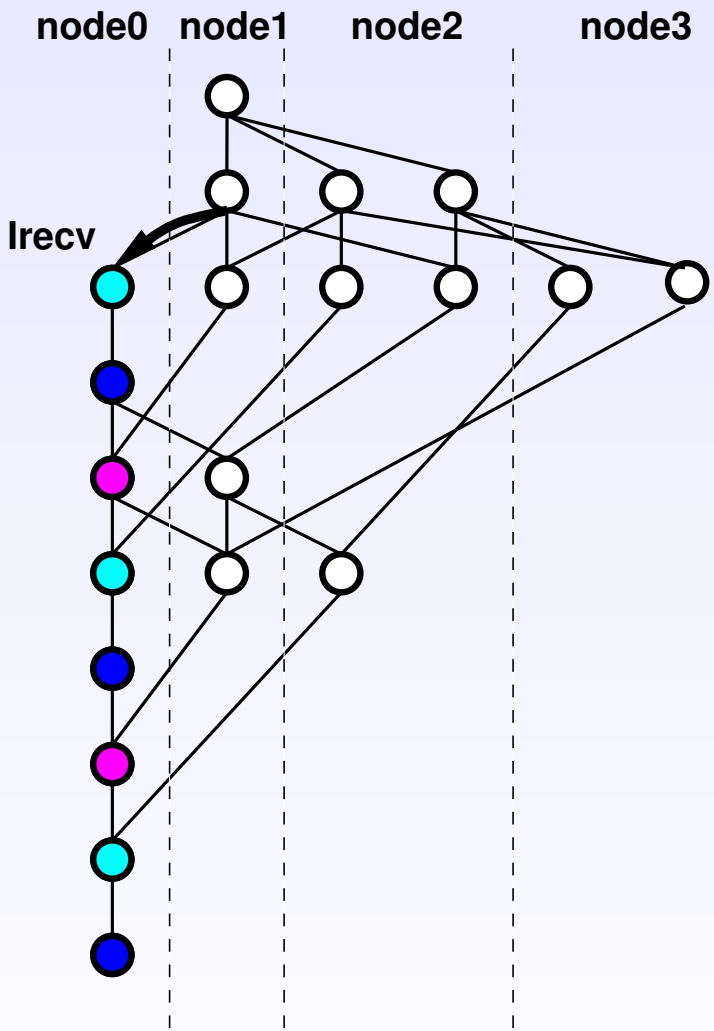


Node 0 execution

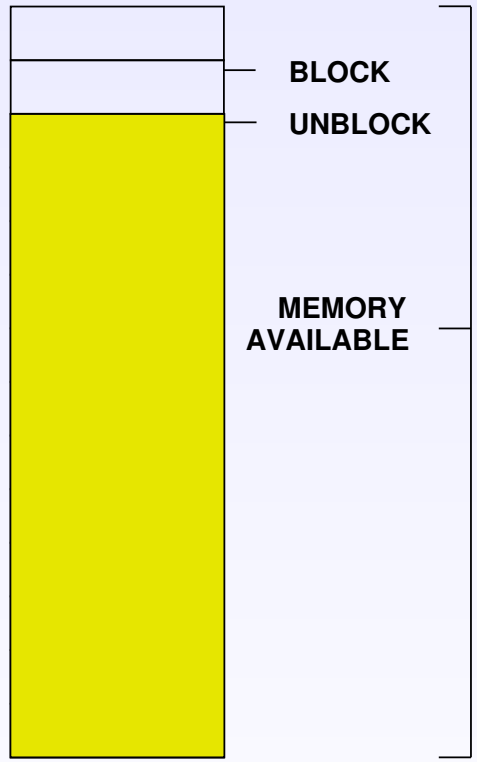


Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

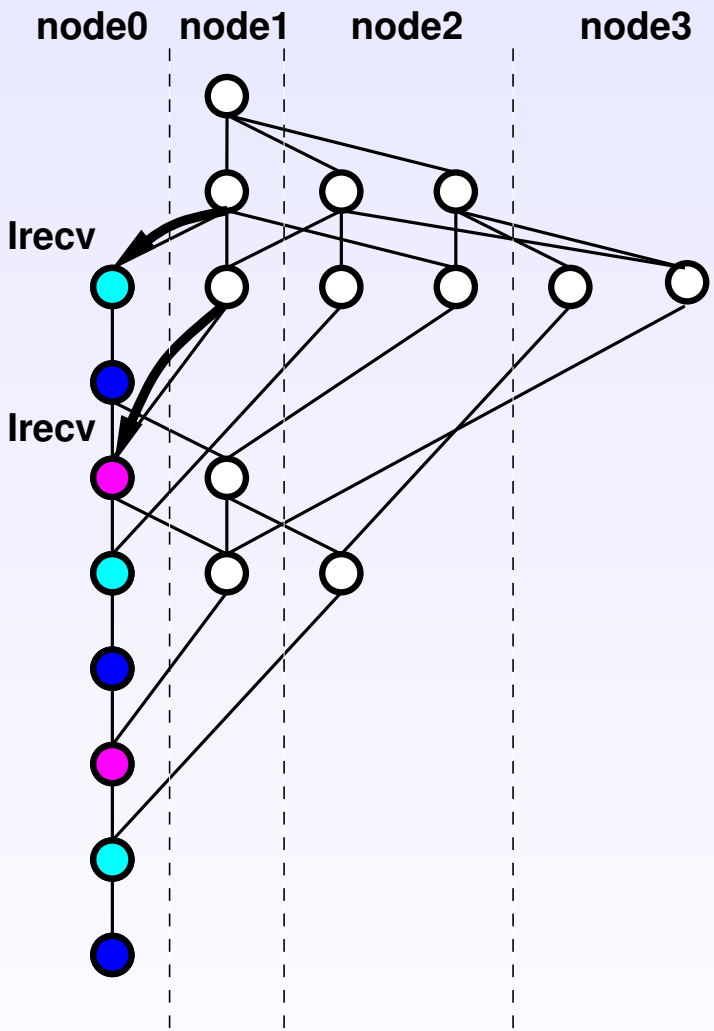


Node 0 execution

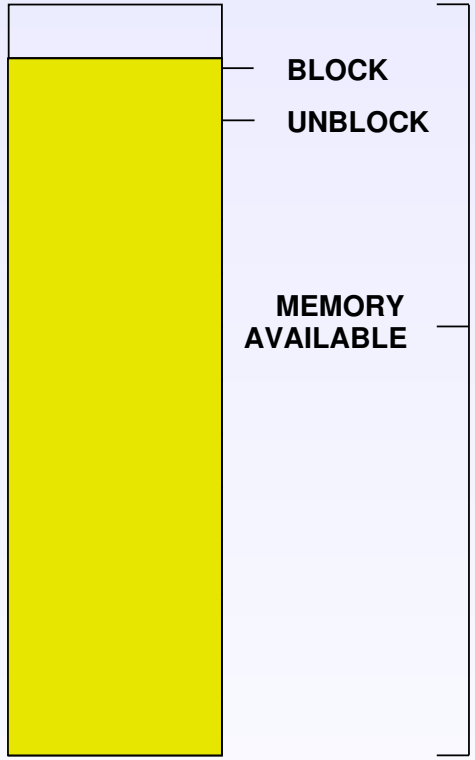


Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

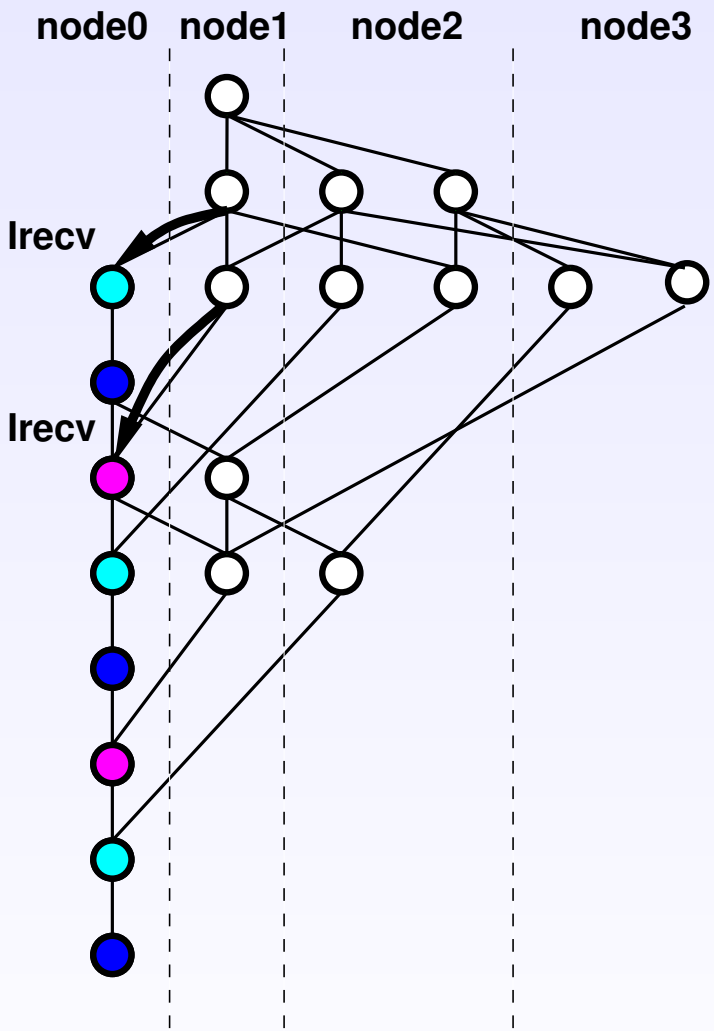


Node 0 execution

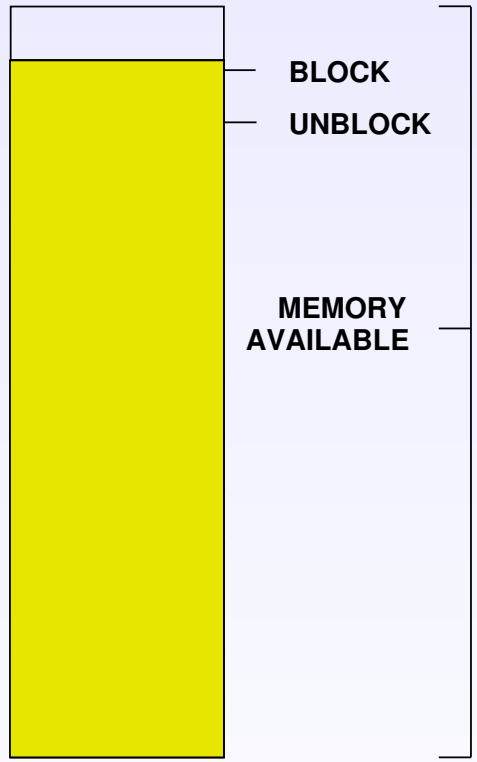


Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

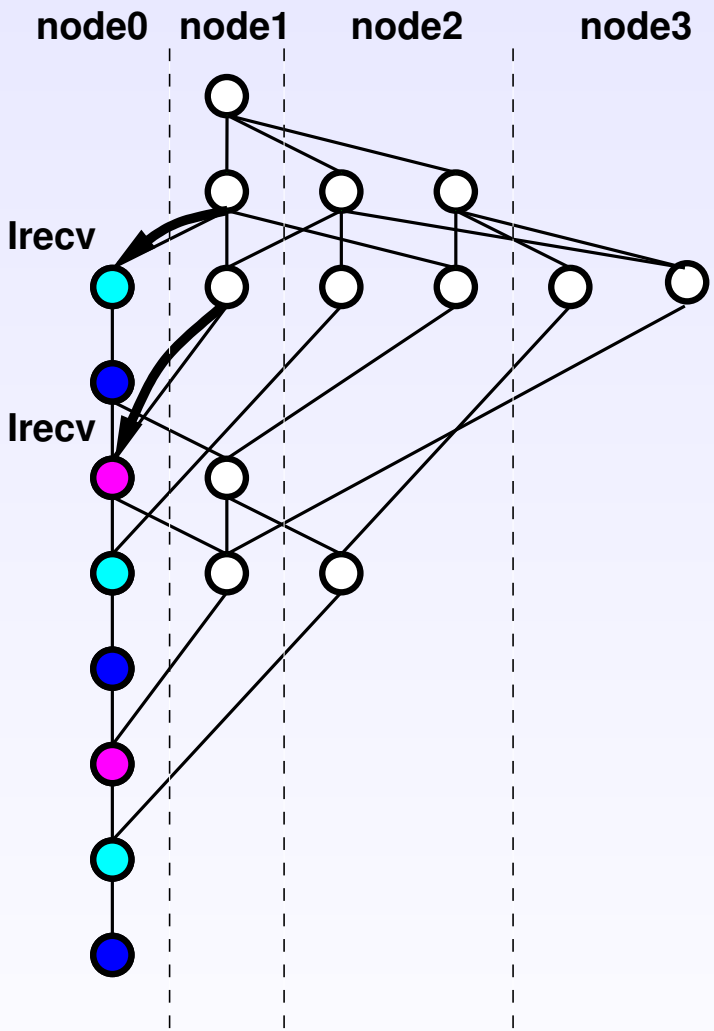


Node 0 execution

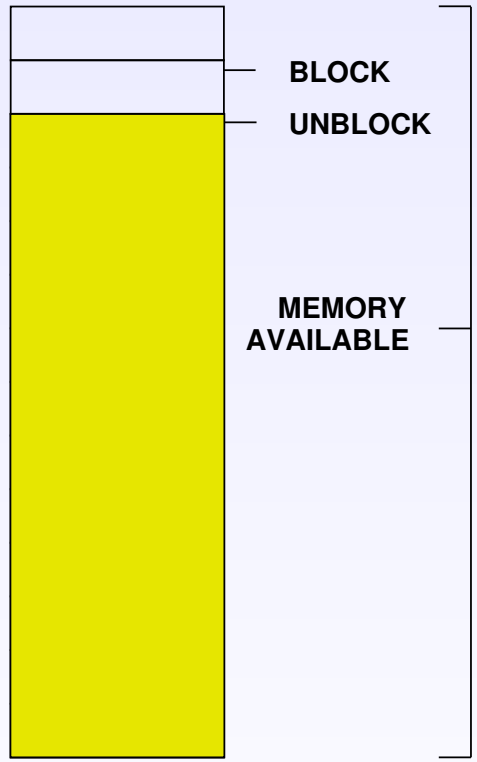


Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

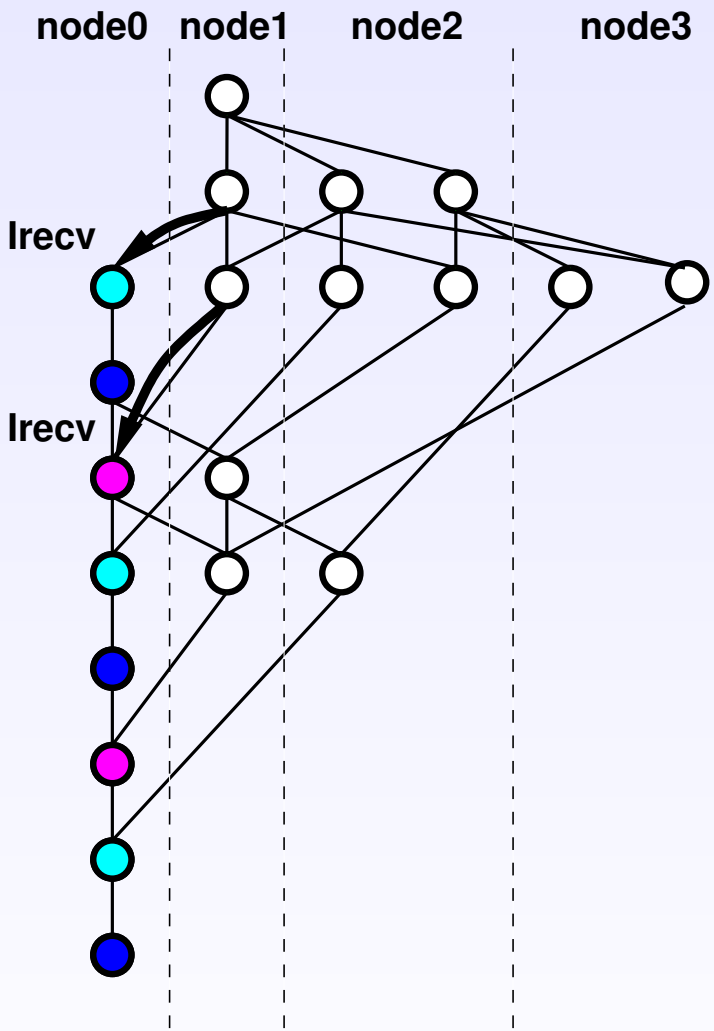


Node 0 execution

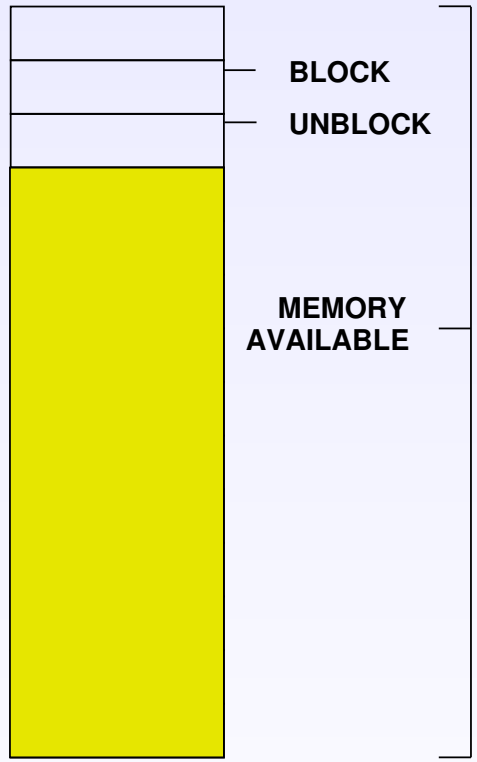


Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

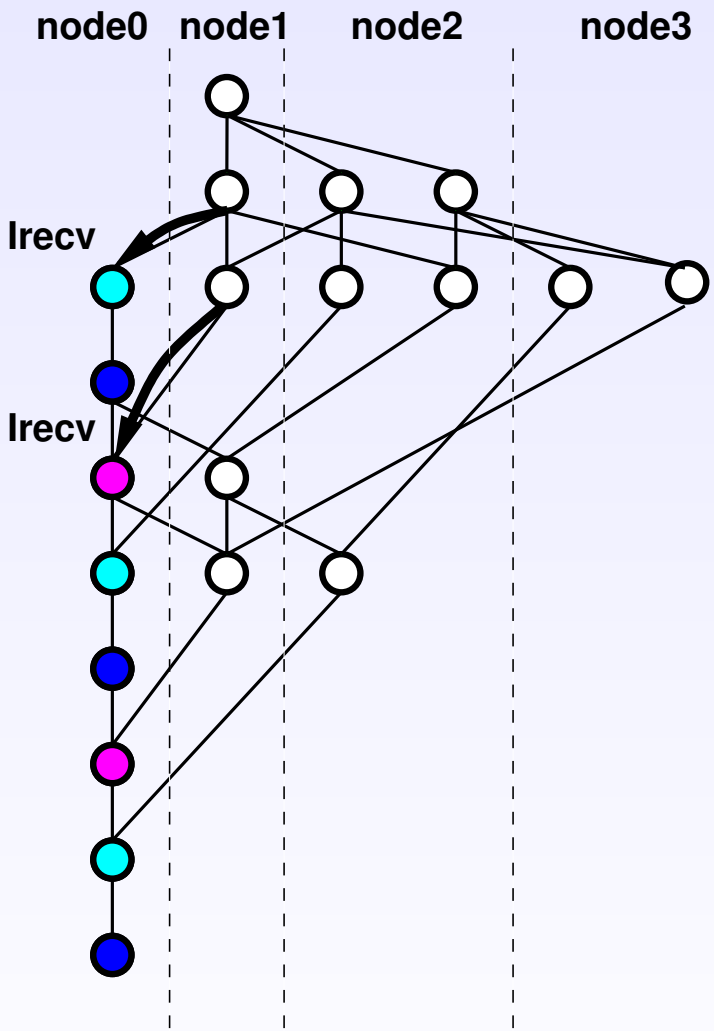


