TASK-BASED PROGRAMMING MODELS FOR SCALABLE ALGORITHMS

New features for scalable algorithms

E. Agullo, A. Buttari, A. Guermouche, A. Jego

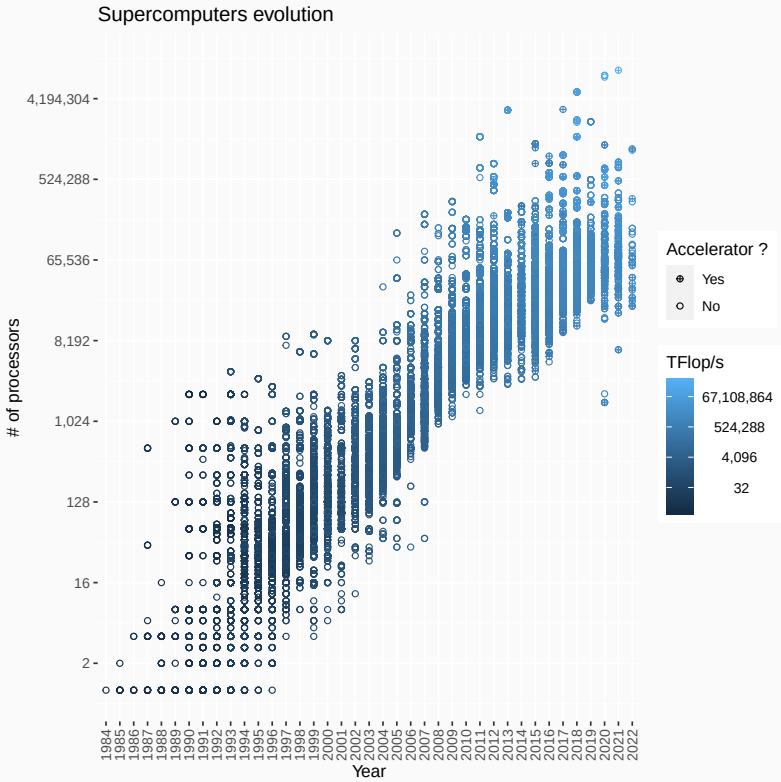
28 mars 2023

INTRODUCTION

Algorithms, architectures and programming models

Architectures are becoming increasingly **large**, **heterogeneous** and **complex**. This has two effects :

- Algorithms have to handle communications to achieve high scalability
- Software should deliver a reliable abstraction to build mathematical software

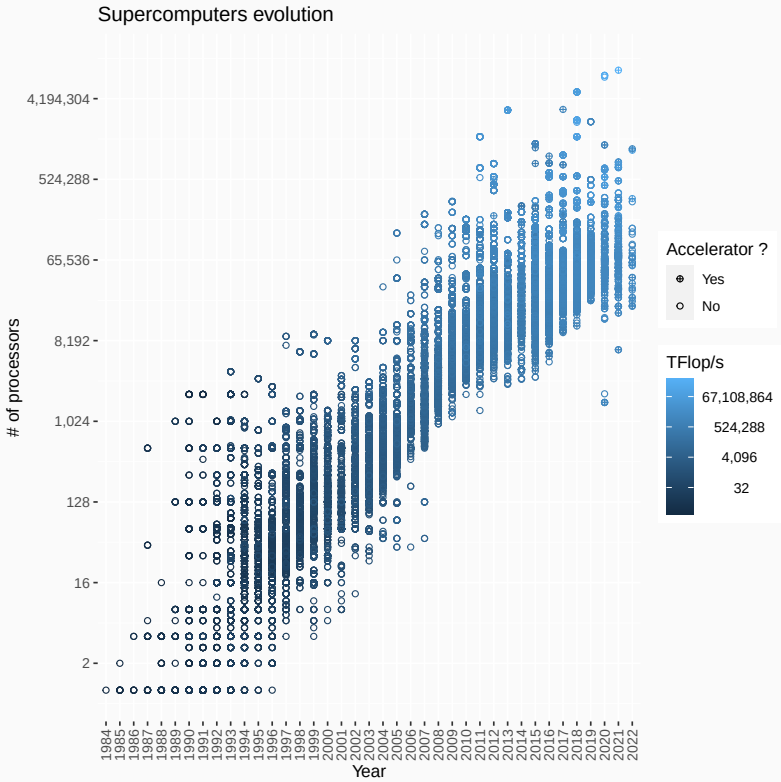


Are currently available parallel programming models and runtimes expressive enough to implement scalable algorithms in an efficient, portable and maintainable way ?

Algorithms, architectures and programming models

Architectures are becoming increasingly **large**, **heterogeneous** and **complex**. This has two effects :

- Algorithms have to handle communications to achieve high scalability
- Software should deliver a reliable abstraction to build mathematical software

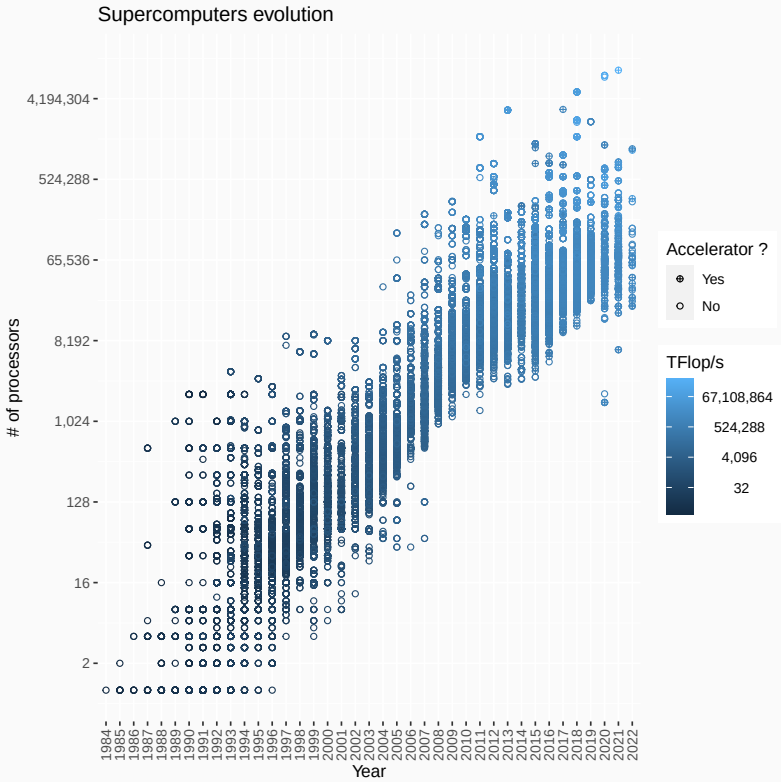


Are currently available parallel programming models and runtimes expressive enough to implement scalable algorithms in an efficient, portable and maintainable way ?

Algorithms, architectures and programming models

Architectures are becoming increasingly **large**, **heterogeneous** and **complex**. This has two effects :

- Algorithms have to handle communications to achieve high scalability
- Software should deliver a reliable abstraction to build mathematical software



Are currently available parallel programming models and runtimes expressive enough to implement scalable algorithms in an efficient, portable and maintainable way ?

Algorithms, architectures and programming models

We have taken the following examples to answer this large question

- Algorithms : scalable dense matrix multiplication (GEMM) algorithms such as SUMMA and 2.5D SUMMA

$$C = \alpha AB + \beta C$$

- Software : task-based parallel programming paradigm through the sequential task flow (STF) model

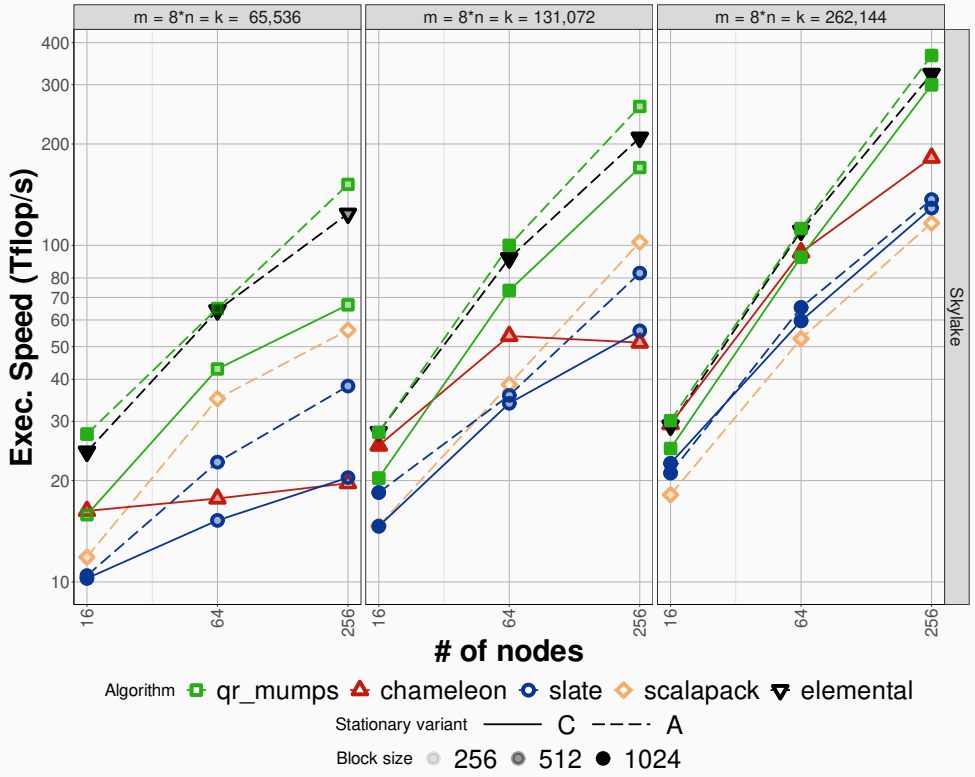
E. Agullo, A. Buttari, A. Guermouche, J. Herrmann et A. Jego. « Task-Based Parallel Programming for Scalable Matrix Products Algorithms ». In : ACM Transactions on Mathematical Software (2023)

Algorithms, architectures and programming models

Takeaway

Three nested loops yield high performance.

One code encapuslates 6 variants ({2D, 3D} × {A, B, C})




```
!data_map: A,B,C distributed on
      one layer
!task_map: executing node of
     A_{i,l} B_{l,j}
!dyn_tree: type of broadcast
      tree
call bind_methods(C, init, sum)
do i=1,m
  do j=1,n
    do l=1,k
      call insert_task(gemm,
  A_{i,l}:R,
  B_{l,j}:R,
  C_{i,j}:RANK_REDUX,
  task_map[i,j,l])
    end do
  end do
end do
```

Algorithms, architectures and programming models

The core features useful in the extended STF model have been

- Mapping tasks
- Implicit reduction patterns
- Dynamic broadcast communications

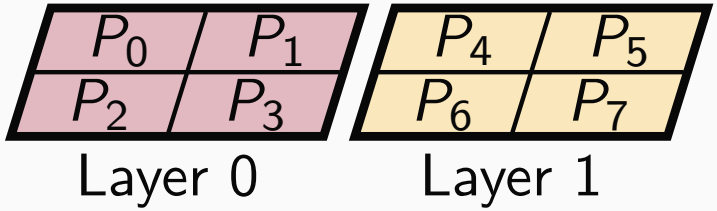
Are these features enough to achieve scalability in other algorithms through the STF model ?

