TASK-BASED PROGRAMMING MODELS FOR SCALABLE ALGORITHMS

Application to matrix multiplication

Antoine Jego

November 6, 2022

Quick presentation

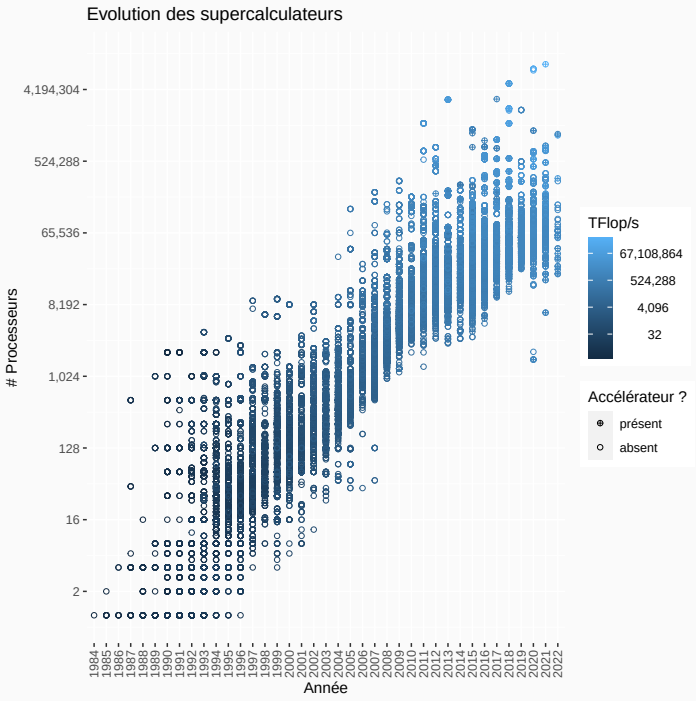
- 2019-2020: Finishing my engineering degree in ENSEEIHT ("HPC & Big Data")
  - My PFE was done in ENSEEIHT laboratory to quickstart the Phd Thesis

- 2020-2022 : 2 years as a PhD student
  - Collaborating with researchers from Bordeaux, Lyon, Toulouse as part of the SOLHARIS ANR project.

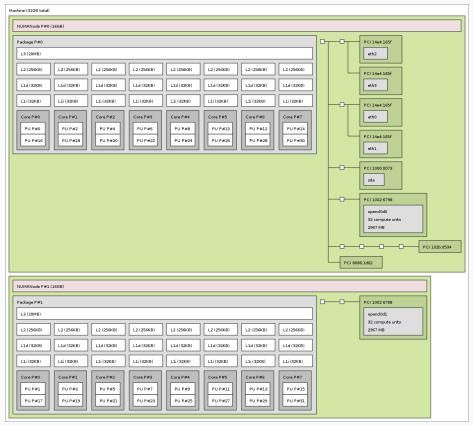
INTRODUCTION

Algorithms, architectures and programming models

Architectures are becoming increasingly **large**, **heterogeneous** and **complex**.



- Architecture
  - Single core 60s
  - Vectorized 70s
  - Multinode 90s
  - Multicore 00s
  - Accelerated 10s
- Software
  - Fortran '55
  - C '72
  - MPI '92
  - OpenMP '97
  - CUDA '07



A typical node architecture, with two multiple cores CPUs and two GPUs

Algorithms, architectures and programming models

Architectures are becoming increasingly **large**, **heterogeneous** and **complex**.

- Algorithms have to handle communications to achieve high scalability
  - Communication/computation overlap
  - Communication-avoiding algorithms

- Software should deliver a reliable abstraction to build mathematical software
  - parallel programming models
  - runtime systems

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**.

- Algorithms have to handle communications to achieve high scalability
  - Communication/computation overlap
  - Communication-avoiding algorithms

- Software should deliver a reliable abstraction to build mathematical software
  - parallel programming models
  - runtime systems

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**.