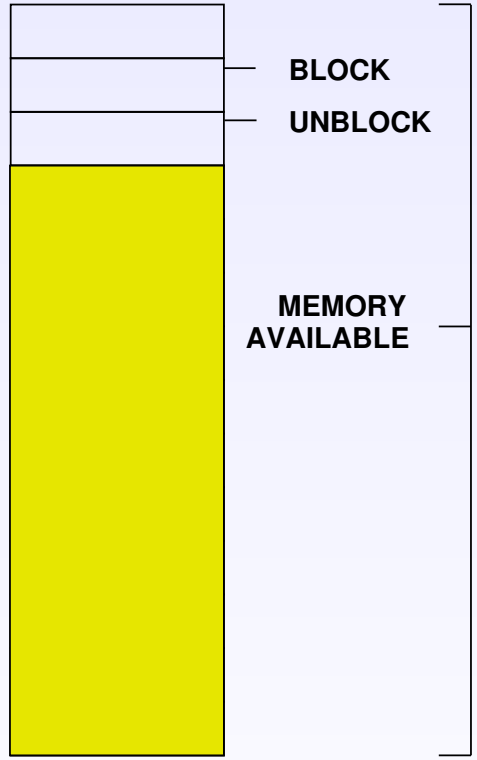
Node 0 execution



Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

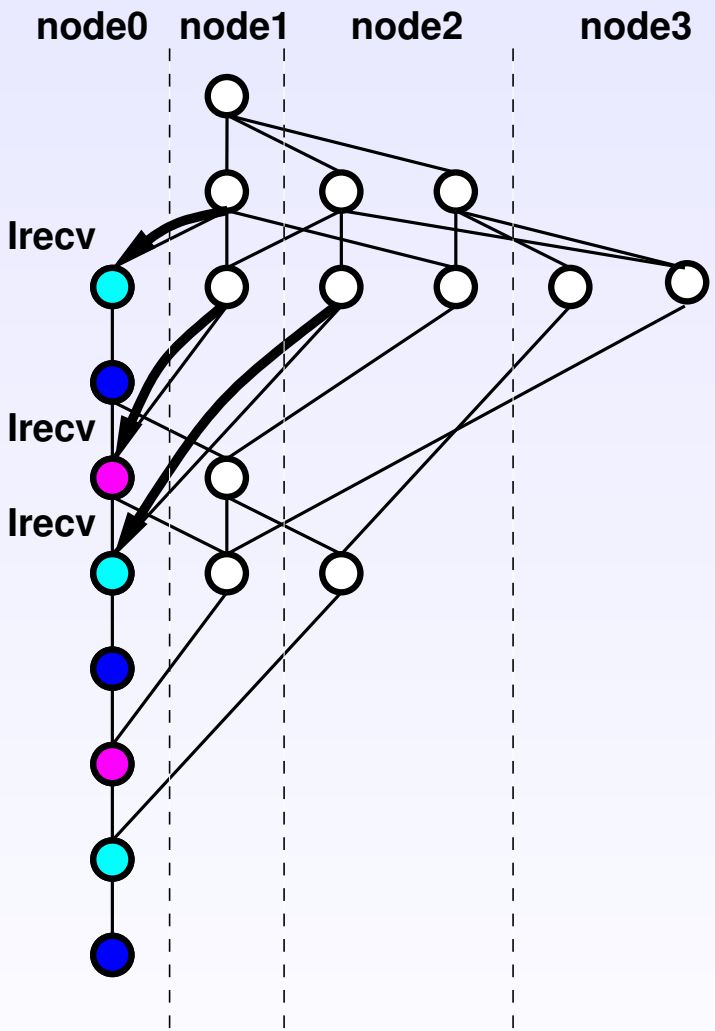


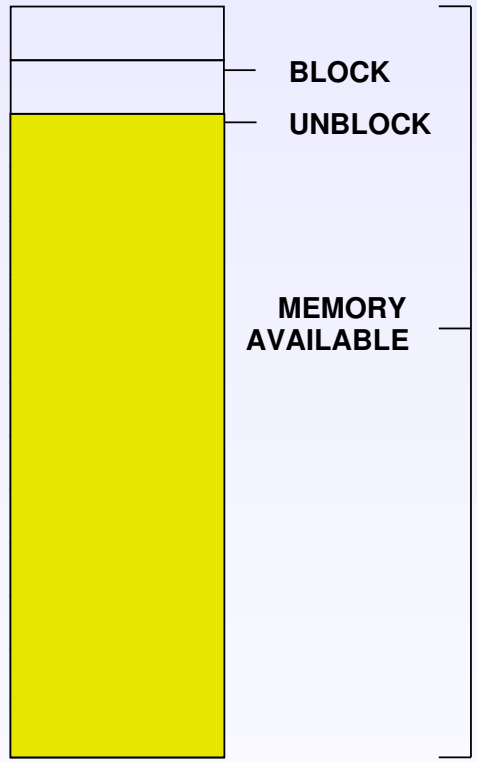


Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

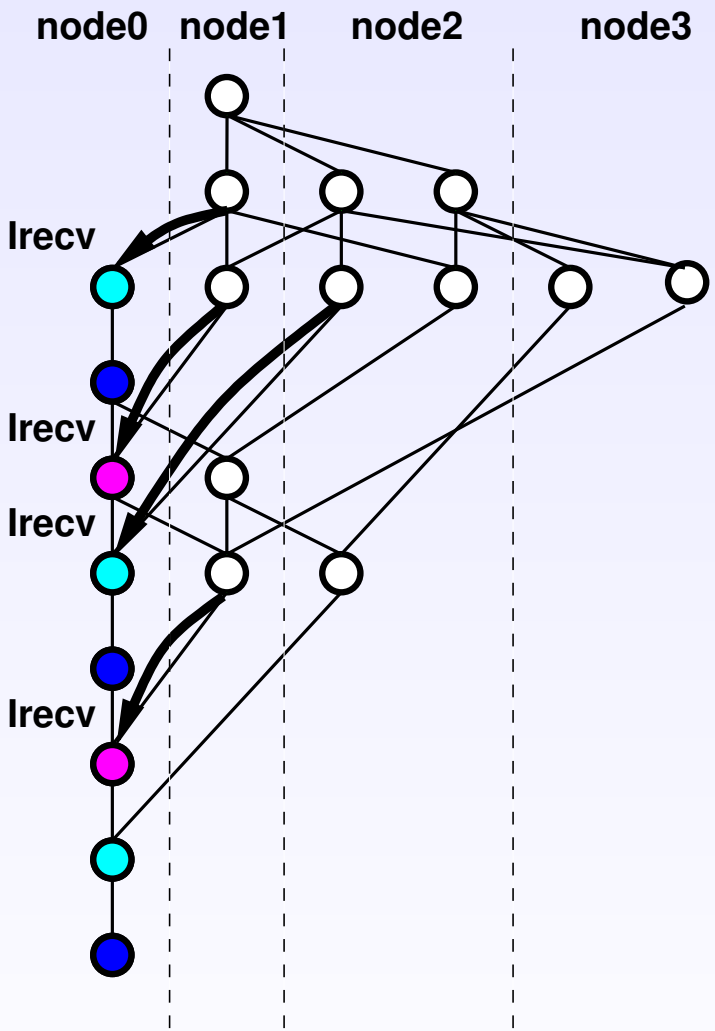




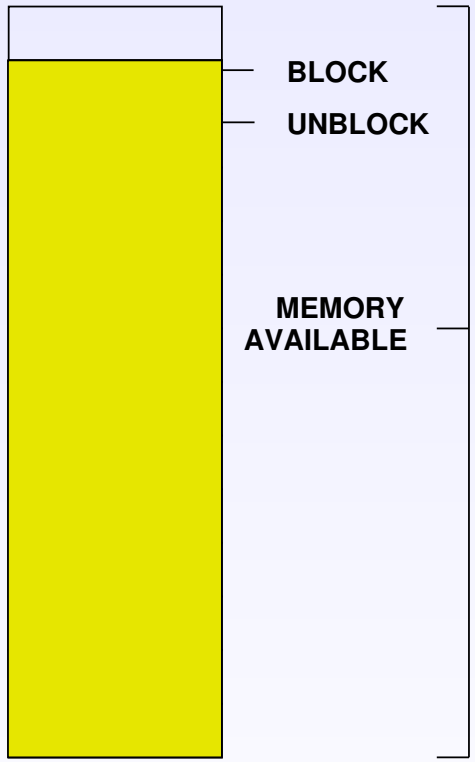
Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