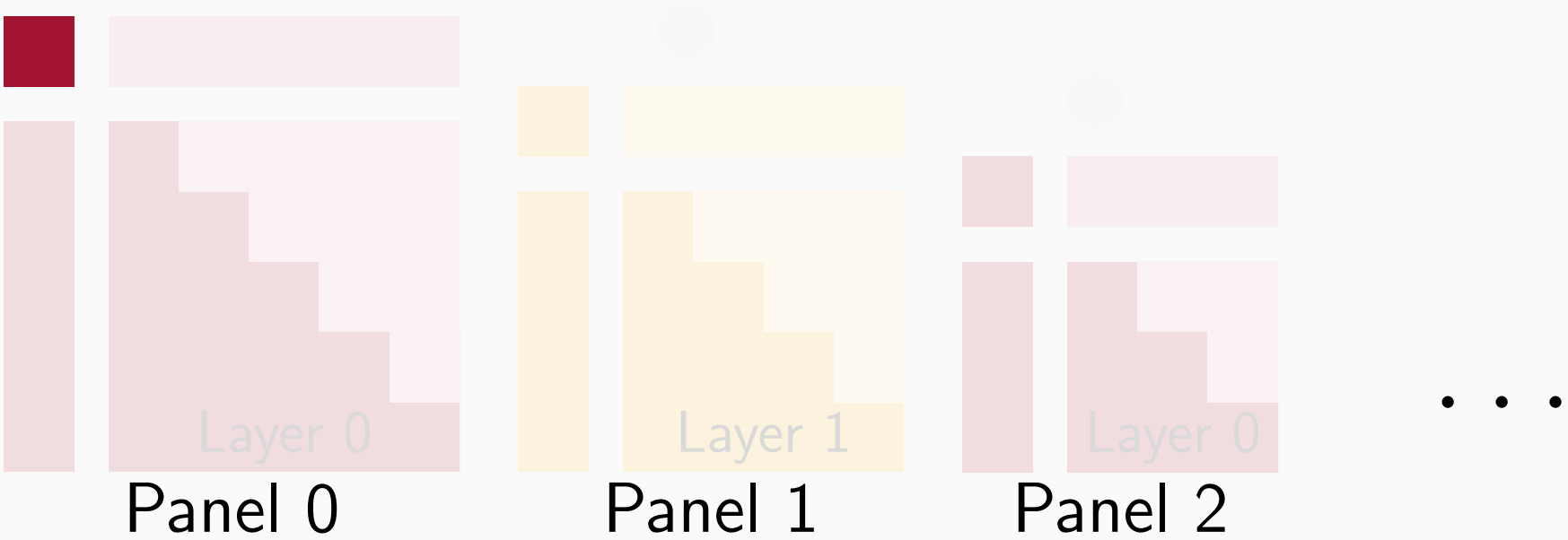
BACKGROUND

Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

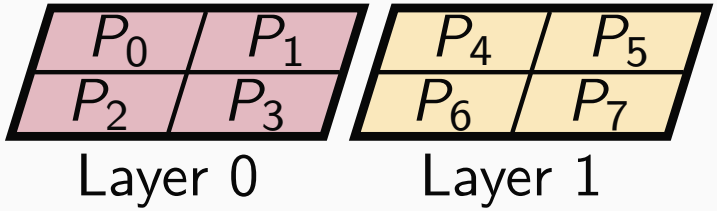


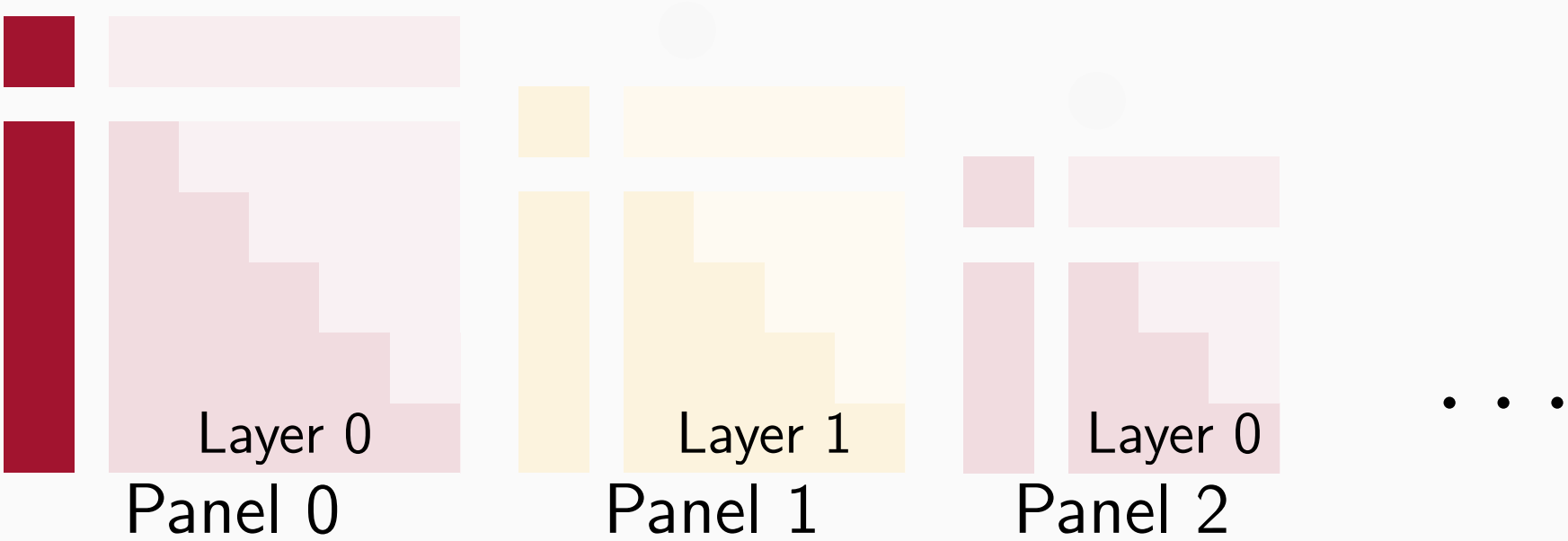


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

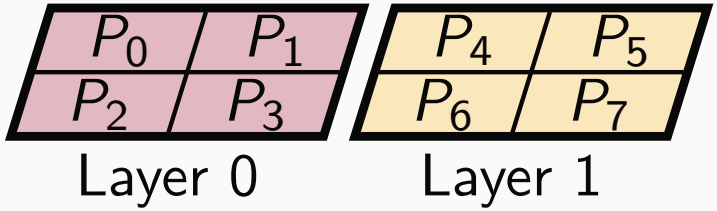


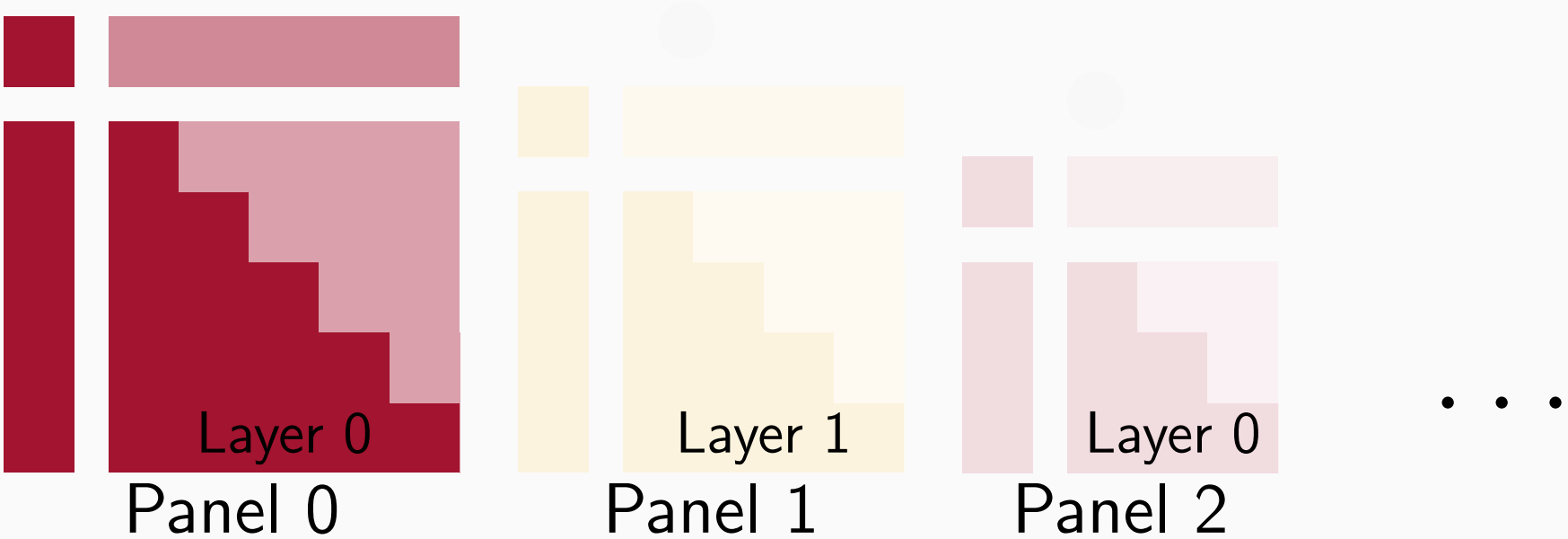


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

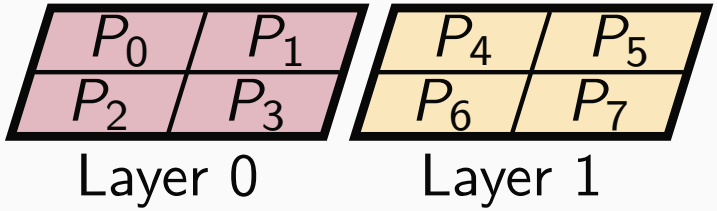


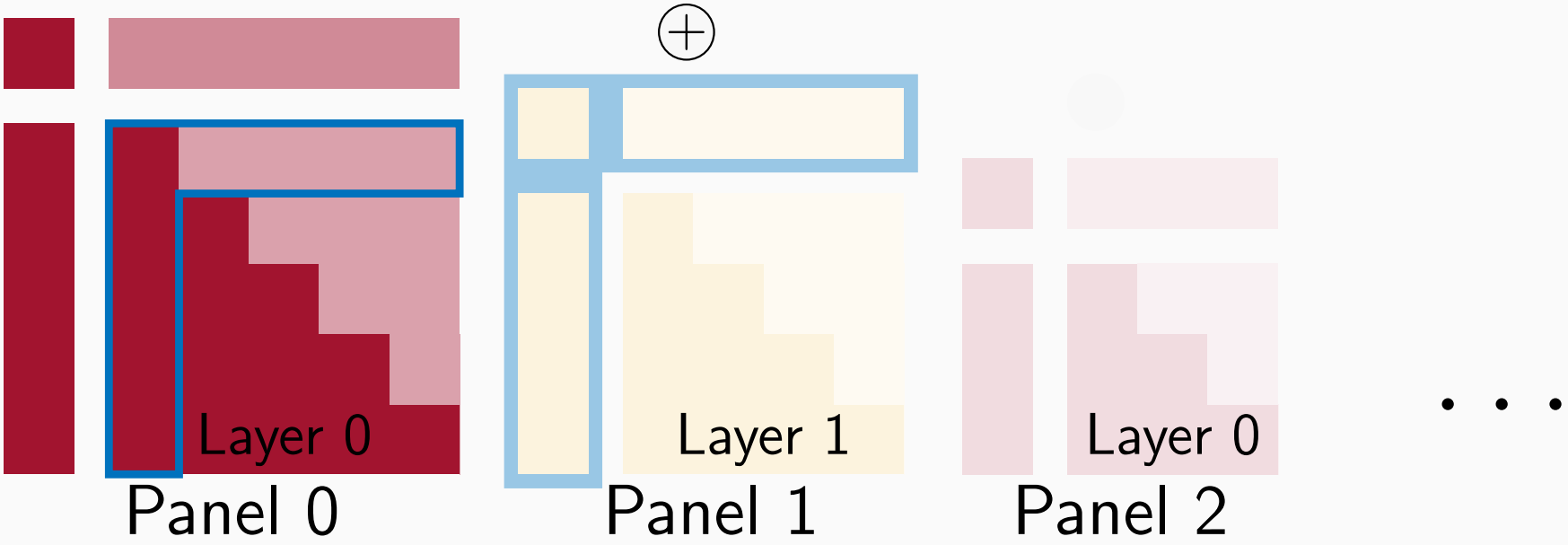


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

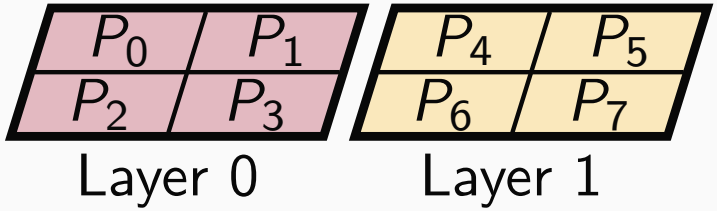


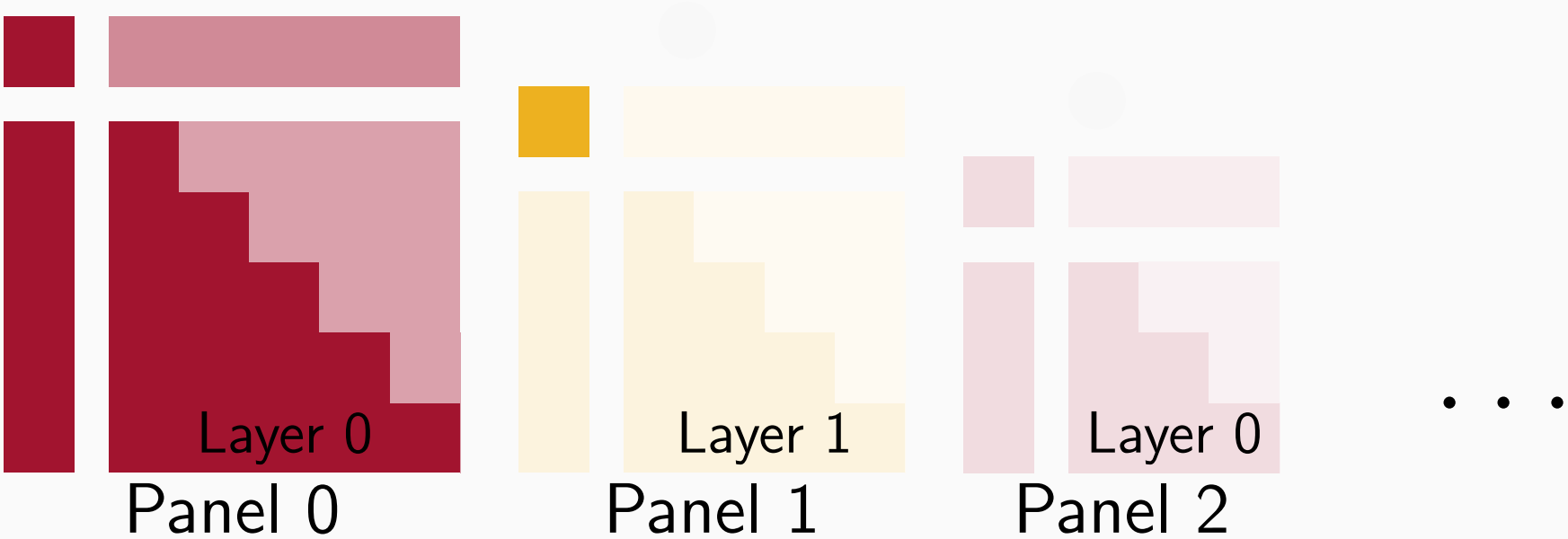


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

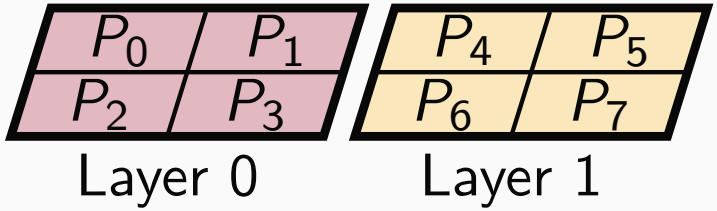


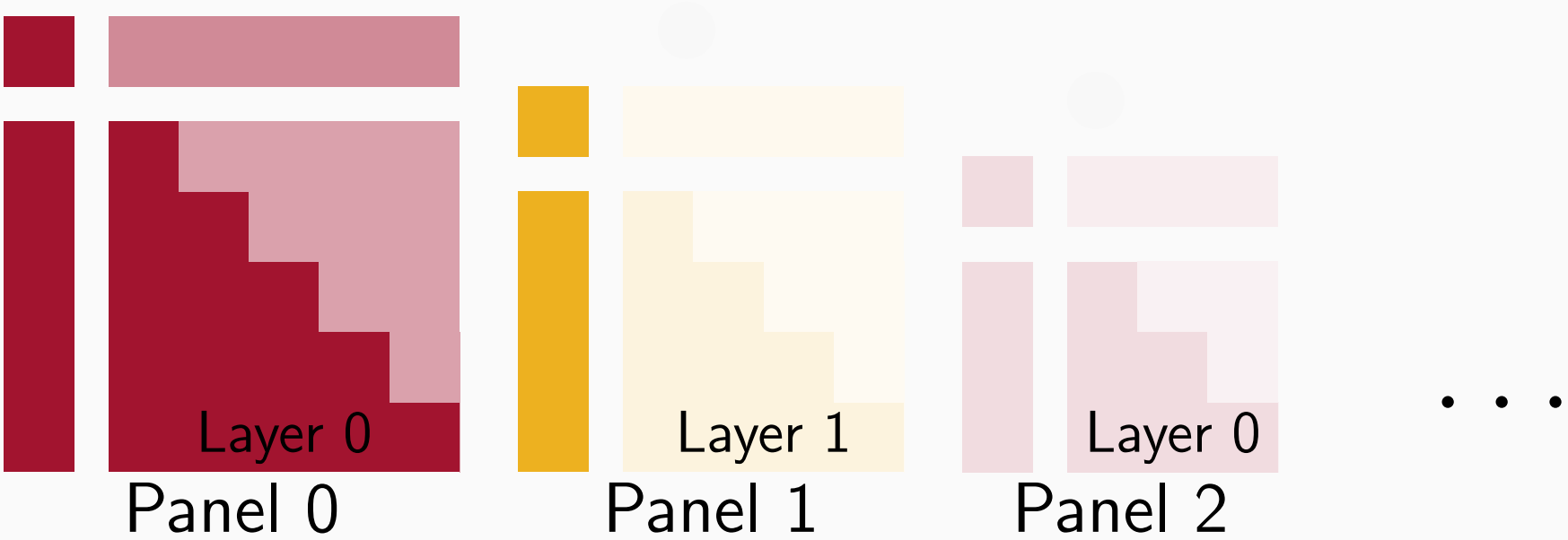


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

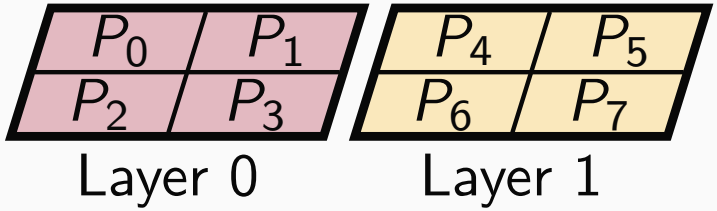


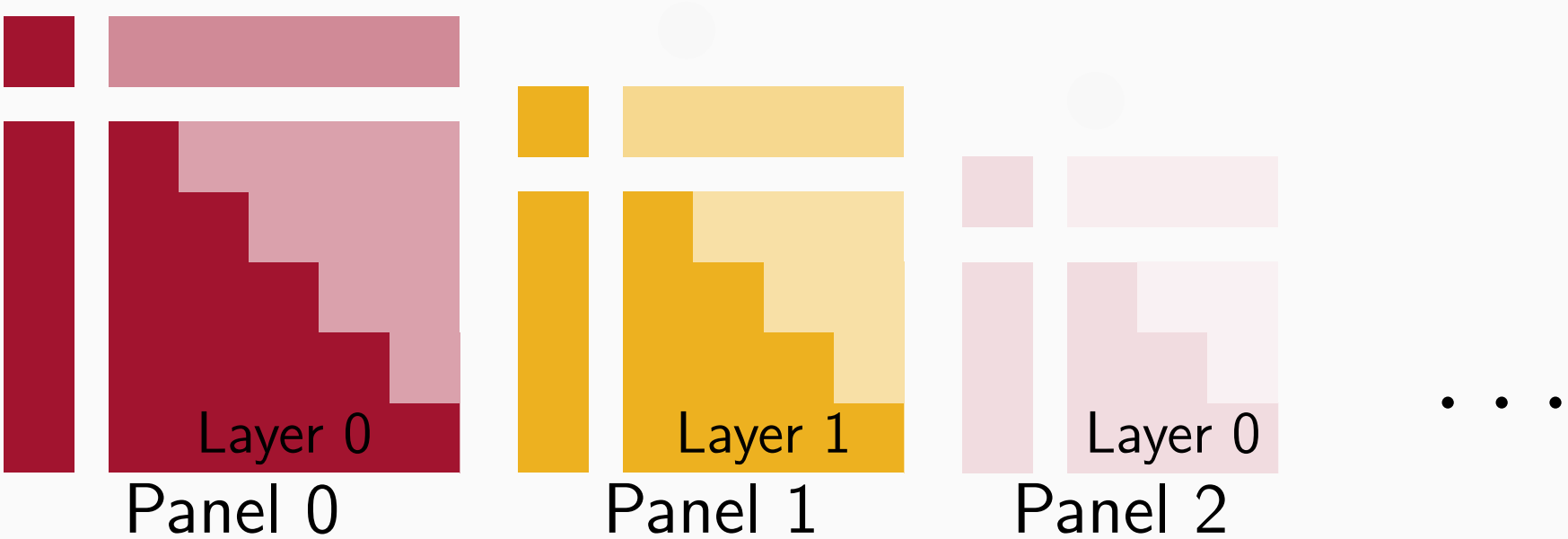


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

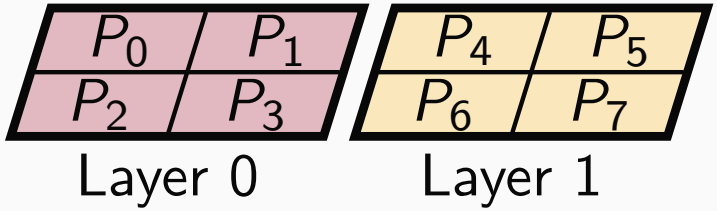


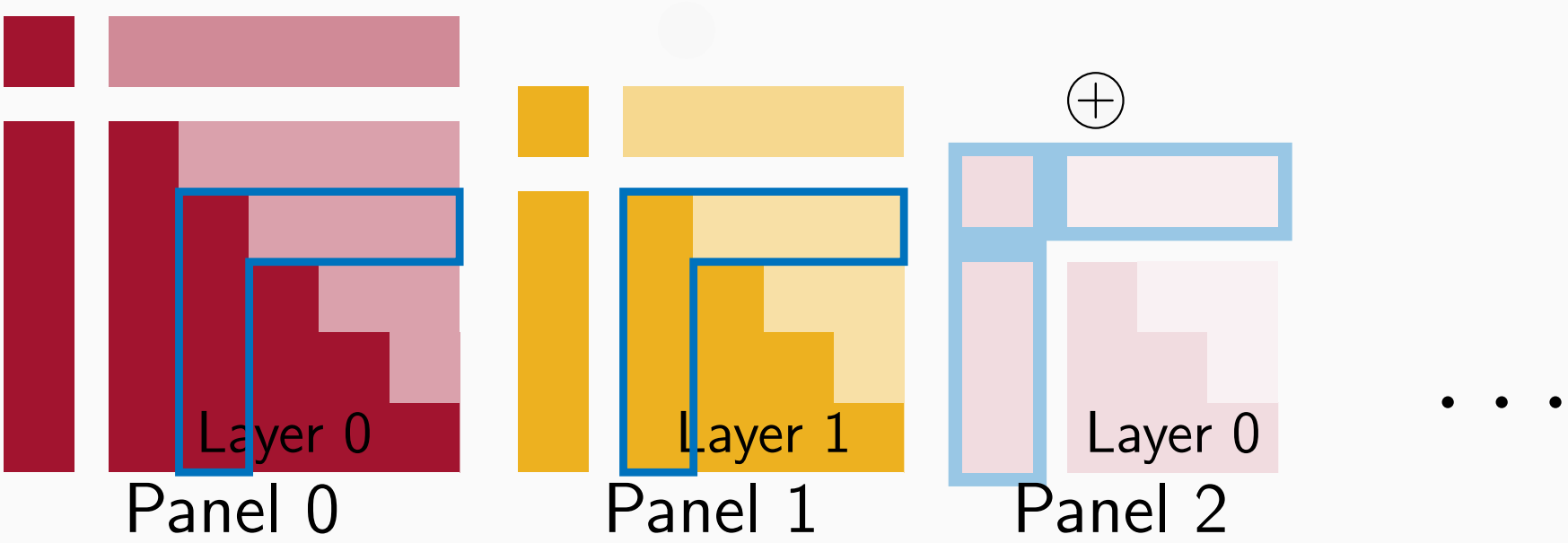


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