- Algorithms have to handle communications to achieve high scalability
  - Communication/computation overlap
  - Communication-avoiding algorithms
- Software should deliver a reliable abstraction to build mathematical software
  - parallel programming models
  - runtime systems

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**.

- Algorithms have to handle communications to achieve high scalability
  - Communication/computation overlap
  - Communication-avoiding algorithms
- Software should deliver a reliable abstraction to build mathematical software
  - parallel programming models
  - runtime systems

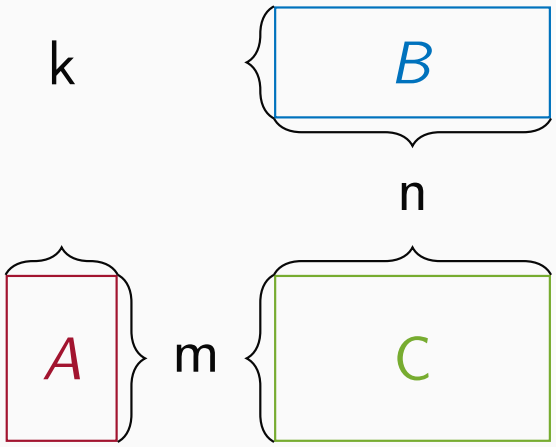
Are currently available parallel programming models and runtimes expressive enough to implement scalable algorithms in an efficient, portable and maintainable way?

Algorithms, architectures and programming models

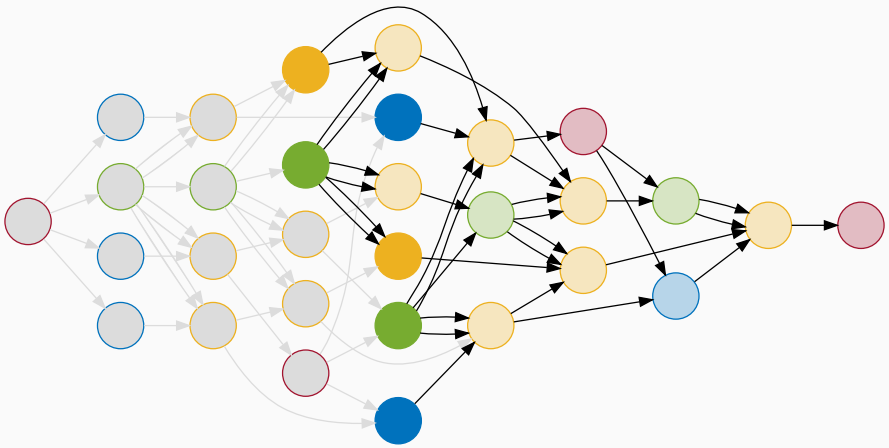
We take the following examples to answer this large question

- 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



BACKGROUND

Generalities

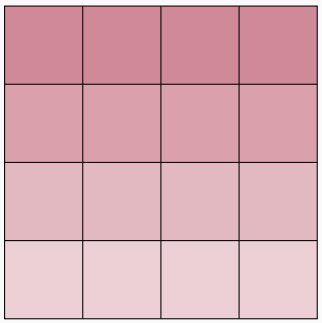
We focus on $C = AB$. Matrices are partitioned by blocks.

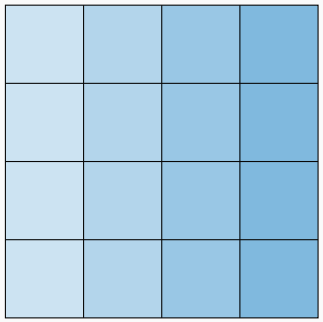
For each block $C_{i,j}$ of C, we compute $C_{i,j} = \sum_{l=1}^{k} A_{i,l} B_{l,j}$.
Each contribution $A_{i,l} B_{l,j}$ is identified as $T_{i,j,l}$ in a tensor T.