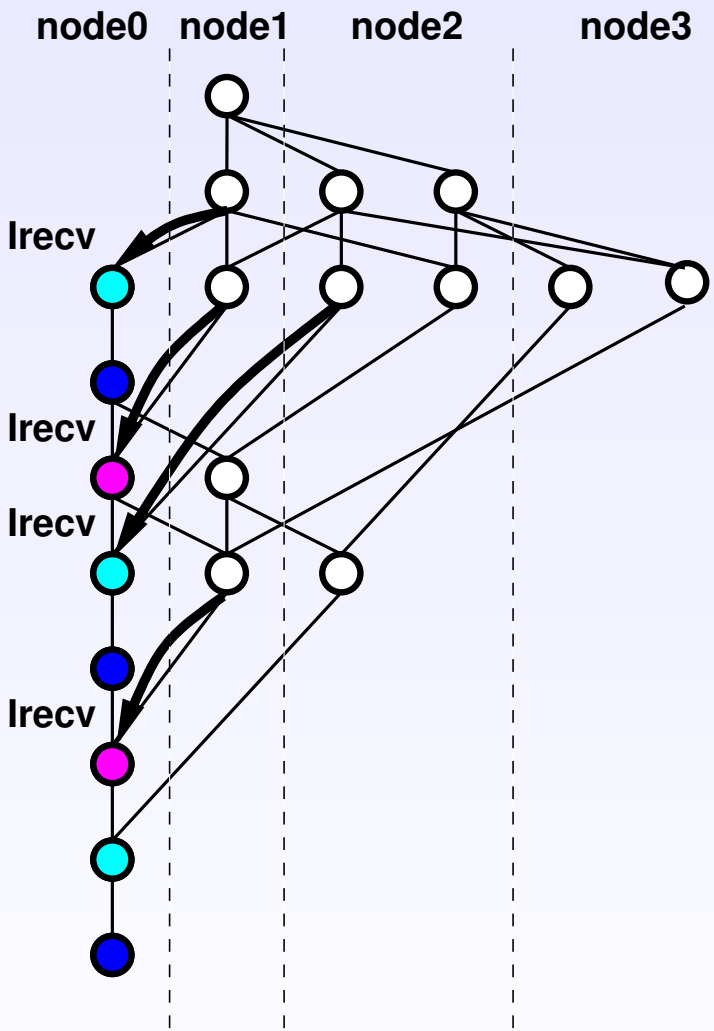


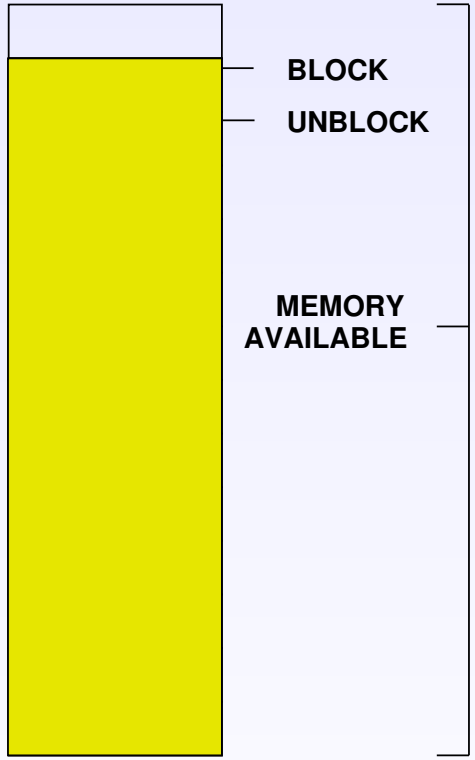
Node 0 execution



Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

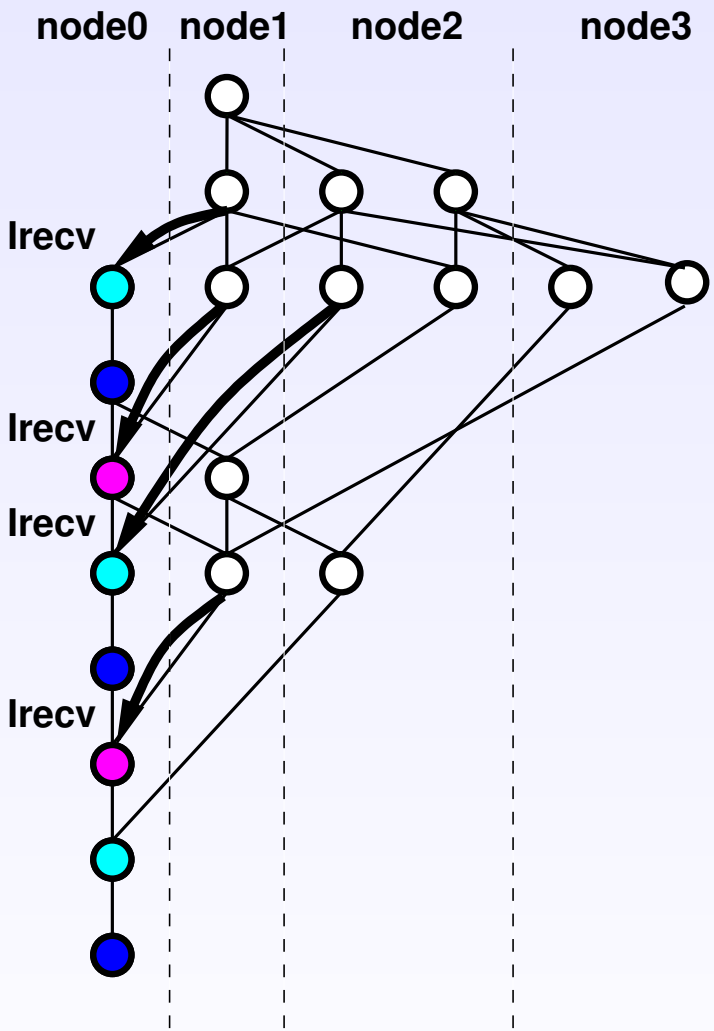




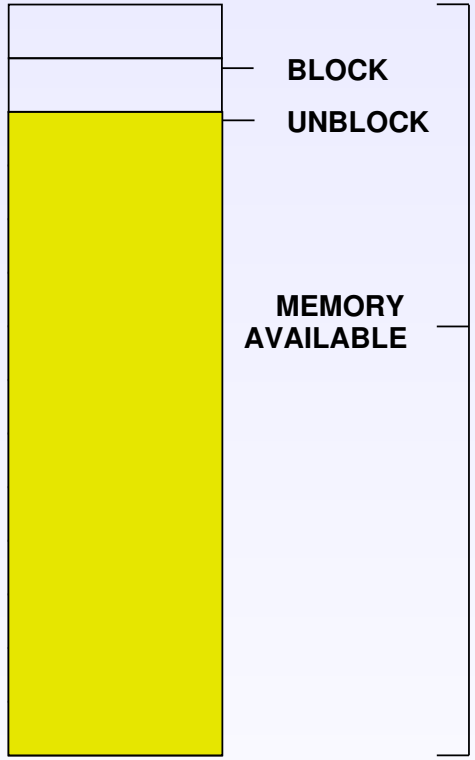
Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