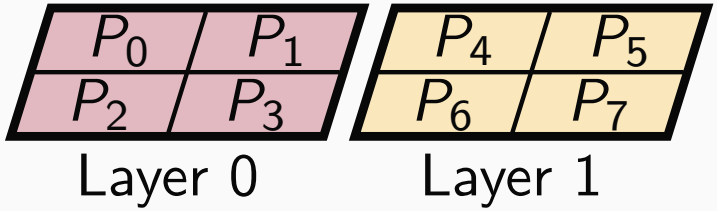


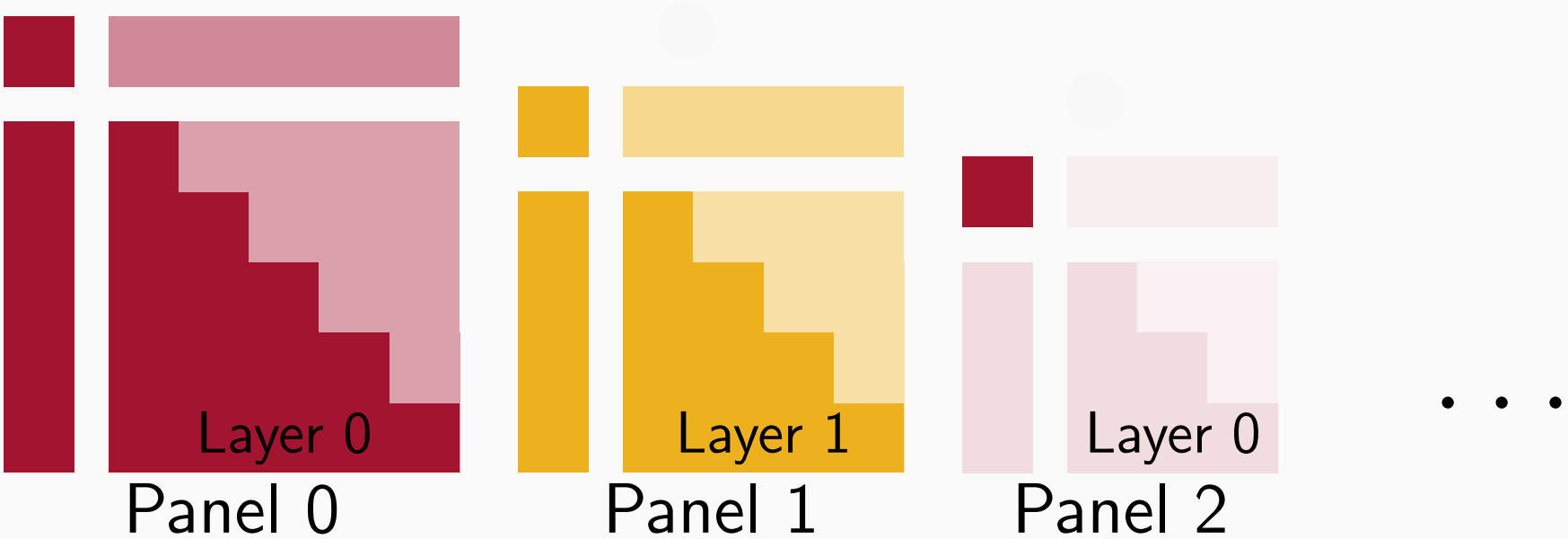


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

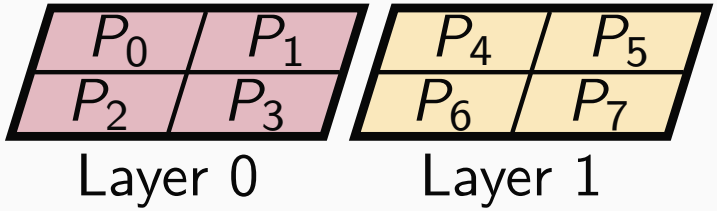


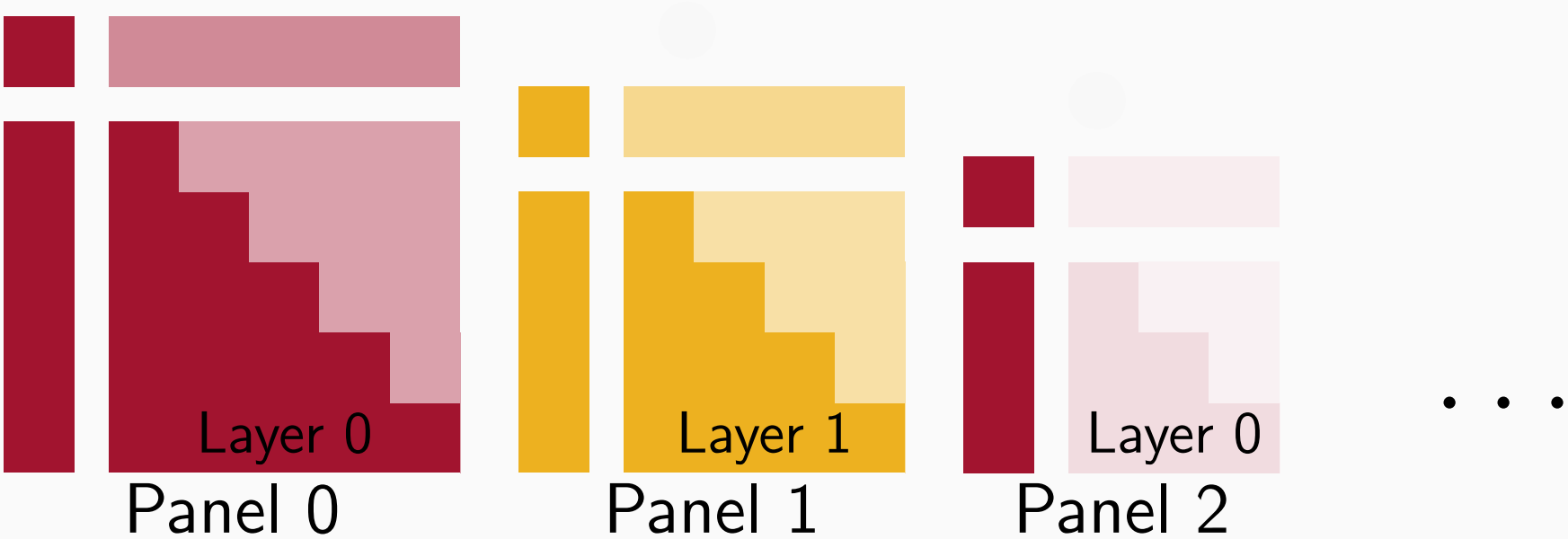


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

3D computation → processes virtually arranged in a 3D grid

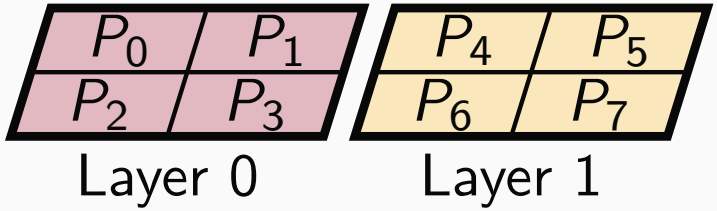


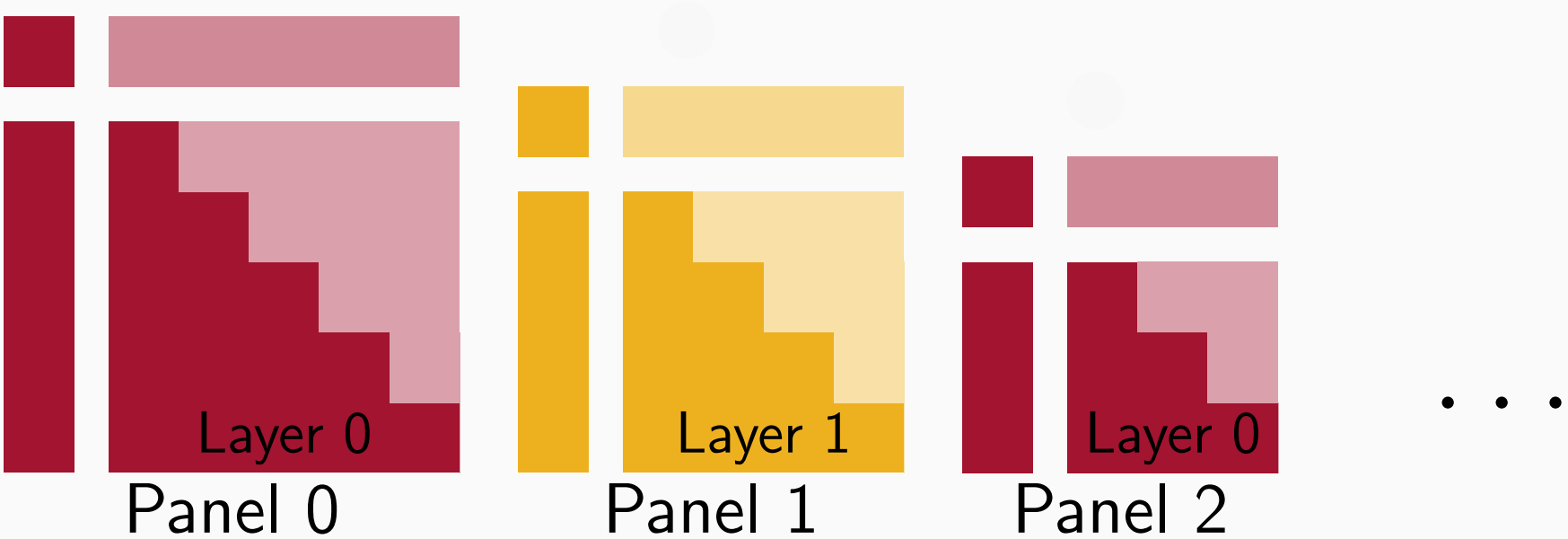


Distributed-memory Cholesky factorization

```
do l=1,m ! for each panel
  call insert_task(potrf, A_ll:RW)
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm,
        A_ij:RANK_REDUX, A_il:R,A_jl:R,
        ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

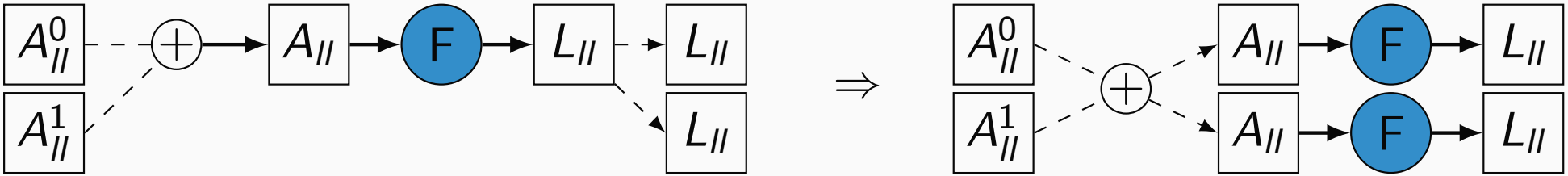
3D computation → processes virtually arranged in a 3D grid





Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.

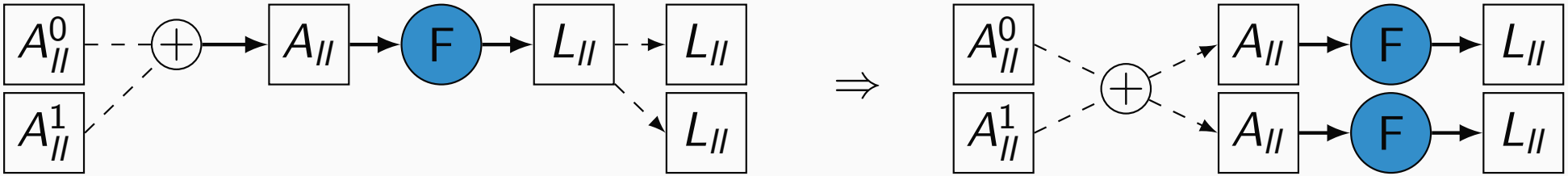


The factorization must be executed over **several layers**.

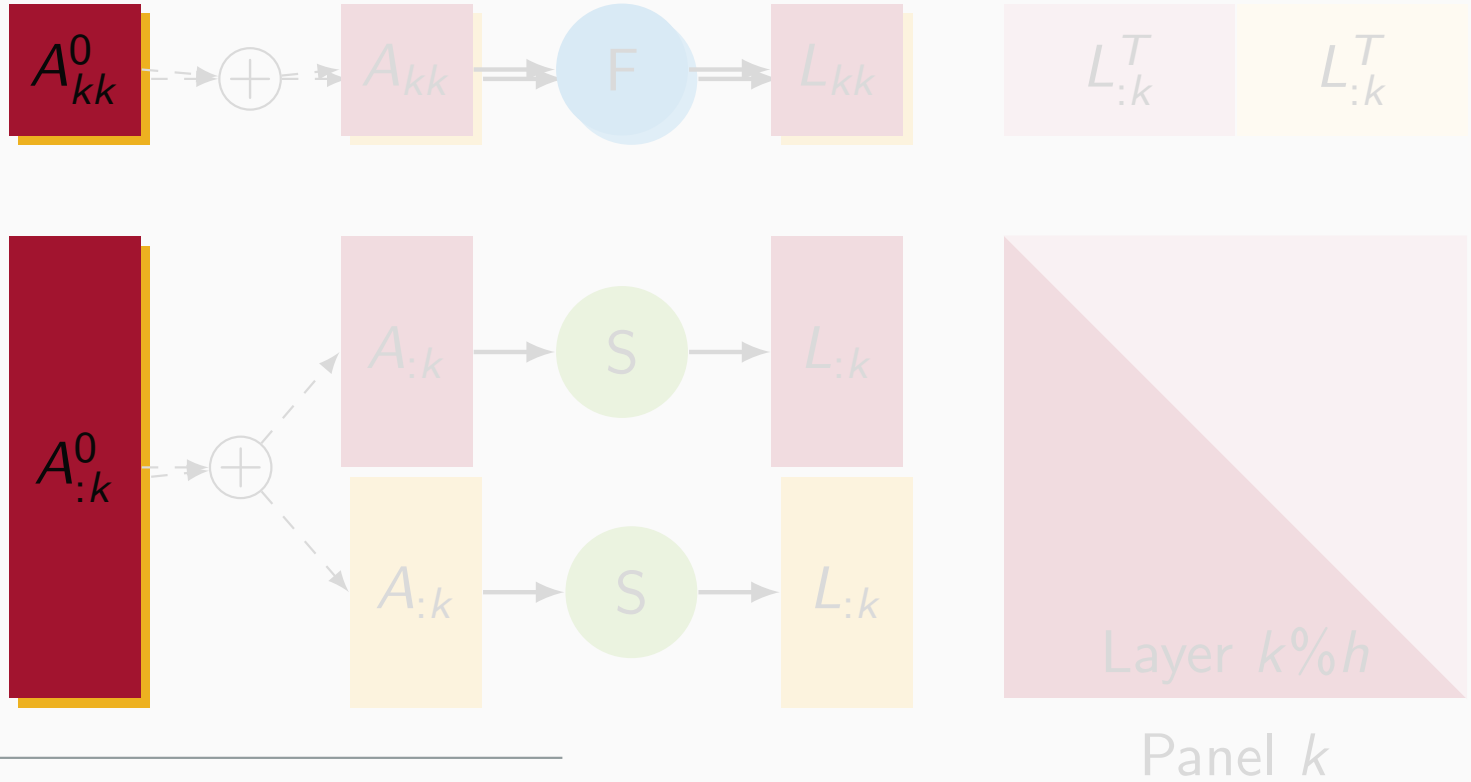
1. Solomonik et Demmel, 2011

Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.



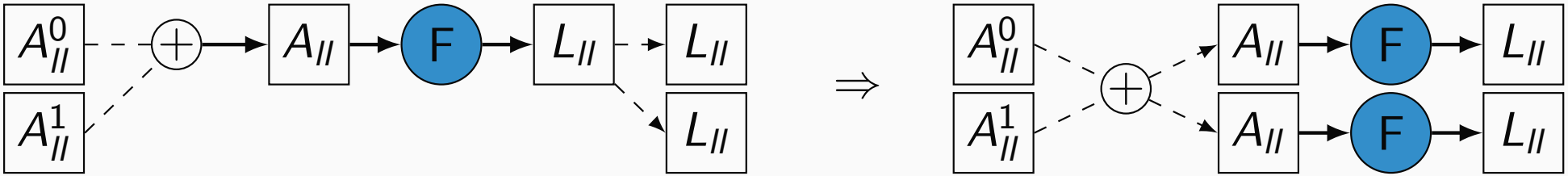
The factorization must be executed over **several layers**.



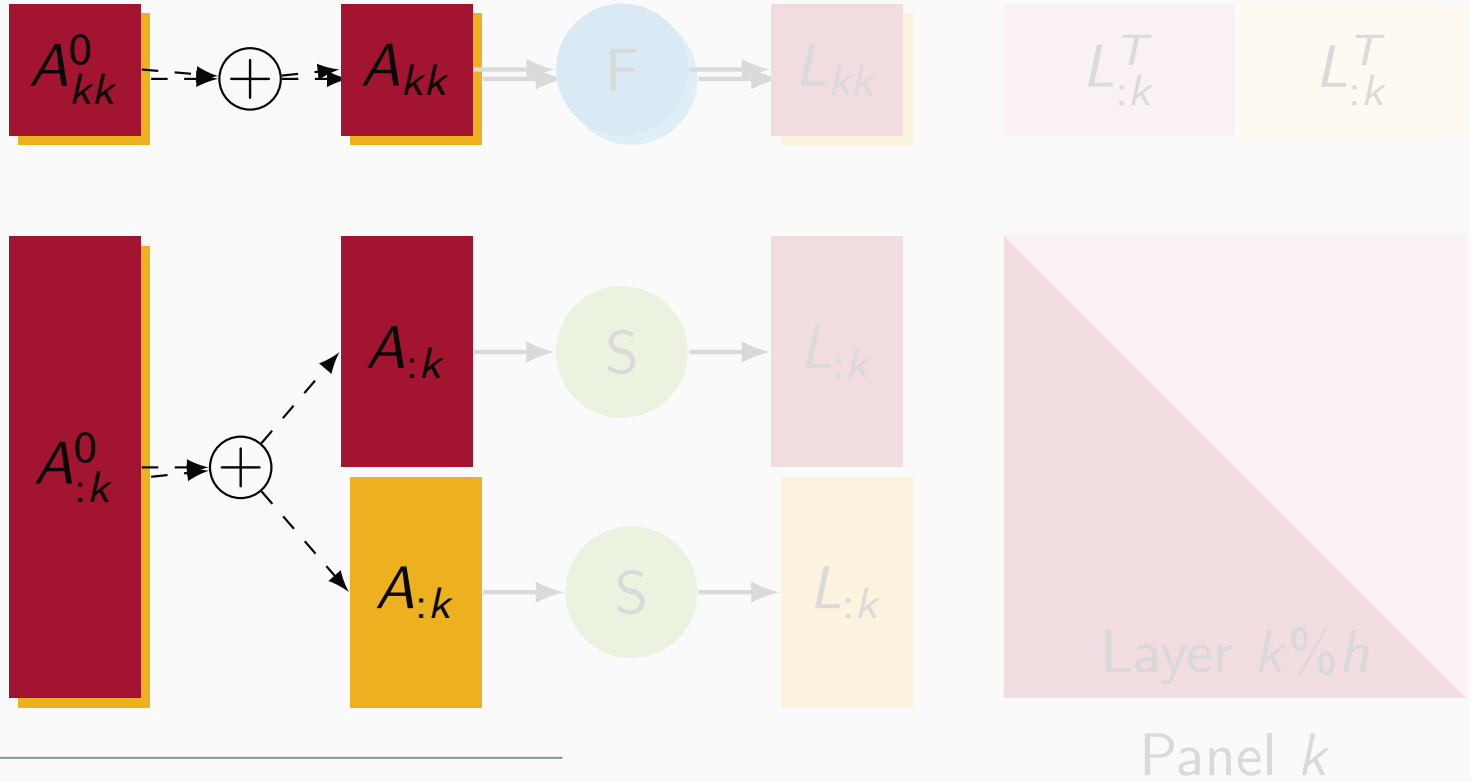
1. Solomonik et Demmel, 2011

Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.



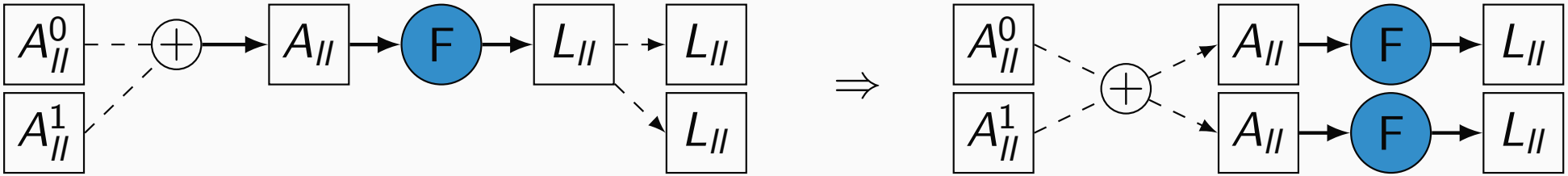
The factorization must be executed over **several layers**.



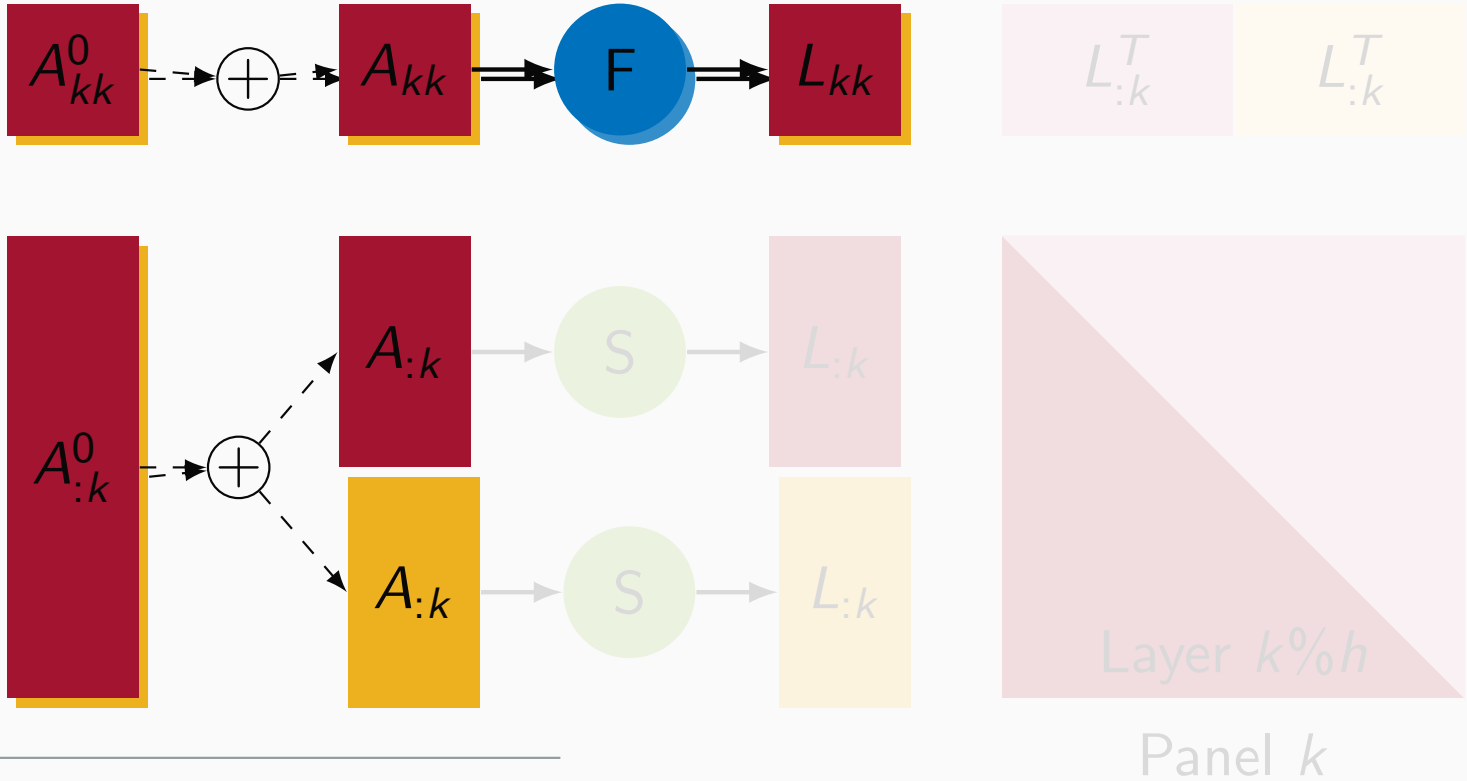
1. Solomonik et Demmel, 2011

Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.



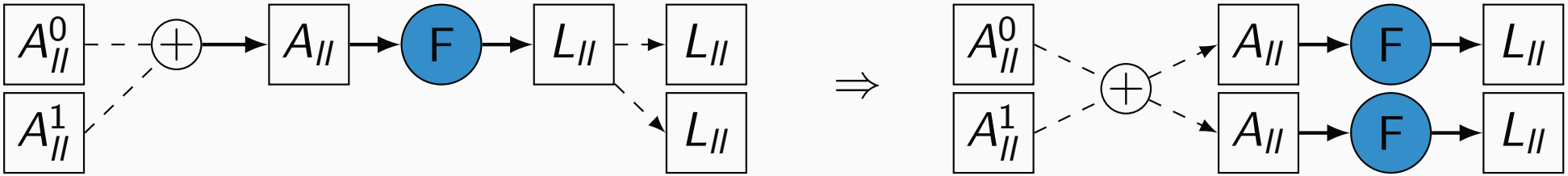
The factorization must be executed over **several layers**.



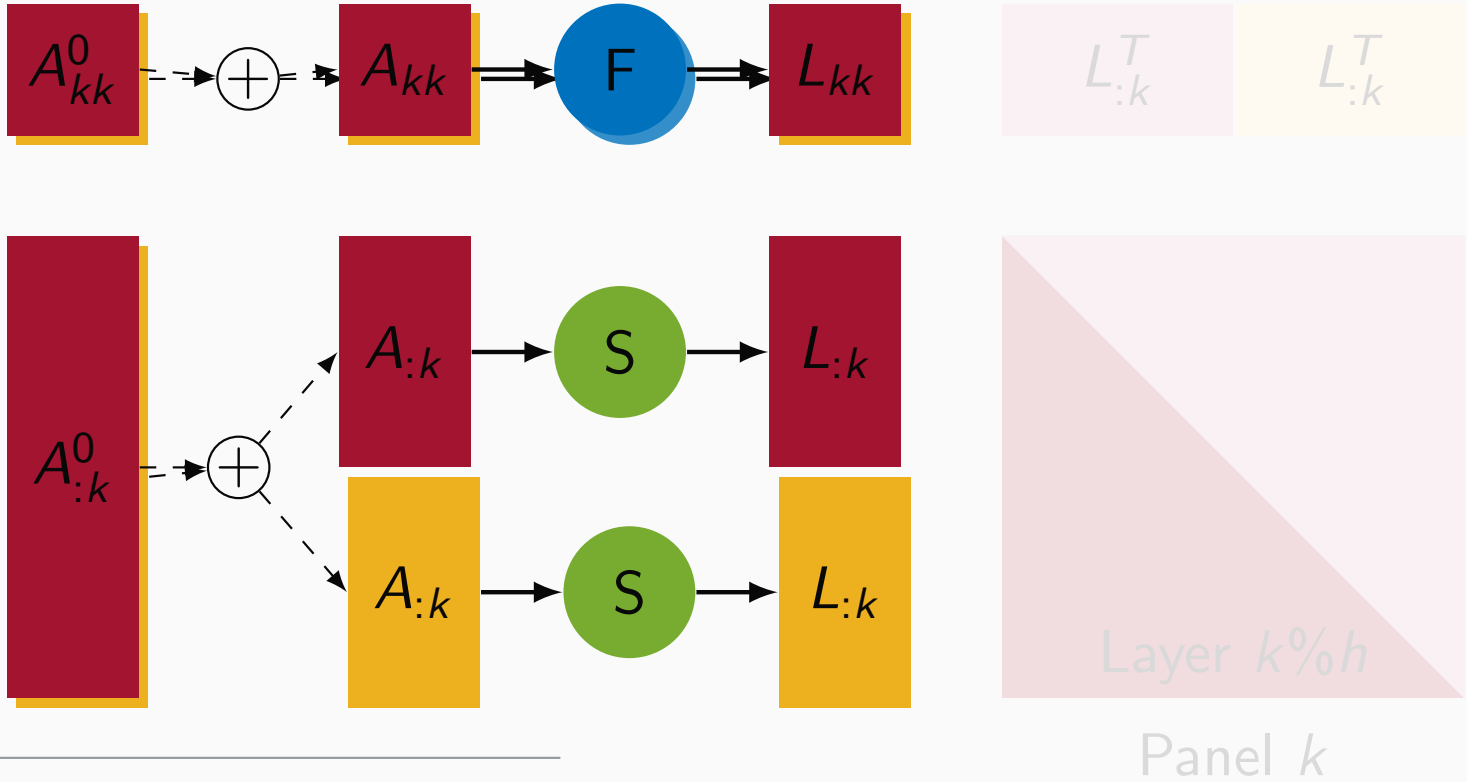
1. Solomonik et Demmel, 2011

Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.



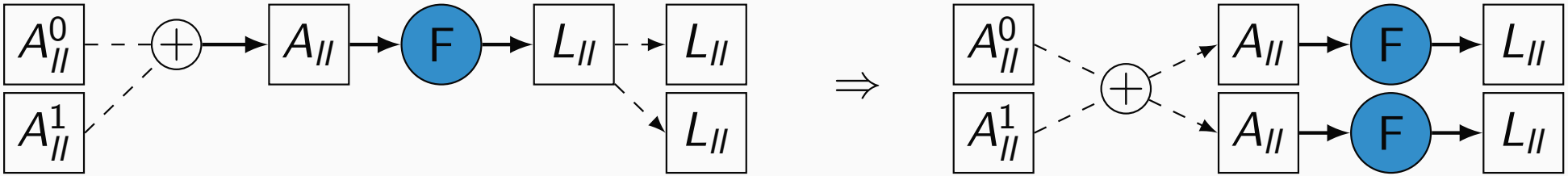
The factorization must be executed over **several layers**.



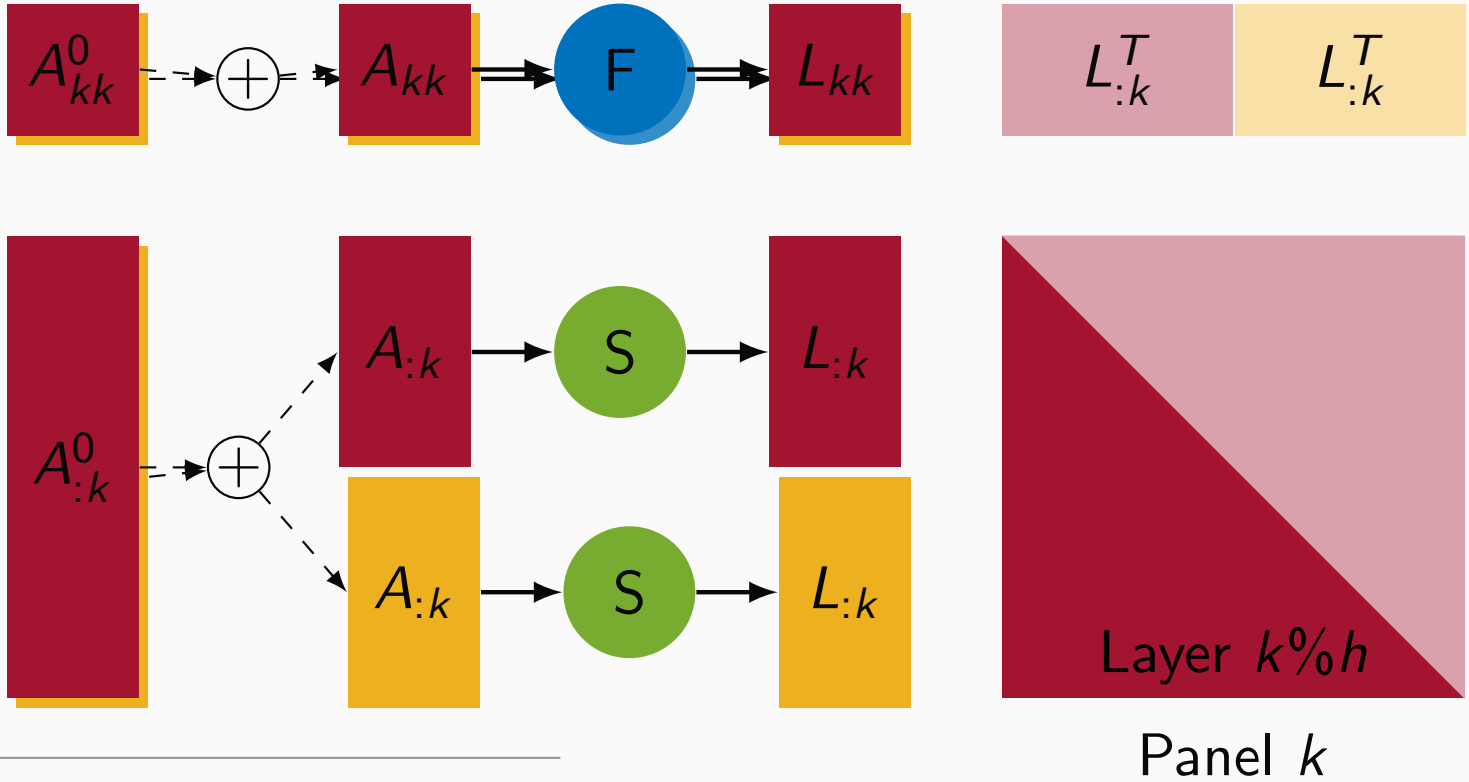
1. Solomonik et Demmel, 2011

Distributed-memory Cholesky factorization

Key idea [1] **:** replicate computation → ↘ cost of communications.



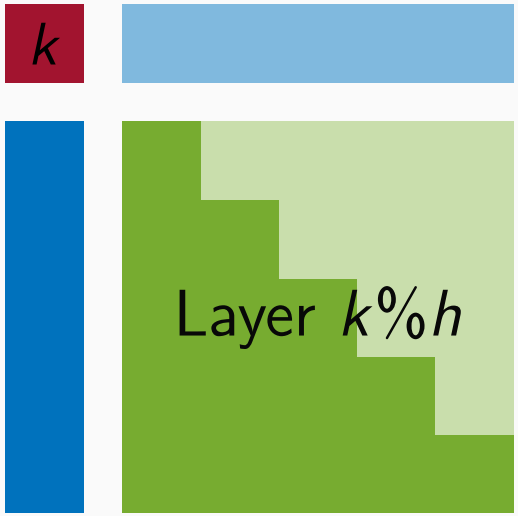