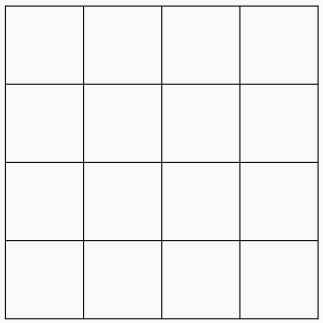
Algorithms for GEMM have to

- state an initial data distribution
- state what process handles $T_{i,j,l}$
- state how blocks are exchanged
- state a final distribution

Distributed-memory GEMM Algorithms







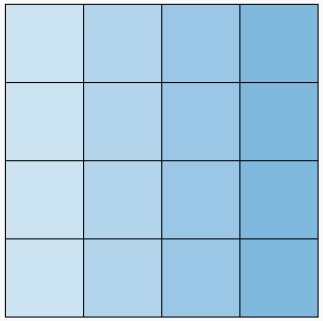
Cannon algorithm

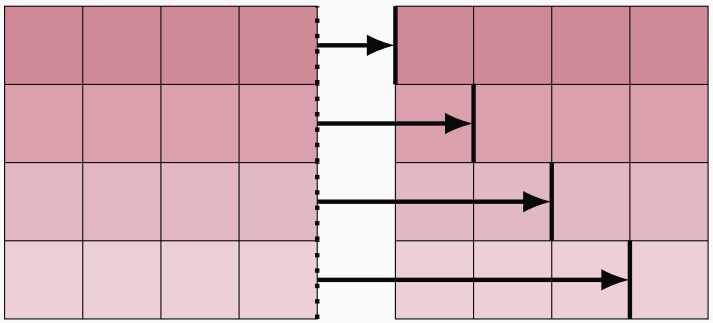
- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C computes
- **Communications**
  - initial row-wise skew of A
  - initial column-wise skew of B
  - row-wise circular shift of A
  - column-wise circular shift of B

```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms





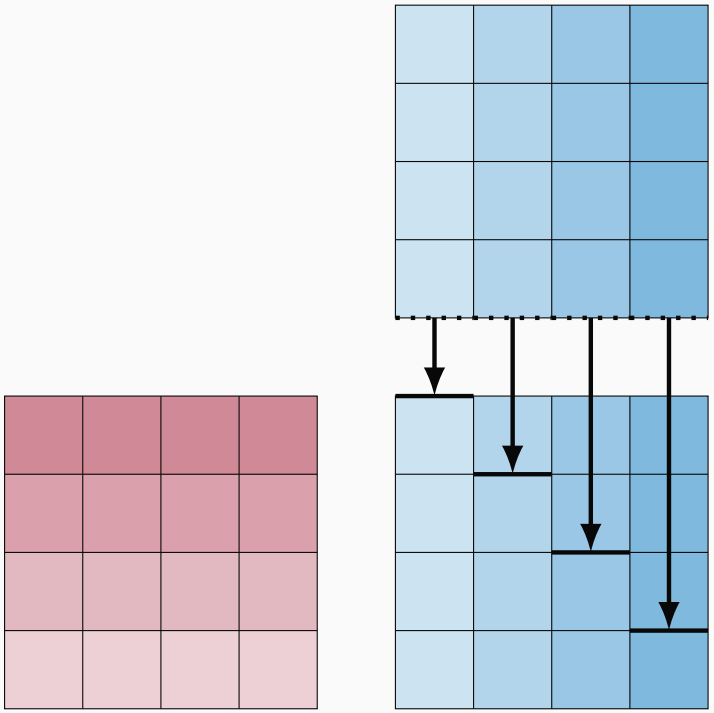
Cannon algorithm

- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C computes
- **Communications**
  - initial row-wise **skew** of A
  - initial column-wise skew of B
  - row-wise circular shift of A
  - column-wise circular shift of B