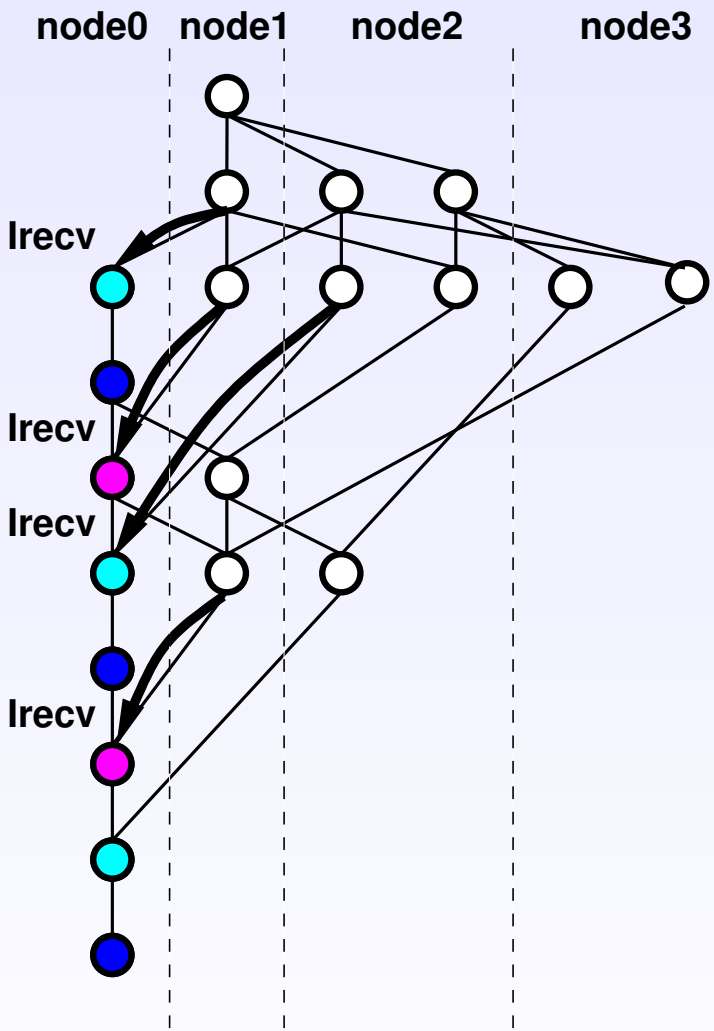


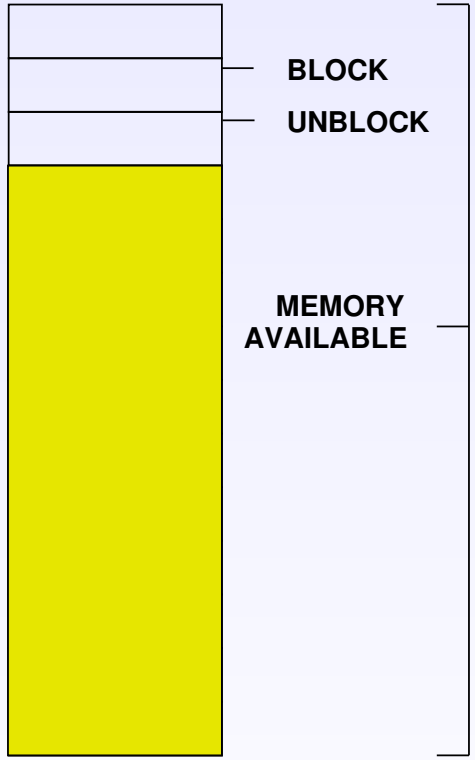
Node 0 execution



Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

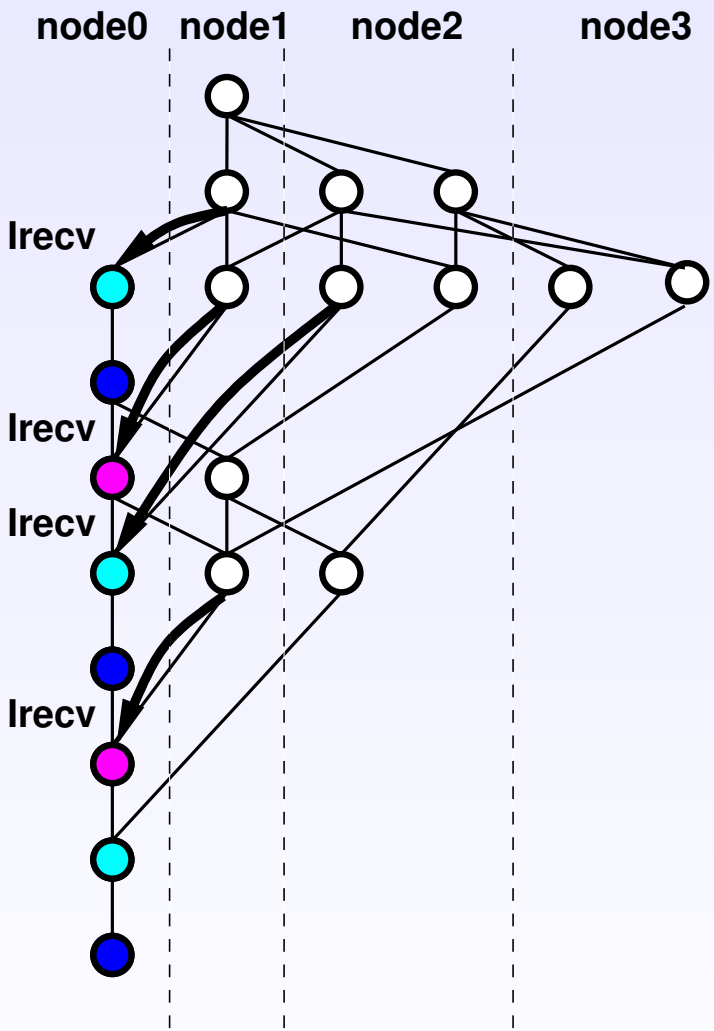


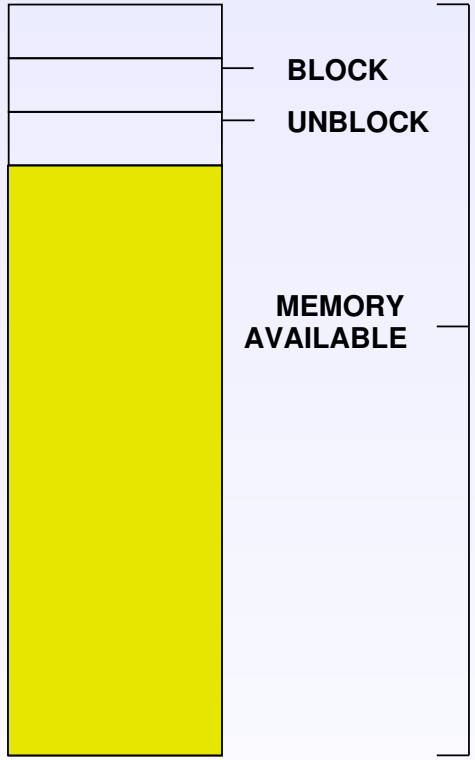


Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

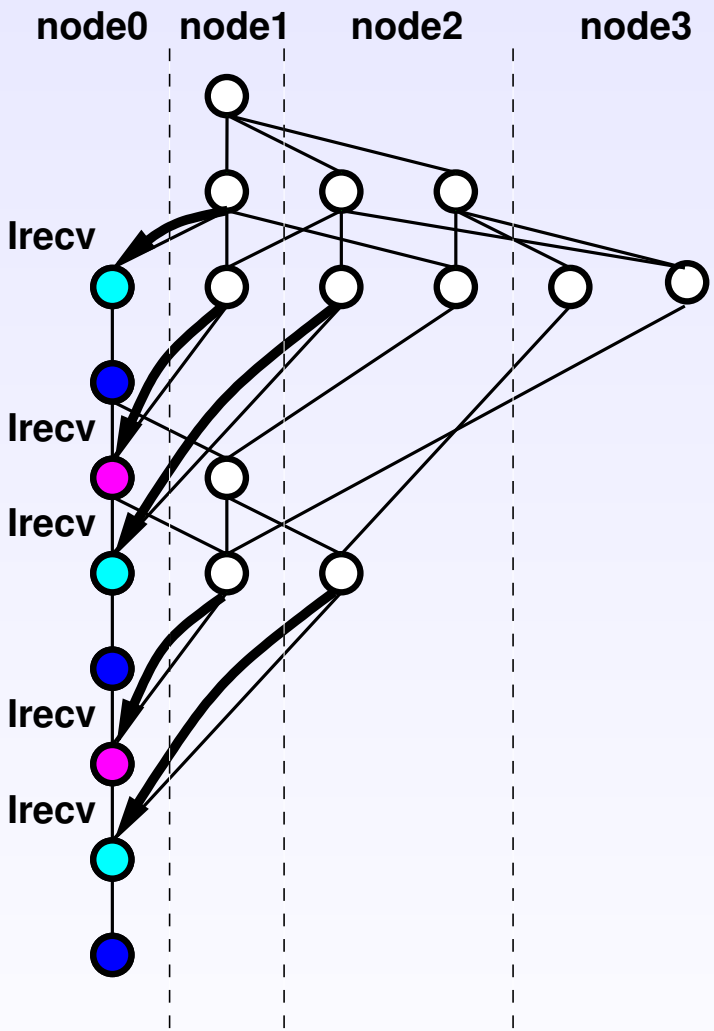


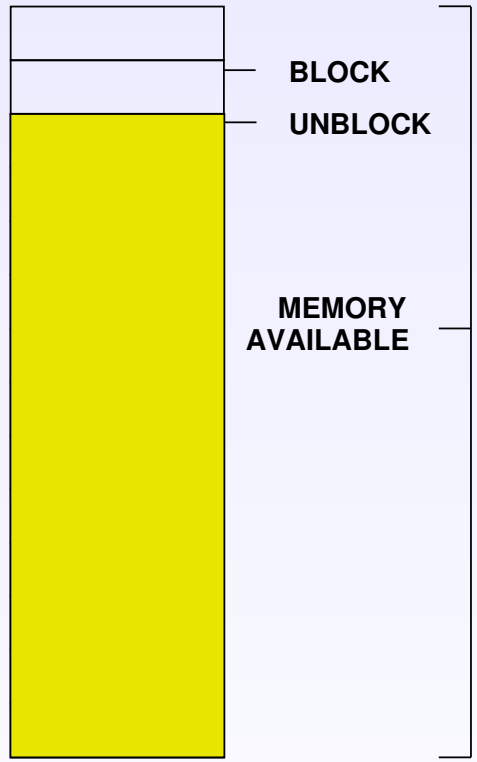


Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

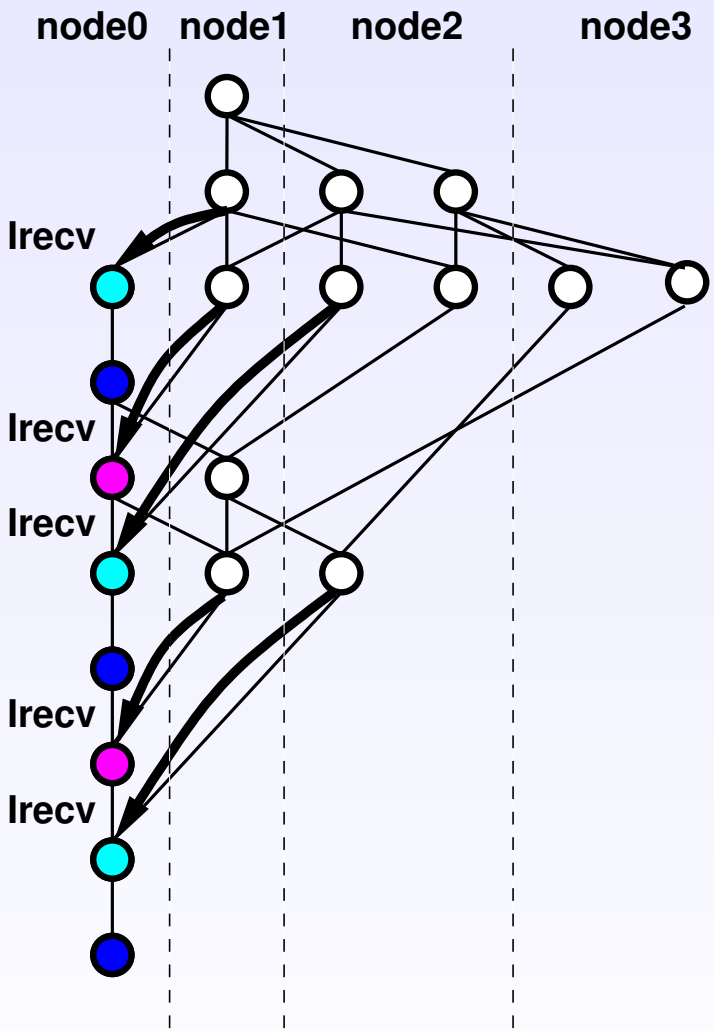


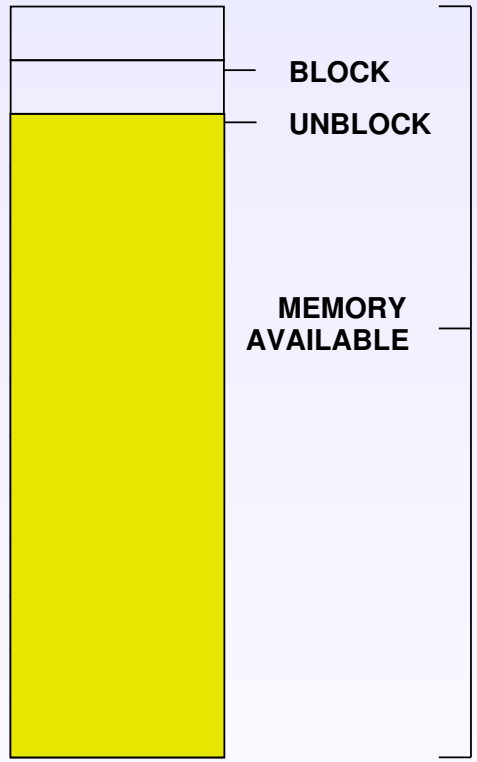


Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

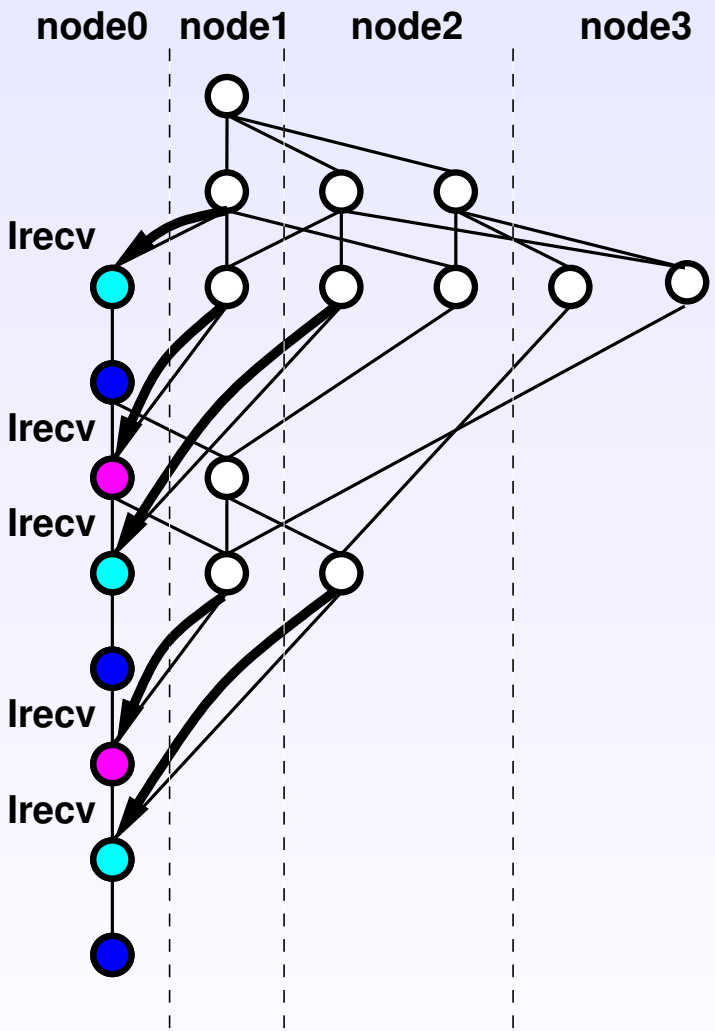


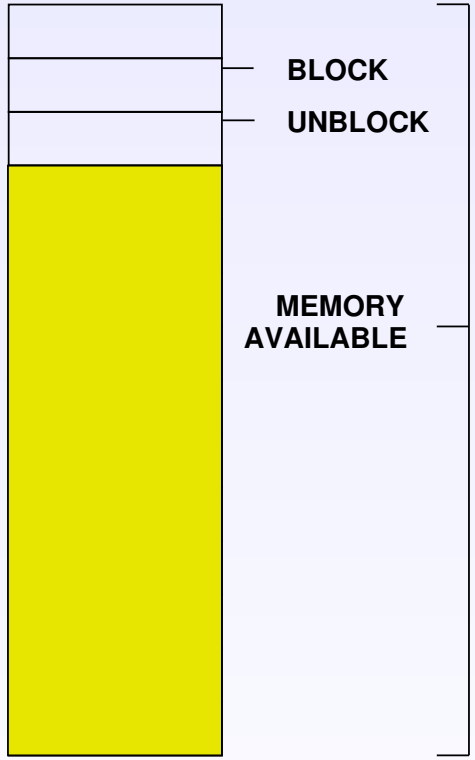


Node 0 execution

Memory control and sequential-task flow

```