The factorization must be executed over **several layers**.



1. Solomonik et Demmel, 2011

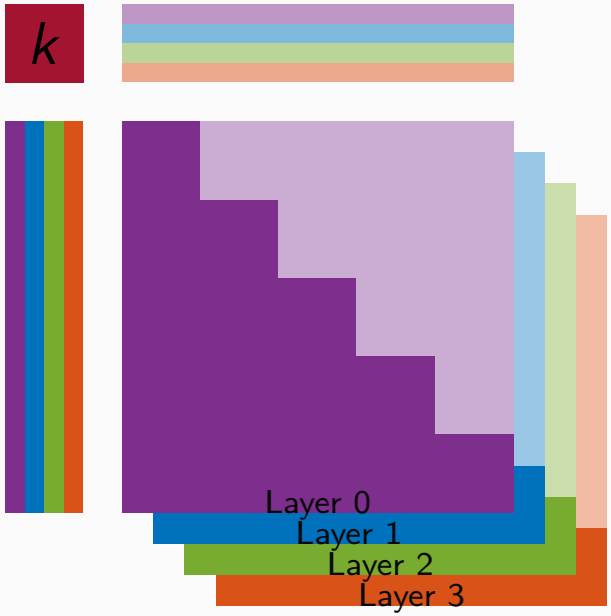
Distributed-memory Cholesky factorization

Usual approach



One panel $\iff$ One dense block update

Partitioning approach [1]



One panel $\iff$ Several partial block updates

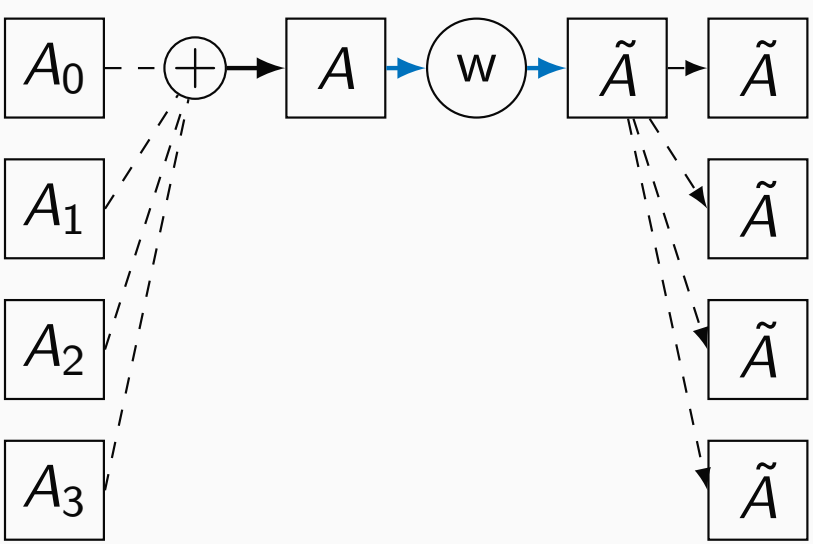
1. Kwasniewski, Kabic, Ben-Nun et al., 2021

MOTIVATION

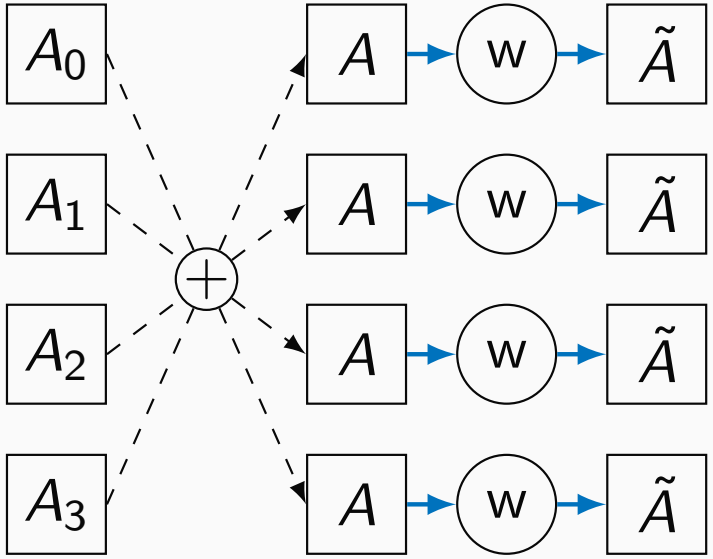
Abstract core computation

The following computation is used in the previous algorithms

- Several nodes hold a contribution A_i to a result.
- The assembled result A is used to compute an output $\tilde{A}$.
- The output is required across all nodes.



$2P$ messages, $2\log(P)$ steps

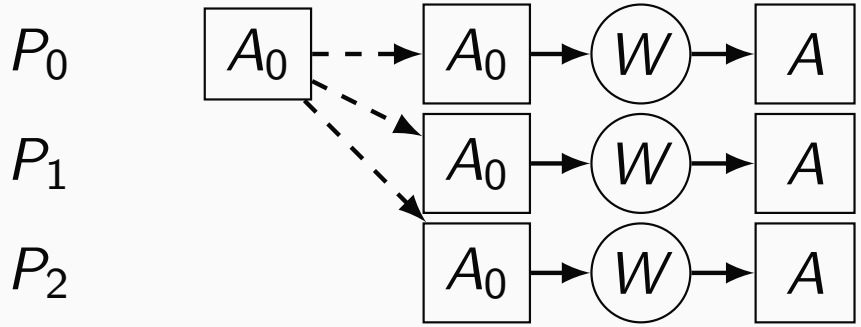


$P\log(P)$ messages, $\log(P)$ steps

TASK REPLICATION IN THE STF MODEL

Replicating tasks

Simple example



- P_0 initializes A
- each process replicates a computation W over A

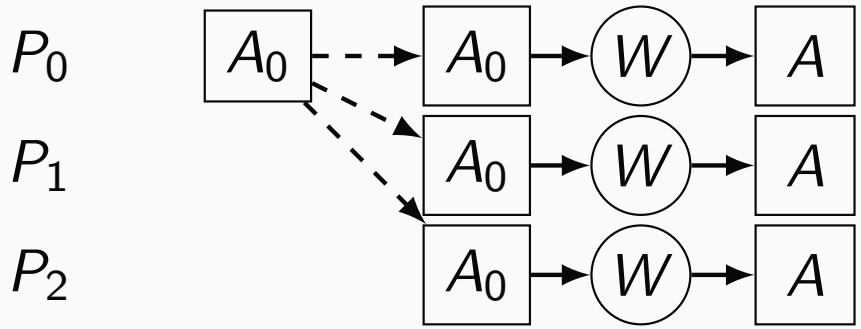
In a STF code, we could create one output data per replicating local memory.