```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms



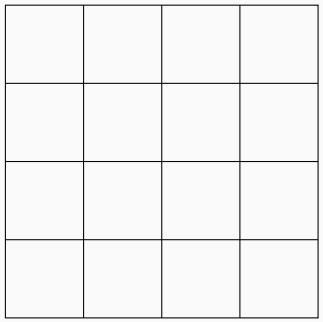
Cannon algorithm

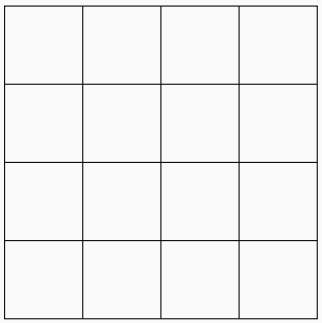
- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C computes
- **Communications**
  - initial row-wise skew of A
  - initial column-wise **skew** of B
  - row-wise circular shift of A
  - column-wise circular shift of B

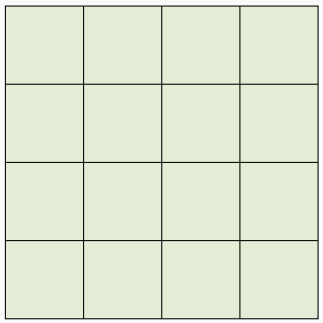
```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms







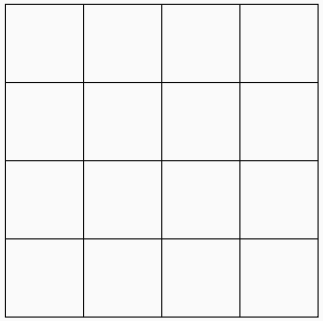
Cannon algorithm

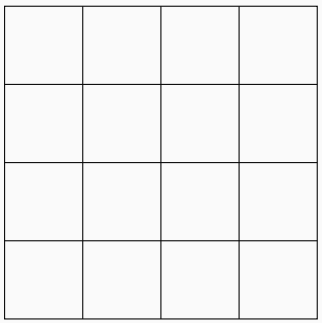
- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C **computes**
- **Communications**
  - initial row-wise skew of A
  - initial column-wise skew of B
  - row-wise circular shift of A
  - column-wise circular shift of B

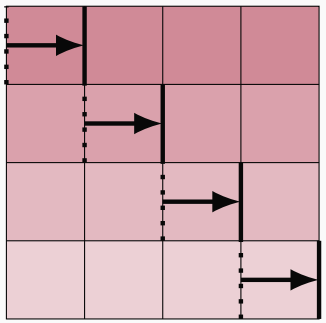
```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms







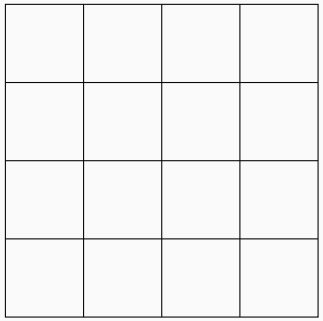
Cannon algorithm

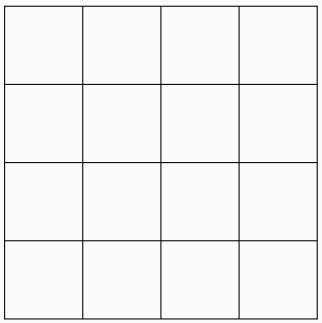
- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C computes
- **Communications**
  - initial row-wise skew of A
  - initial column-wise skew of B
  - row-wise circular **shift** of A
  - column-wise circular shift of B