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]);
  flush(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]);
    flush(A[i][j]);
  }
}

task_wait_for_all();
```

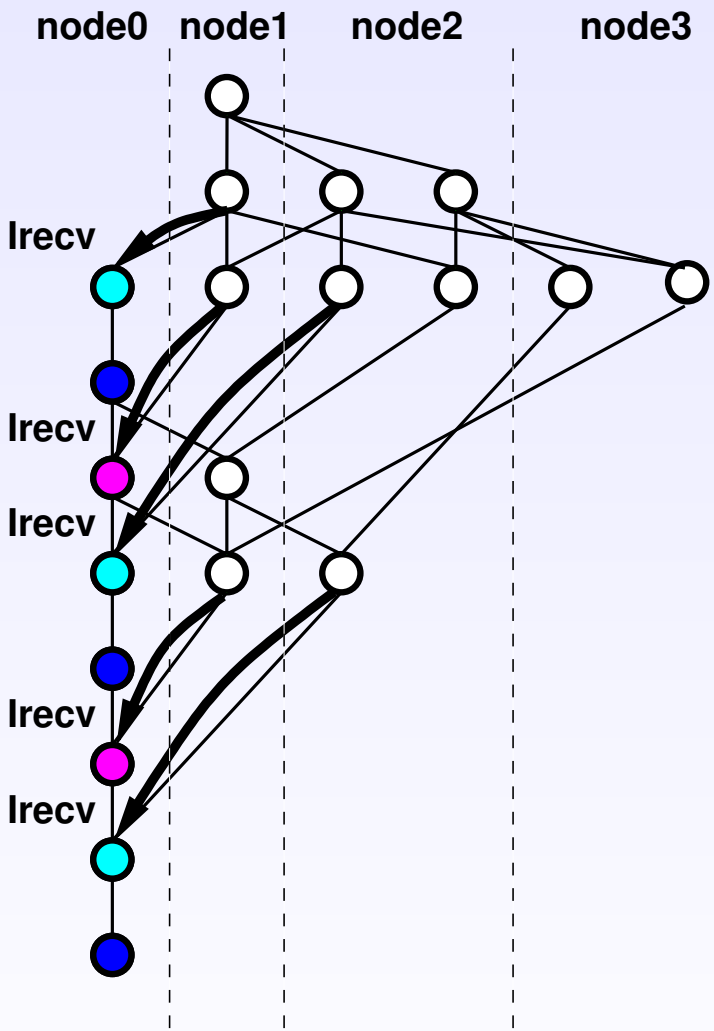


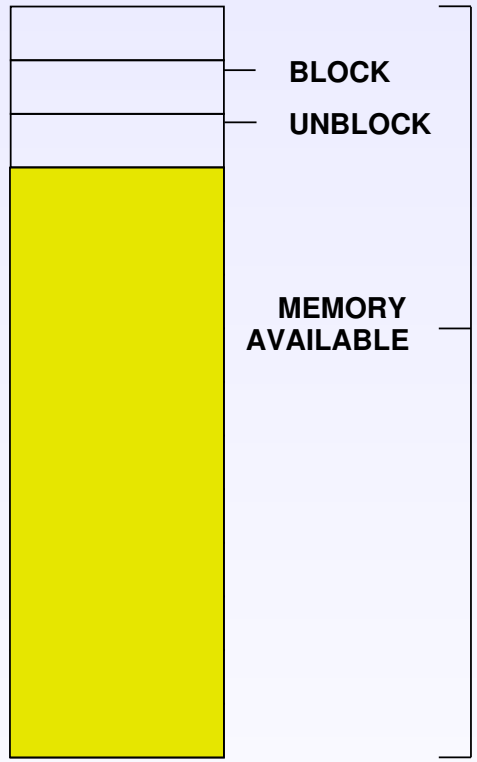


Node 0 execution

Memory control and sequential-task flow

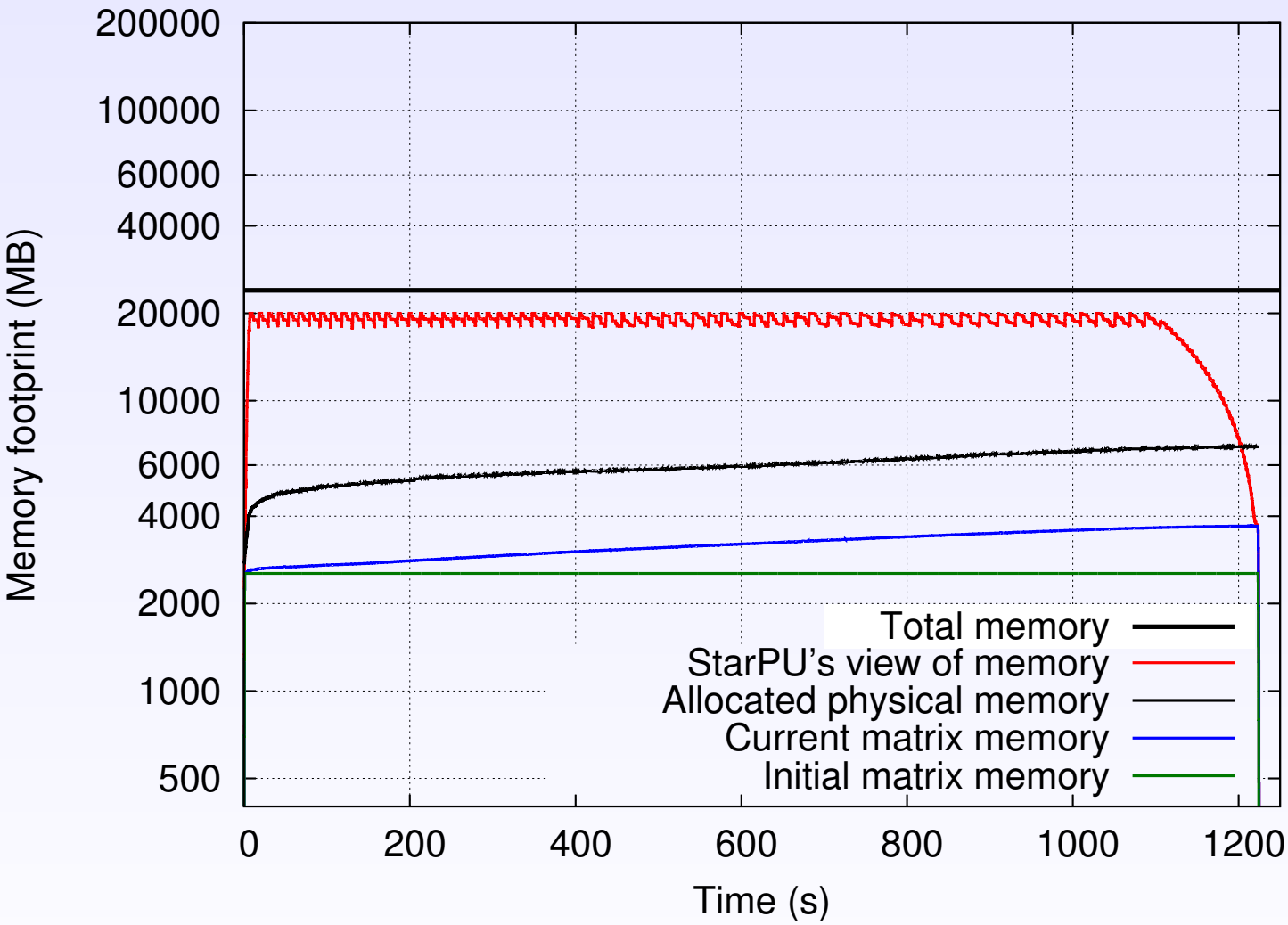
```
for (j = 0; j < N; j++) {
  POTRF (RW,A[j][j]);
  for (i = j+1; i < N; i++)
     TRSM (RW,A[i][j], R,A[j][j]);
  flush(A[j][j]);
  for (i = j+1; i < N; i++) {
    SYRK (RW,A[i][i], R,A[i][j]);
    for (k = j+1; k < i; k++)
      GEMM (RW,A[i][k],
            R,A[i][j], R,A[k][j]);
    flush(A[i][j]);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

task_wait_for_all();
```