```
data_register(hdl, (me == 0) ? A : NULL)
do node = 1, NODES
  data_register(hdl_replicated[node], (me == node) ? A_repl[me] : NULL)
end do
do node = 1, NODES
  insert_task( work_replicated, hdl:R, hdl_replicated[node]:RW, ON:node)
end do
insert_task( work, hdl:RW, ON:0)
```

Creating handles to manage memory is tedious.

Replicating tasks

Simple example



- P_0 initializes A
- each process replicates a computation W over A

In a STF code, we could create one output data per replicating local memory.

```
data_register(hdl, (me == 0) ? A : NULL)
do node = 1, NODES
  data_register(hdl_replicated[node], (me == node) ? A_repl[me] : NULL)
end do
do node = 1, NODES
  insert_task( work_replicated, hdl:R, hdl_replicated[node]:RW, ON:node)
end do
insert_task( work, hdl:RW, ON:0)
```

Creating handles to manage memory is tedious.

Replicating tasks - New feature

We provide a function (`insert_tasks`) to insert the same tasks over several nodes.

```
data_register(hdl, (me == 0) ? A : NULL)
insert_tasks(work, hdl:RW, ON: [0..NODES])
```

Under the hood, we add **a flag** (`SAME`) to enable replication.

```
data_register(hdl, (me == 0) ? A : NULL)
do node = 1, NODES
  insert_task(work, hdl:RW & SAME, ON:node)
end do
insert_task(work, hdl:RW & SAME, ON:0)
```

Sequential consistency requires that the task of the replicants are inserted before the task of the owner.

Replicating tasks - New feature

We provide a function (`insert_tasks`) to insert the same tasks over several nodes.

```
data_register(hdl, (me == 0) ? A : NULL)
insert_tasks(work, hdl:RW, ON: [0..NODES])
```

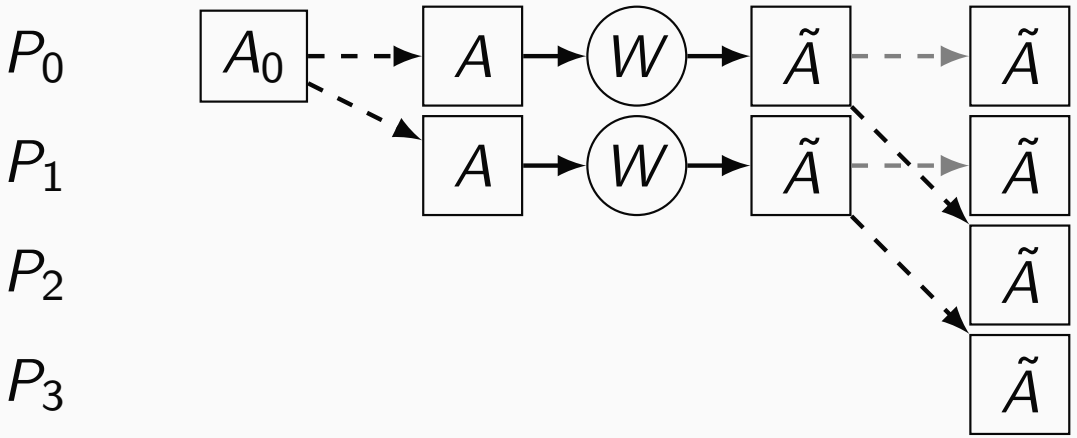
Under the hood, we add **a flag** (`SAME`) to enable replication.

```
data_register(hdl, (me == 0) ? A : NULL)
do node = 1, NODES
  insert_task(work, hdl:RW & SAME, ON:node)
end do
insert_task(work, hdl:RW & SAME, ON:0)
```

Sequential consistency requires that the task of the replicants are inserted before the task of the owner.

Replicating tasks - what for ?

To benefit from replication, replicated results should be used.

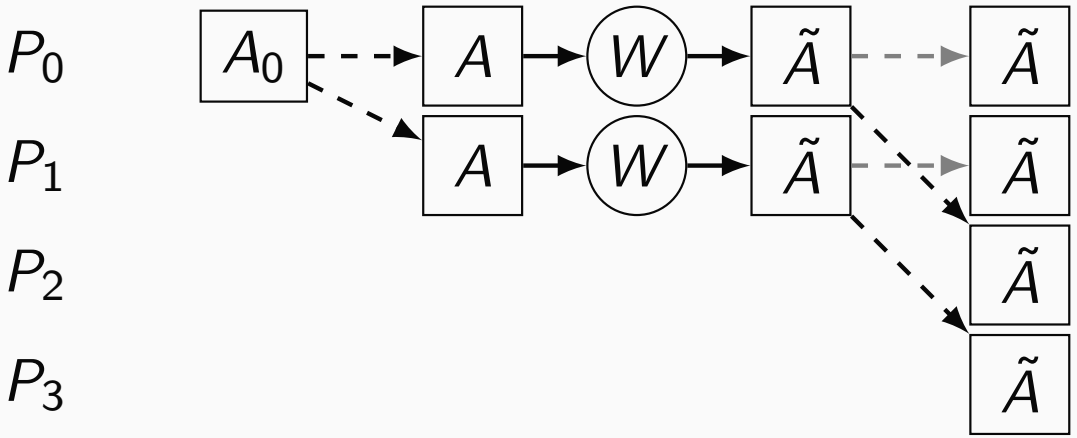


We set up an alternative source for a handle through the `set_source` function.