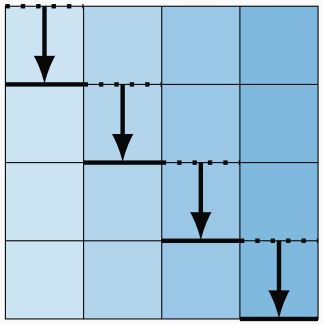
```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms







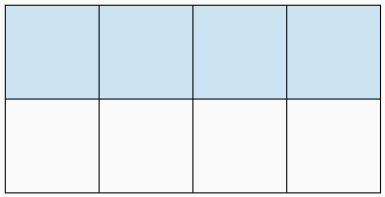
Cannon algorithm

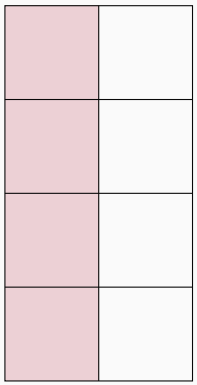
- **Data mapping**
  - 2D block
- **Workload mapping**
  - owner of C computes
- **Communications**
  - initial row-wise skew of A
  - initial column-wise skew of B
  - row-wise circular shift of A
  - column-wise circular **shift** of B

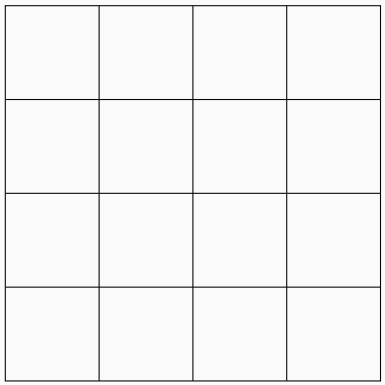
```
! I am r × c
send_recv(A, mod(r+c,n)×mod(r+c,n))
send_recv(B, mod(r+c,n)×mod(r+c,n))
do l=range(1,n).sort_first(mod(r+c),n)
   call gemm(A, B, C_{r,c})
   send_recv(A, mod(r+1,p)×c)
   send_recv(B, r×mod(c+1,p))
end do
```

Cannon, 1969

Distributed-memory GEMM Algorithms







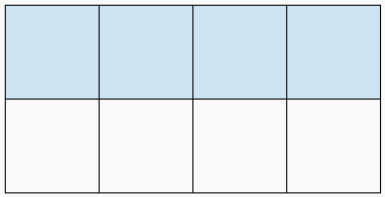
2D C-stationary SUMMA

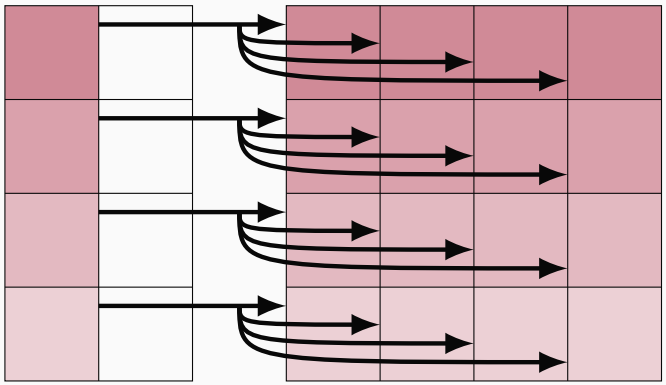
- **Data mapping**
  - 2D block-cyclic
- **Workload mapping**
  - owner of C computes
- **Communications**
  - row-wise broadcast of A
  - column-wise broadcast of B

```
! I am r × c
do l=1, k
   forall i, i%p==r :
      bcast(A_{i,l}, r×*)
   forall j, j%q==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall j, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms





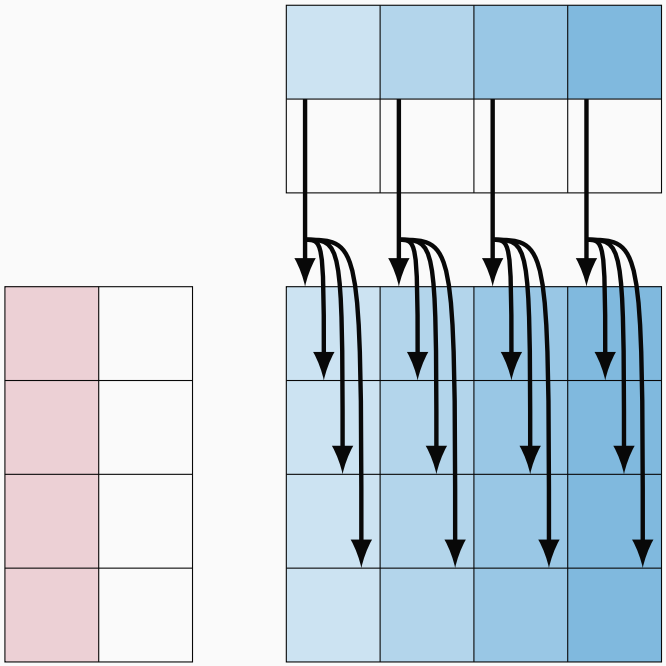
2D C-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic
- **Workload mapping**
  - owner of C computes
- **Communications**
  - row-wise **broadcast** of A
  - column-wise broadcast of B

```
! I am r × c
do l=1, k
   forall i, i%p==r :
      bcast(A_{i,l}, r×*)
   forall j, j%q==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall j, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms



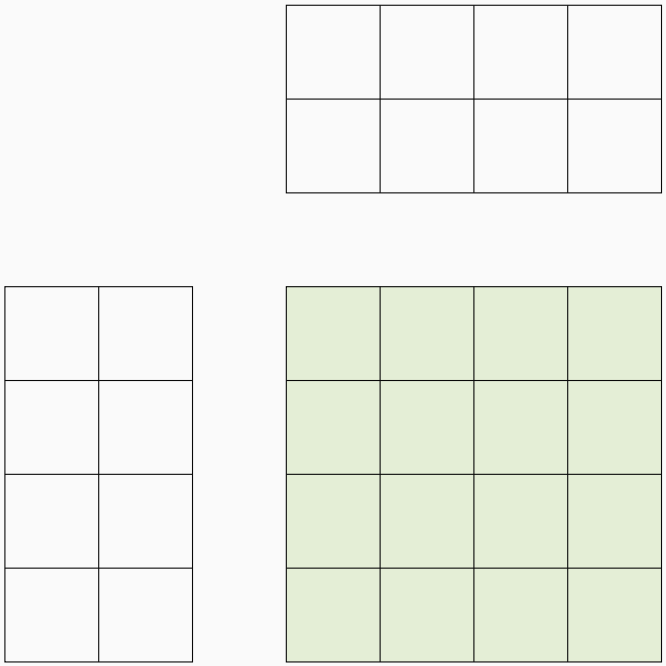
2D C-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic
- **Workload mapping**
  - owner of C computes
- **Communications**
  - row-wise broadcast of A
  - column-wise **broadcast** of B

```
! I am r × c
do l=1, k
   forall i, i%p==r :
      bcast(A_{i,l}, r×*)
   forall j, j%q==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall j, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms



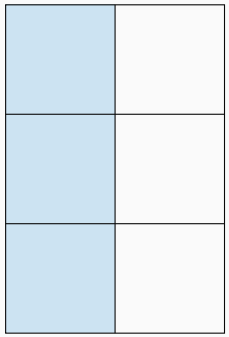
2D C-stationary SUMMA

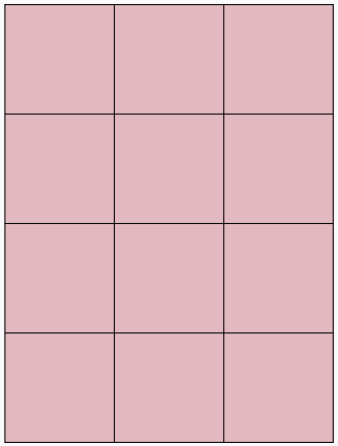
- **Data mapping**
  - 2D block-cyclic
- **Workload mapping**
  - owner of C **computes**
- **Communications**
  - row-wise broadcast of A
  - column-wise broadcast of B