Node 0 execution

Example of memory behaviour on a 24GB machine



Memory thresholds for blocking/unblocking task submission : 20GB/18GB

Outline

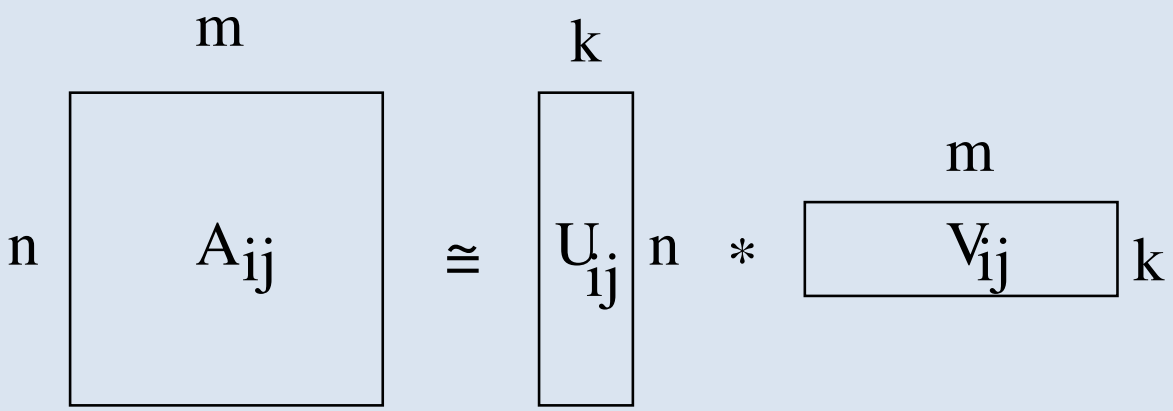
What if the size of data grows during the execution ?

Goal : overestimate the real memory subscription

- Need an upper-bound of data growth **for each submitted task**
- If cannot be known
  - Need a maximum value of data growth for each piece of data
- Example : linear algebra solver for Block Low-Rank (BLR) matrices

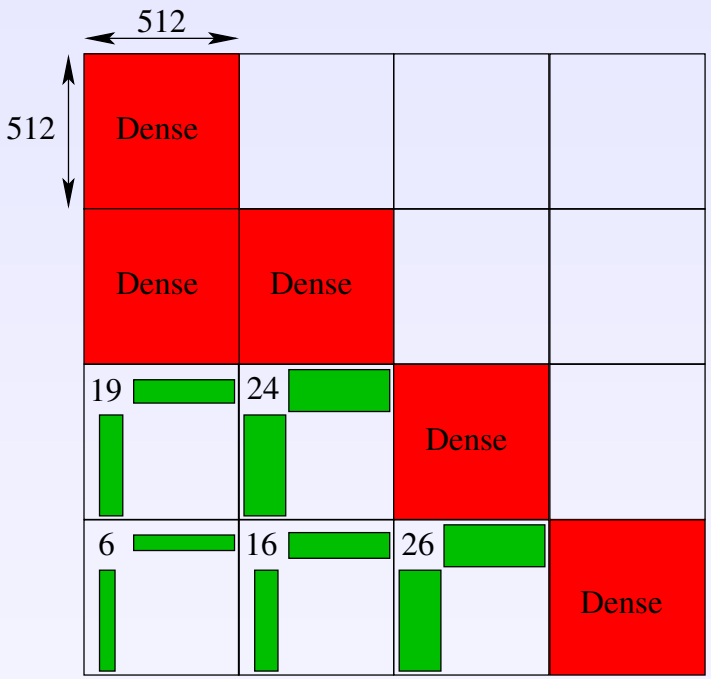
Block low-rank (BLR) matrices

Compressed tiles



- $A_{ij} \simeq U_{ij} V_{ij}^t$
- k is the *rank* of the compressed tile
- Average compression rate on target matrices : **over 97%**

Typical BEM matrix with compressed tiles



- The *rank* of each compressed tile of the matrix grows during the execution
  - Filling phenomenon of the matrix
  - Unpredictable prior to the execution
- A compressed tile **never** grows until it becomes dense again
  - Upper-bound of data growth : size of the dense tile

Memory control with upper-bound and BLR solver

Upper-bound of data growth : size of the dense tile

- Cannot overestimate the size of all tiles with dense tiles
  - Average overestimation equals the compression rate **(over 97%)**

Pragmatic solution

- Do not overestimate the size of local tiles
  - Memory margin needed to compensate for local growth of data
- Overestimate the memory space required for remote data
  - And correct it when the data is recieved
- Works in practice
  - Memory overestimation of remote data overlaps with the local filling of the matrix

Outline

Experimental setup

Application settings

- Complex double-precision distributed compressed **Cholesky factorization**
- Tile size: **512x512**
- Matrix order: **450.000** unknowns
- **Chameleon** solver (Inria HiePACS team) on top of **StarPU** runtime

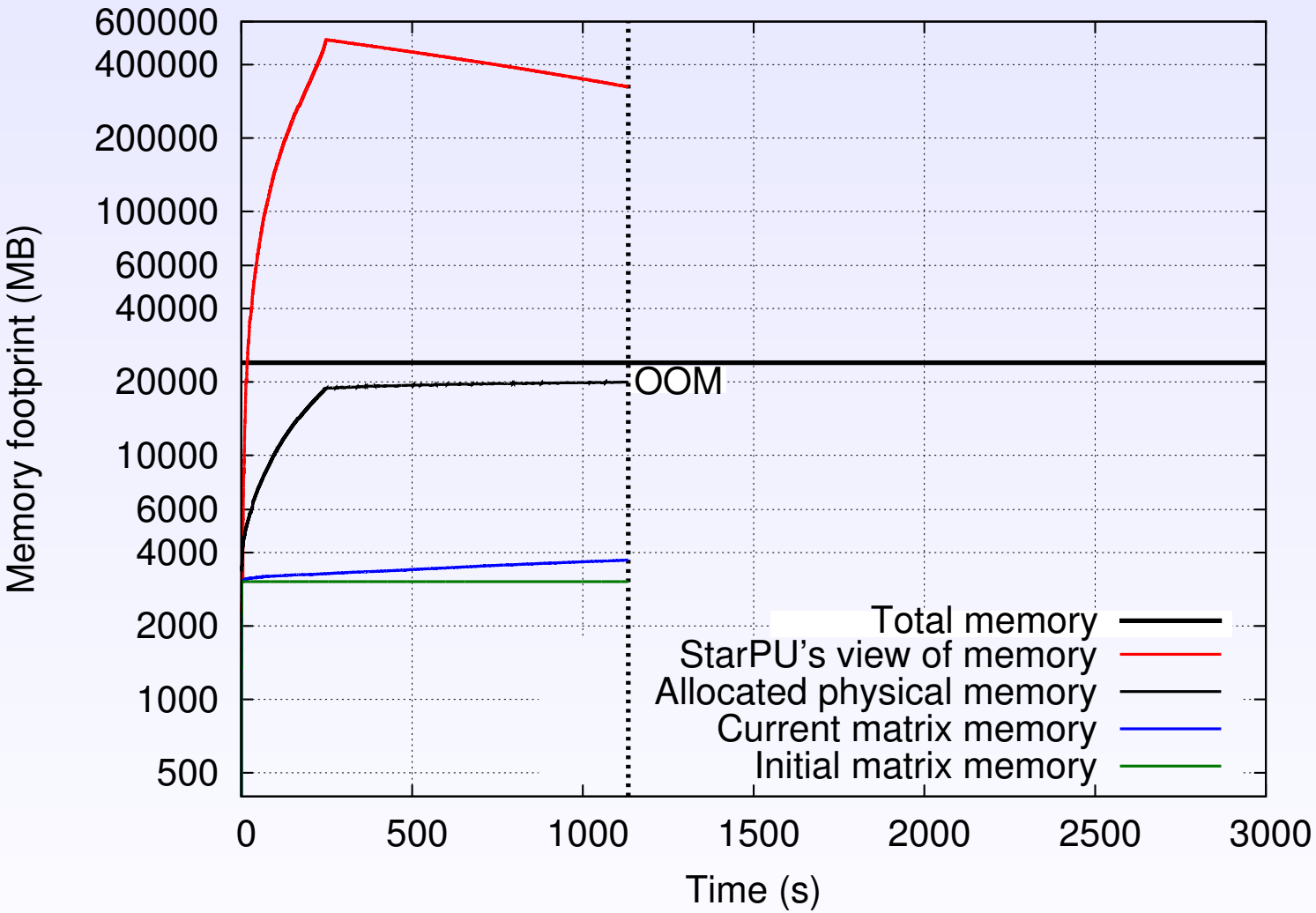
Experimental cluster

- From **16 to 81 nodes** of TERA100 Hybrid cluster
- 2 Intel Xeon X5620 @ 2.40 GHz (**8 cores** per node)
- **24 GB RAM** per node