```
data_register(hdl, (me == 0) ? A : NULL)
insert_tasks(work, hdl:RW, ON: [0,1])
do node = 2, NODES
  set_source(hdl, node, node % 2)
  insert_task(additional_work, hdl:R, ON: node)
end do
```

Replicating tasks - what for ?

To benefit from replication, replicated results should be used.



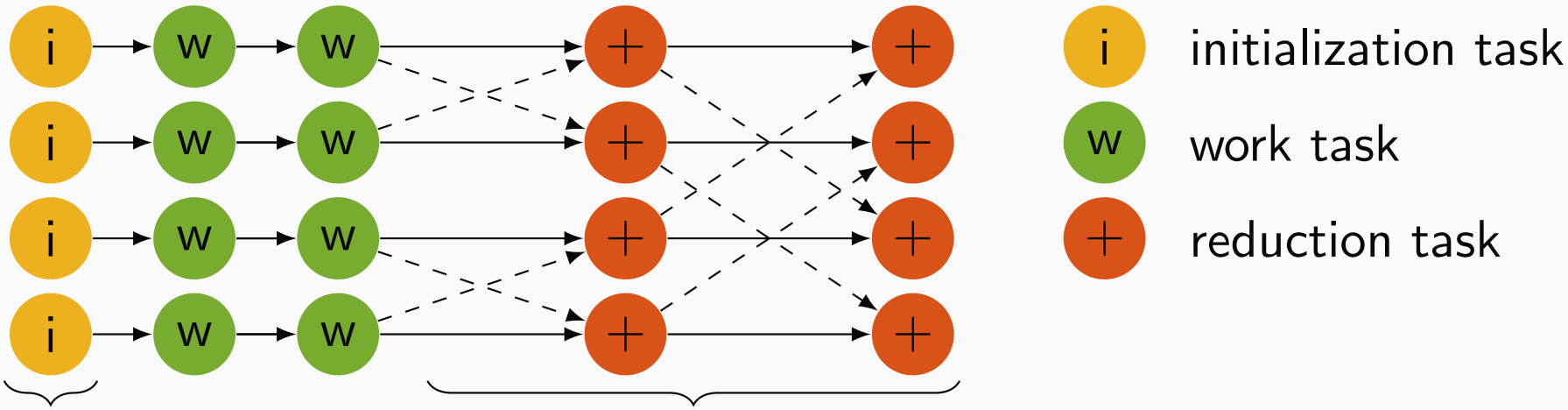
We set up an alternative source for a handle through the `set_source` function.

```
data_register(hdl, (me == 0) ? A : NULL)
insert_tasks(work, hdl:RW, ON: [0,1])
do node = 2, NODES
  set_source(hdl, node, node % 2)
  insert_task(additional_work, hdl:R, ON: node)
end do
```

ALL-REDUCE IN THE STF MODEL

All-reduce pattern

- Essentially the same as the simple reduction pattern
- *Recursive doubling* is unrolled

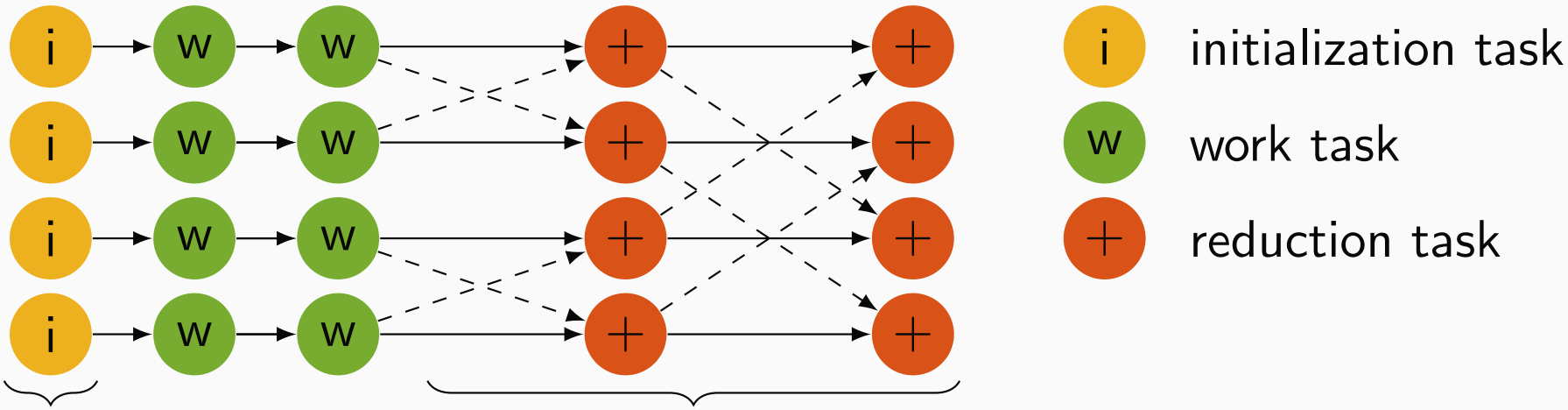


Implicit – handled by the runtime

- Communicators do not need to be setup
- Runtime can progress multiple patterns with available resources

All-reduce pattern

- Essentially the same as the simple reduction pattern
- *Recursive doubling* is unrolled



Implicit – handled by the runtime

- Communicators do not need to be setup
- Runtime can progress multiple patterns with available resources

IMPLEMENTATION

Replicating tasks – Cholesky factorization

```
do l=1,m
  call insert_tasks(potrf, A_ll:RW, ON:[A_ll%owner + i*NODES,i=0..h-1])
  do layer=0,h-1; do node=1,NODES
      call set_source(A_ll, node + layer*p*q, A_ll%owner + layer*NODES )
  end do; end do
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm, A_ij:RANK_REDUX, A_il:R,A_jl:R,ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

1. `insert_tasks` is used to replicated diagonal factorization
2. `set_source` is used to ensure nodes receive the factorized diagonal block from the correct replicant
3. `RANK_REDUX` is used to leverage reduction patterns

Note : It is possible to apply the partitioning idea in this code. Simple, static partitioning exists within classical implementations of the STF model.

Replicating tasks – Cholesky factorization

```
do l=1,m
  call insert_tasks(potrf, A_ll:RW, ON:[A_ll%owner + i*NODES,i=0..h-1])
  do layer=0,h-1; do node=1,NODES
      call set_source(A_ll, node + layer*p*q, A_ll%owner + layer*NODES )
  end do; end do
  do i=l,m
    call insert_task(trsm, A_il:RW, A_ll:R)
    call insert_task(syrk, A_ii:RW, A_il:R)
    do j=i+1,m
      call insert_task(gemm, A_ij:RANK_REDUX, A_il:R,A_jl:R,ON:A_ij%owner + NODES*l%h)
    end do
  end do
end do
```

1. `insert_tasks` is used to replicated diagonal factorization
2. `set_source` is used to ensure nodes receive the factorized diagonal block from the correct replicant
3. `RANK_REDUX` is used to leverage reduction patterns

Note : It is possible to apply the partitioning idea in this code. Simple, static partitioning exists within classical implementations of the STF model.

You can use it yourself !

Task replication has been implemented inside StarPU, in separate branches.

The API to replicate tasks is available in MR 66.

- `STARPU_SAME`
- `starpu_mpi_set_source`
- `starpu_mpi_tasks_insert` (on-going work)

All-reduce is available in MR 68.

- `starpu_mpi_allredux`

All-reduce patterns are not detected dynamically.

EXPERIMENTS

Experiments – Benchmark setup

- **Software** : GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model supporting replication + seq. BLAS
  - `ELemental` – OpenMPI 4.0.5, 1 MPI / core
    - Flame runtime + seq. BLAS
  - `slate` – OpenMPI 4.0.5, 2 cores / MPI
    - MPI + OpenMP with `task` + seq. BLAS
  - `COnfLUx` – OpenMPI 4.0.5, 1 MPI / core
    - MPI + OpenMP + seq. BLAS
- **Hardware** : Très Grand Centre de Calcul (TGCC)
  - **Irène** - **Skylake** Infiniband EDR, 4x links
    - Dual-processor Intel Skylake 8168 @ 2.7 GHz
    - 24 cores per processor, 192 GB DDR4 memory

Experiments – Benchmark setup

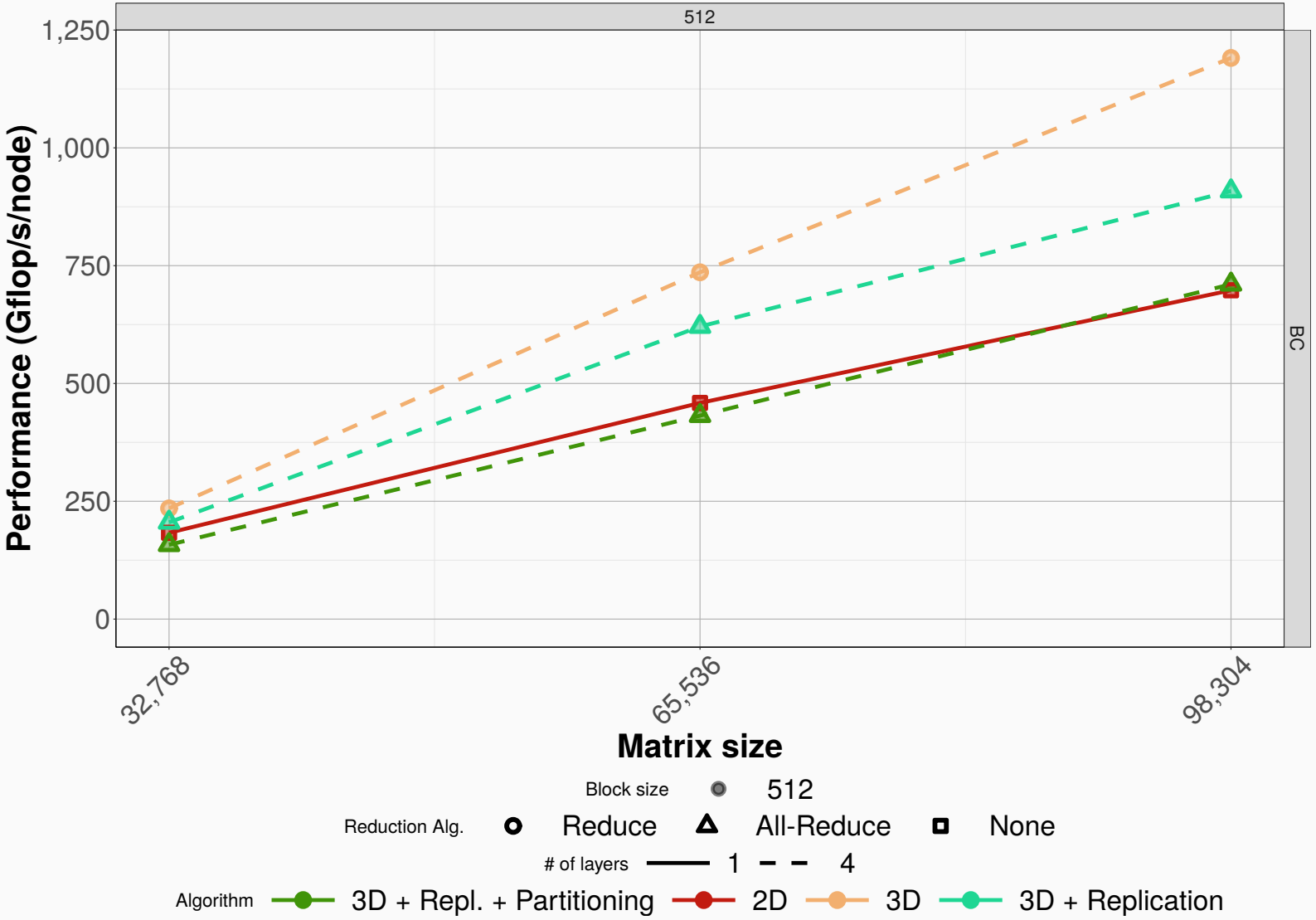
- **Software** : GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model supporting replication + seq. BLAS
  - `ELemental` – OpenMPI 4.0.5, 1 MPI / core
    - Flame runtime + seq. BLAS
  - `slate` – OpenMPI 4.0.5, 2 cores / MPI
    - MPI + OpenMP with `task` + seq. BLAS
  - `COnfLUx` – OpenMPI 4.0.5, 1 MPI / core
    - MPI + OpenMP + seq. BLAS
- **Hardware** : Très Grand Centre de Calcul (TGCC)
  - **Irène** - **Skylake** Infiniband EDR, 4x links
    - Dual-processor Intel Skylake 8168 @ 2.7 GHz
    - 24 cores per processor, 192 GB DDR4 memory

Experiments – Benchmark setup

- **Software** : GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model supporting replication + seq. BLAS
  - `ELemental` – OpenMPI 4.0.5, 1 MPI / core
    - Flame runtime + seq. BLAS
  - `slate` – OpenMPI 4.0.5, 2 cores / MPI
    - MPI + OpenMP with `task` + seq. BLAS
  - `COnfLUx` – OpenMPI 4.0.5, 1 MPI / core
    - MPI + OpenMP + seq. BLAS
- **Hardware** : Très Grand Centre de Calcul (TGCC)
  - **Irène** - **Skylake** Infiniband EDR, 4x links
    - Dual-processor Intel Skylake 8168 @ 2.7 GHz
    - 24 cores per processor, 192 GB DDR4 memory

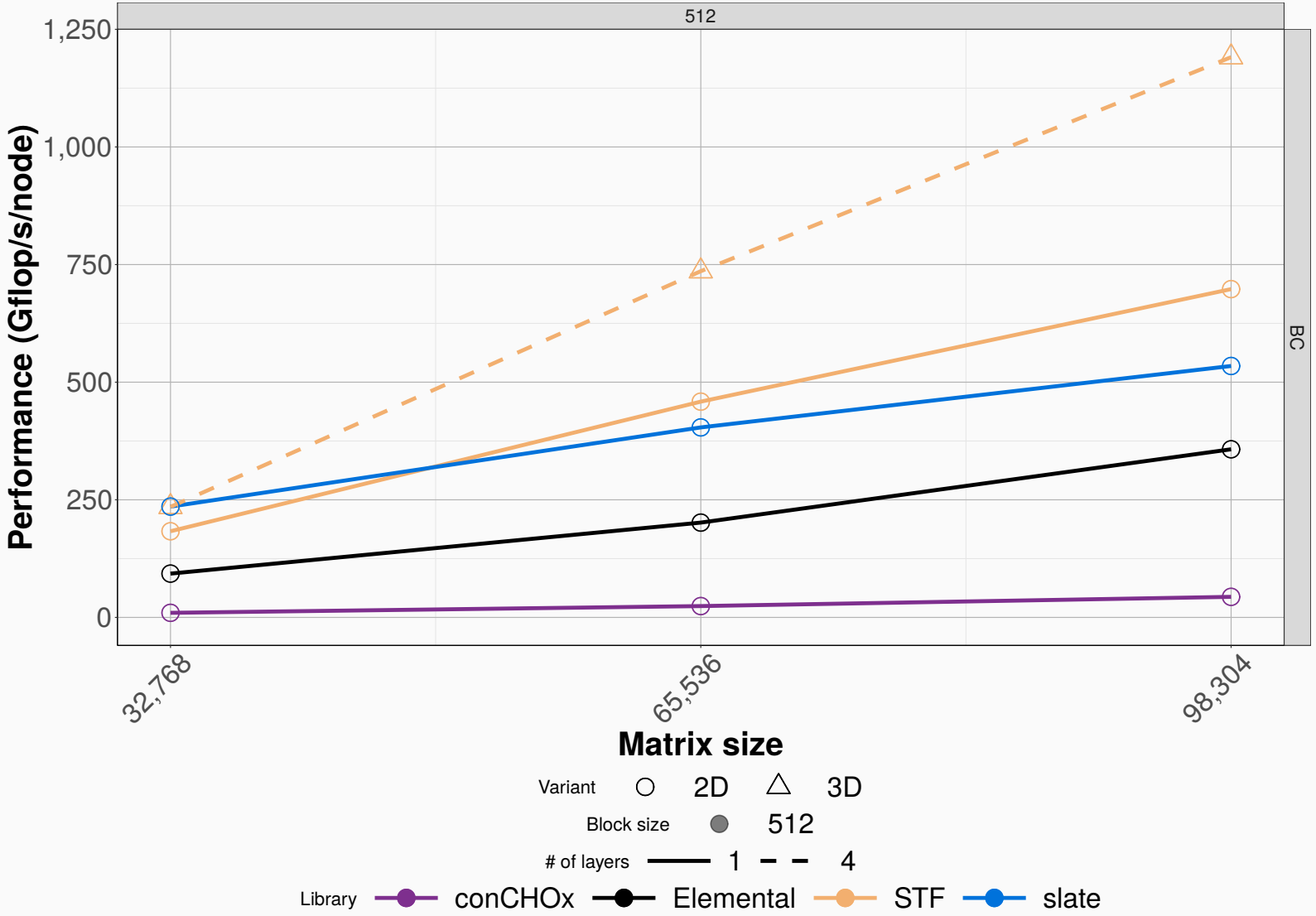
Results with Cholesky factorization – comparing variants