Memory control settings

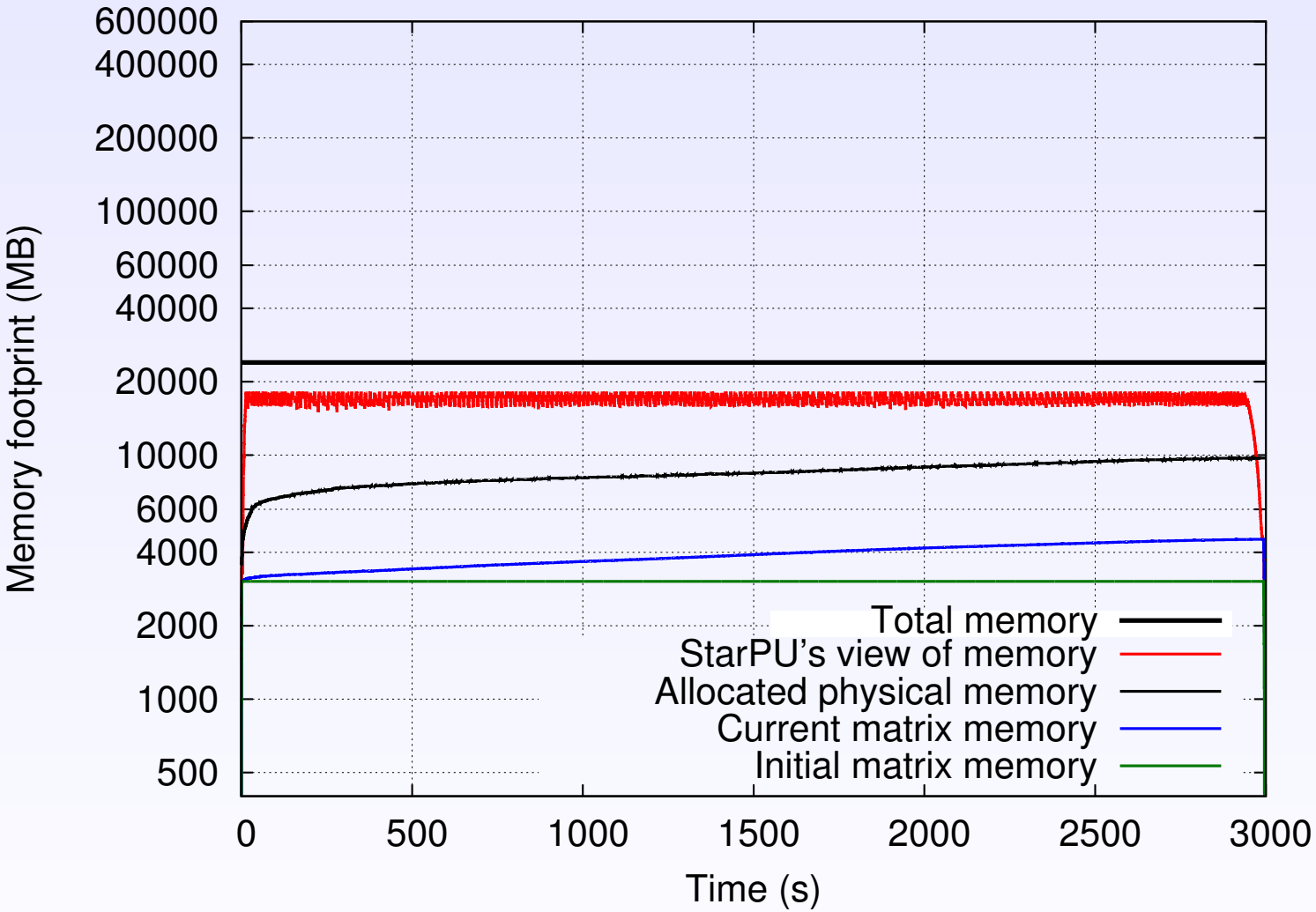
- Blocks when reaches up to **18GB** of memory occupancy
- Unblocks when goes down below **16GB** of memory occupancy

Without memory control (25 nodes, 450k unknowns)



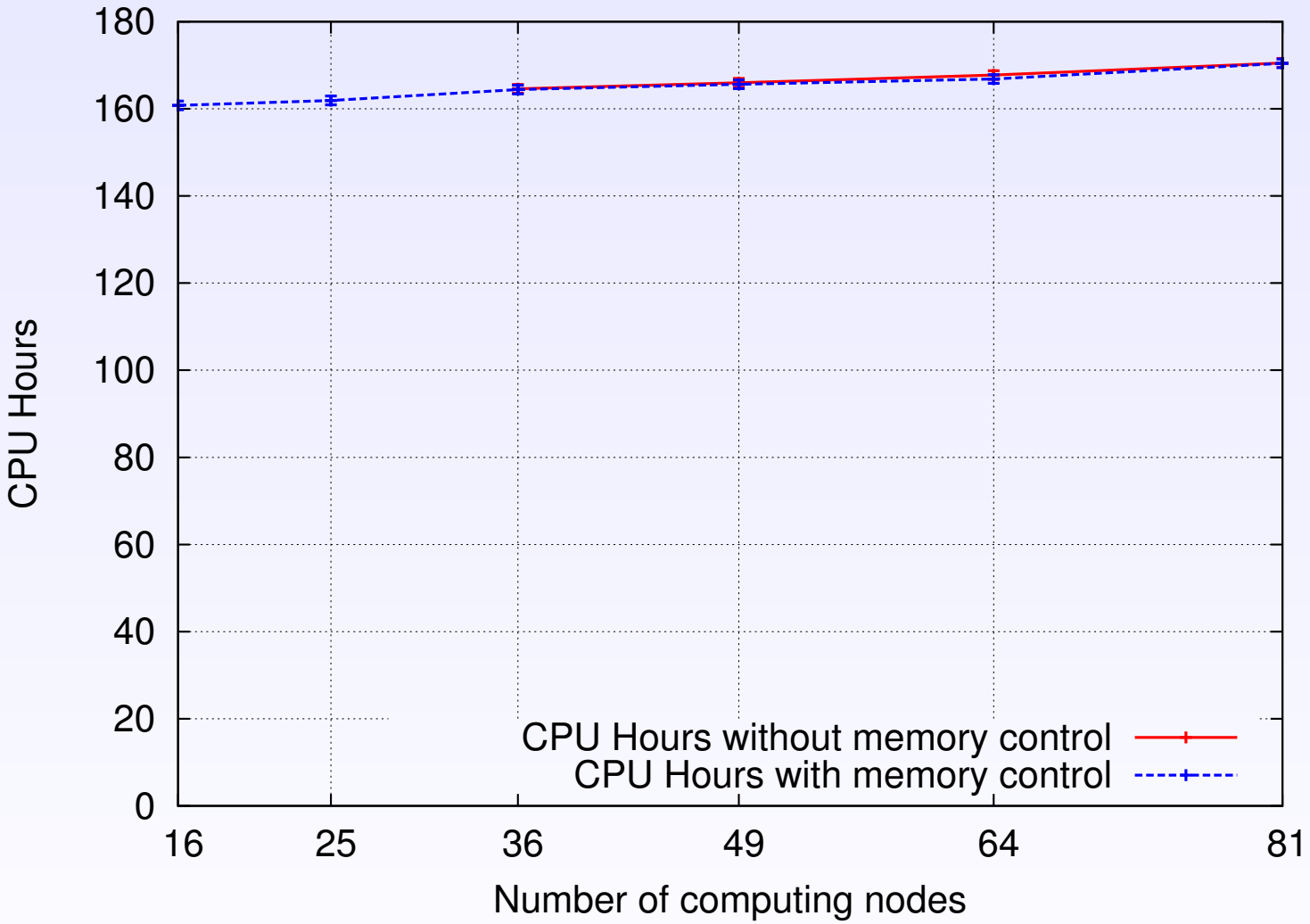
Memory oversubscription for remote data + filling of the matrix = out-of-memory

With memory control (25 nodes, 450k unknowns)



The memory overestimation of remote data overlaps with the local filling of the matrix

Performance results (strong scaling, 450k unknowns)



The memory control does not impact the performance of the application, and allows to use fewer computing nodes

Outline

Conclusion

Memory control with the distributed STF model

- Memory allocations performed at submission time
- When not enough memory available, blocks task submission
- When enough memory have been freed, unblocks task submission

What if the size of data grows during the execution ?

- Need an upper-bound of data growth for each task
- If cannot be known
  - Need at least a maximum value of data growth
  - Works in practice
    - Distributed BLR solver successfully implemented with StarPU

Future work

Task submission order can provoke out-of-memory scenarios

- Example : submit all tasks that allocates memory first, then tasks that frees memory

- Application-level solution : change the task submission order
  - Group the submission of tasks which allocates and deallocates a specific data
- Runtime-level solution : memory-driven activation level for tasks
  - Task ready when there is enough memory available to allocate its data
  - Non-blocking task submission

Task execution order can provoke out-of-memory scenarios

- Memory-aware task scheduling heuristics

Thank you !

Questions ?

MORSE

Research context

The MORSE research project 2011-2013 - 2014-2016

- People
  - ICL
  - INRIA Bordeaux Sud Ouest (HiePACS, RealOpt, STORM)
  - KAUST
  - UCD
- Solvers
  - Sparse direct (PaStiX, qr_mumps)
  - H-matrix (EADS)
  - FMM (ScalFMM)
  - Dense (Chameleon)
- Task-based runtime systems
  - Insert task paradigm: StarPU, Quark
  - JDF: PaRSEC