64 nodes



Results with Cholesky factorization – comparing libraries

64 nodes



More tuning might be required for the concurrent libraries

Results with Cholesky factorization – wrap-up

We observe that **replicating tasks yield not improvement.**

However 3D still improves $\Rightarrow$ Reducing communications is important

Some topics to explore

- Asynchronicity might make replication less useful
- Partitioning lowers tasks' granularity $\Rightarrow$ scheduling overhead ?

We would be delighted to exchange on these matters

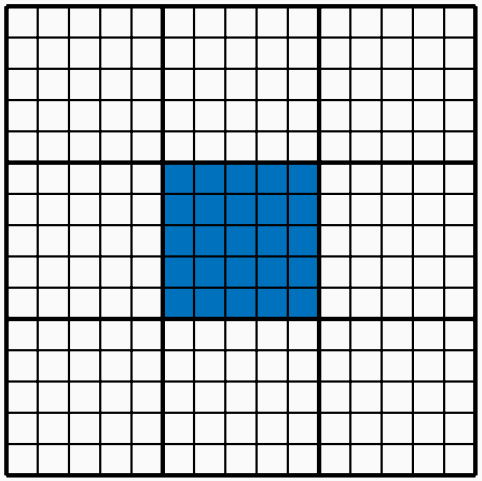
REPLICATING IN STENCIL SOLVERS

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

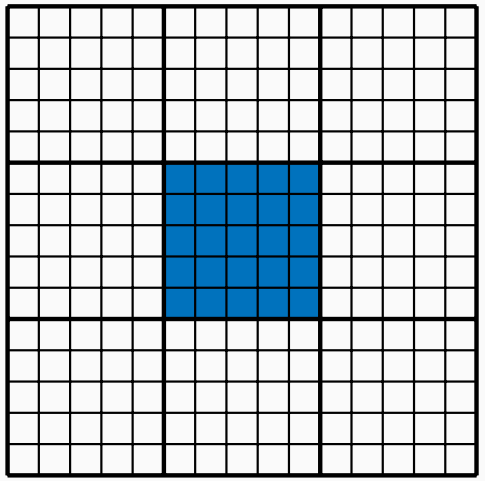
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

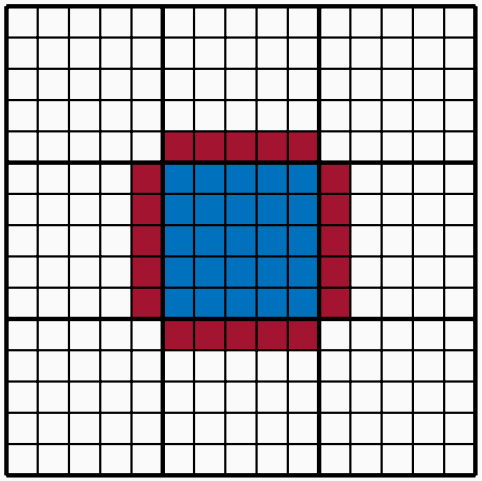
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

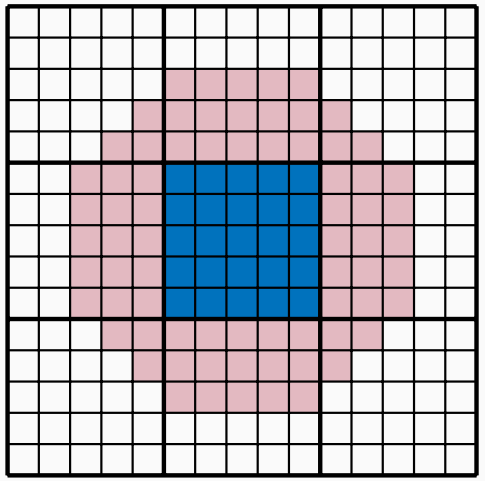
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

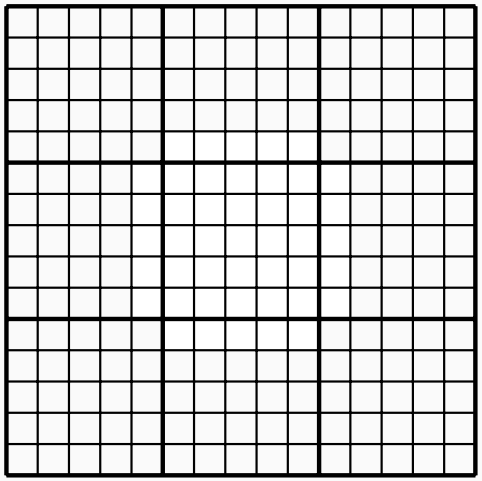
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

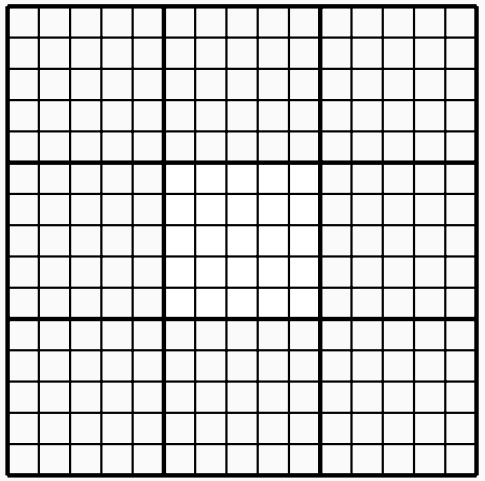
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

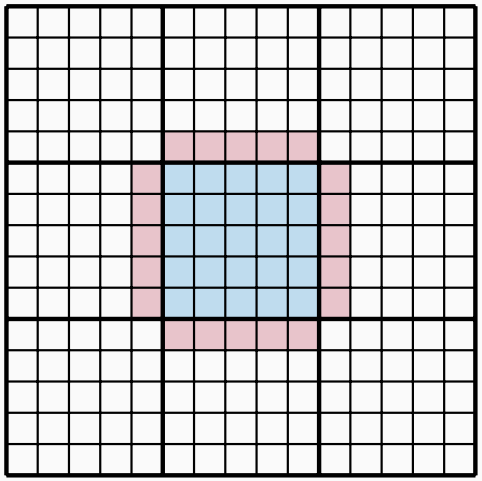
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

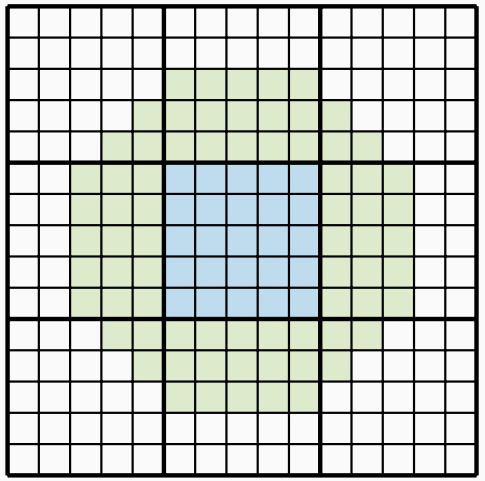
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

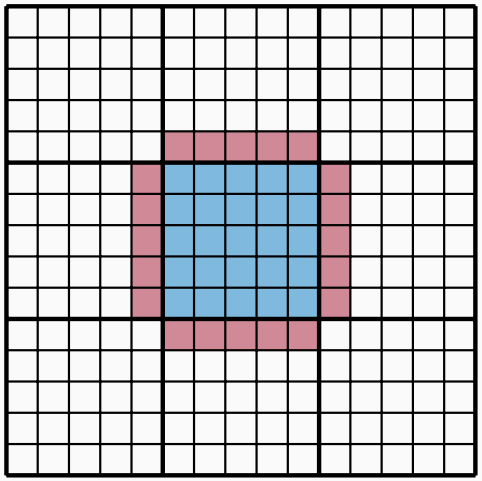
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

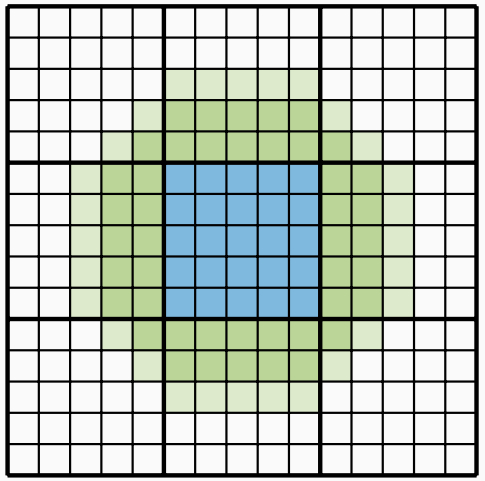
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

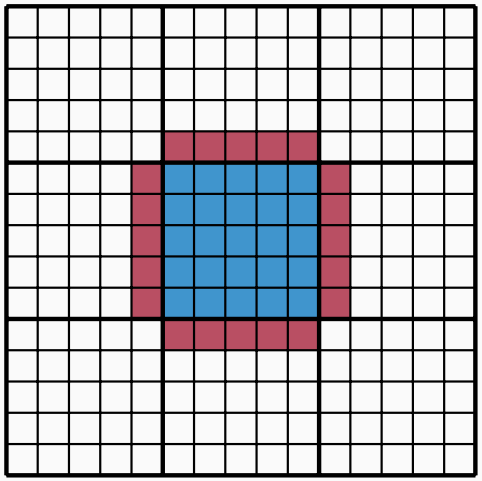
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

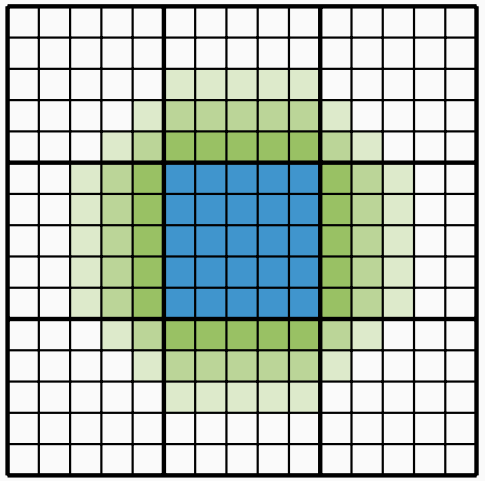
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

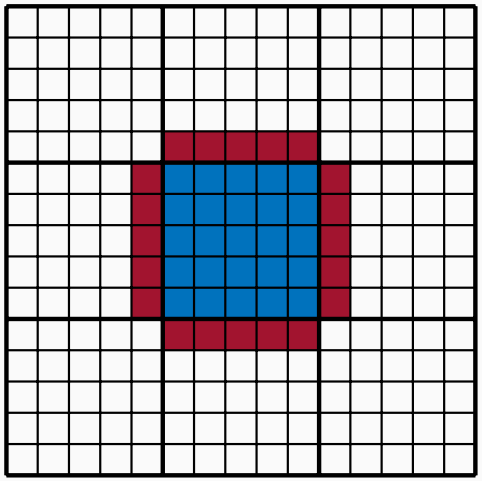
1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

Using replication

Stencil solvers can scale better with replication [1].
The core computation is the **matrix-powers** kernel.

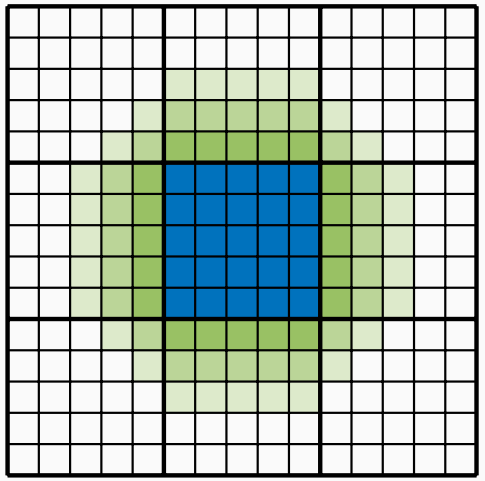
$$V = [x, Ax, \ldots, A^k x]$$

Classical



Latency sensitive

w/ replication mechanism



Bandwidth sensitive

Note : the same mechanism is applicable to *s-steps* iterative solvers.

1. Pei, Cao, Bosilca, Luszczek, Eijkhout et Dongarra, 2020

CONCLUSIONS

Contributions

- Support task replication in a general runtime system to implement scientific computing routines
- Showcase applicability of "Communication-Avoiding" approaches in an asynchronous setting

Future work

- Write my manuscript :-)
- Find a post-doc