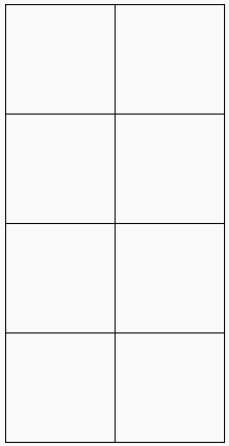
```
! I am r × c
do l=1, k
   forall i, i%p==r :
      bcast(A_{i,l}, r×*)
   forall j, j%q==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall j, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms







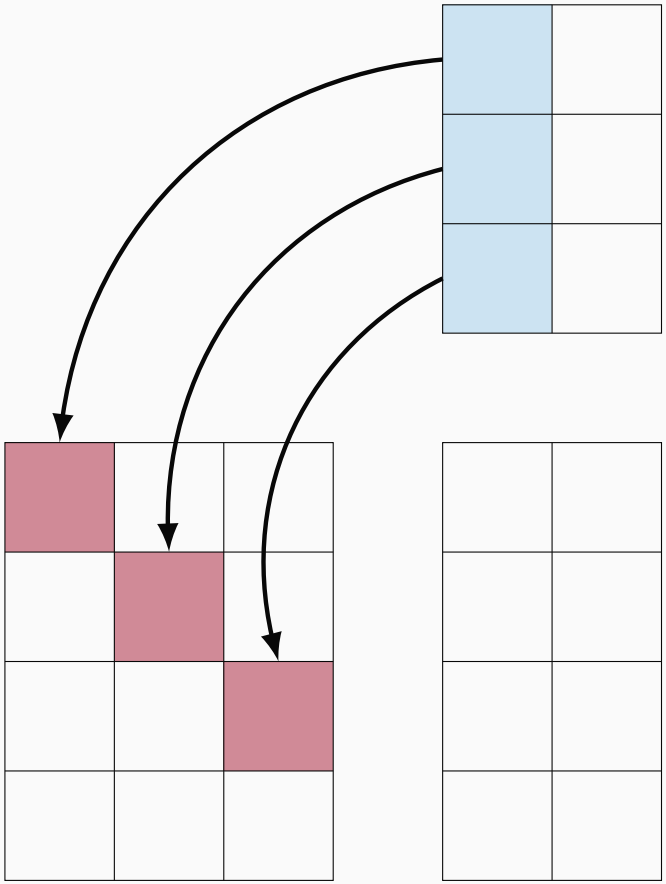
2D A-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic
- **Worload mapping**
  - owner of A computes
- **Communications**
  - column-wise scatter+broadcast of B
  - row-wise reduction of C

```
! I am r × c
do j=1, n
   forall l, l%p==r.and.l%q==c :
      recv(B_{l,j}, l%p×j%q)
   forall l, l%p==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      reduce(C_{i,j}, i%r×j%q)
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms



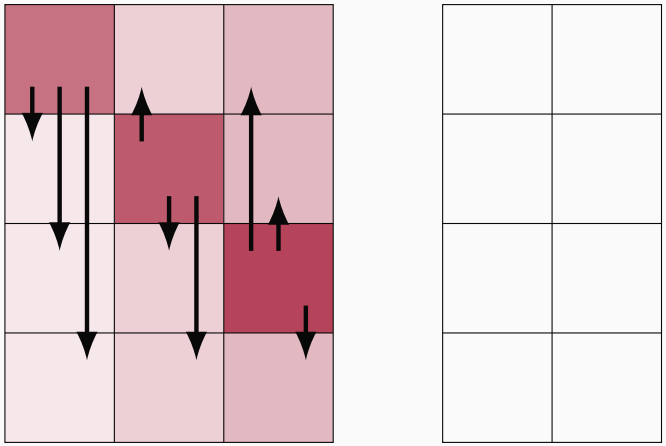
2D **A-stationary** SUMMA

- **Data mapping**
  - 2D block-cyclic
- **Worload mapping**
  - owner of A computes
- **Communications**
  - column-wise **scatter**+broadcast of B
  - row-wise reduction of C

```
! I am r × c
do j=1, n
   forall l, l%p==r.and.l%q==c :
      recv(B_{l,j}, l%p×j%q)
   forall l, l%p==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      reduce(C_{i,j}, i%r×j%q)
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms



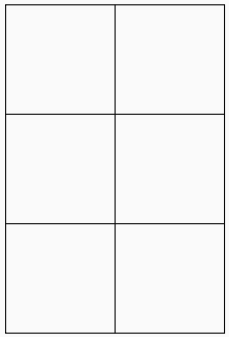
2D **A-stationary** SUMMA

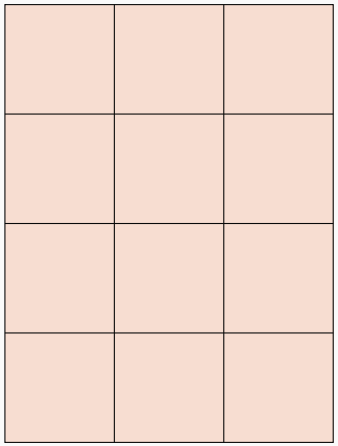
- **Data mapping**
  - 2D block-cyclic
- **Worload mapping**
  - owner of A computes
- **Communications**
  - column-wise scatter+**broadcast** of B
  - row-wise reduction of C

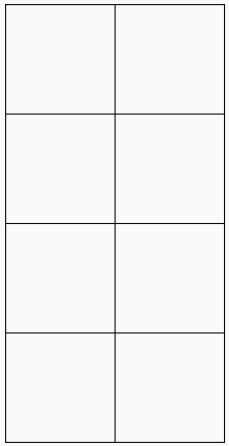
```
! I am r × c
do j=1, n
   forall l, l%p==r.and.l%q==c :
      recv(B_{l,j}, l%p×j%q)
   forall l, l%p==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      reduce(C_{i,j}, i%r×j%q)
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms







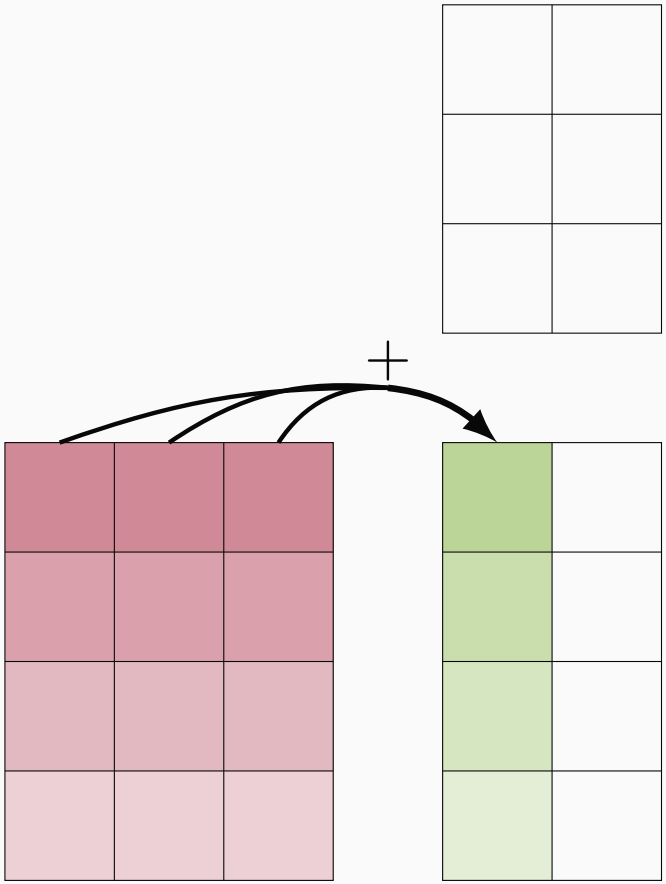
2D A-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic
- **Worload mapping**
  - owner of A **computes**
- **Communications**
  - column-wise scatter+broadcast of B
  - row-wise reduction of C

```
! I am r × c
do j=1, n
   forall l, l%p==r.and.l%q==c :
      recv(B_{l,j}, l%p×j%q)
   forall l, l%p==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      reduce(C_{i,j}, i%r×j%q)
end do
```

Geijn and Watts, 1997

Distributed-memory GEMM Algorithms



2D **A-stationary** SUMMA

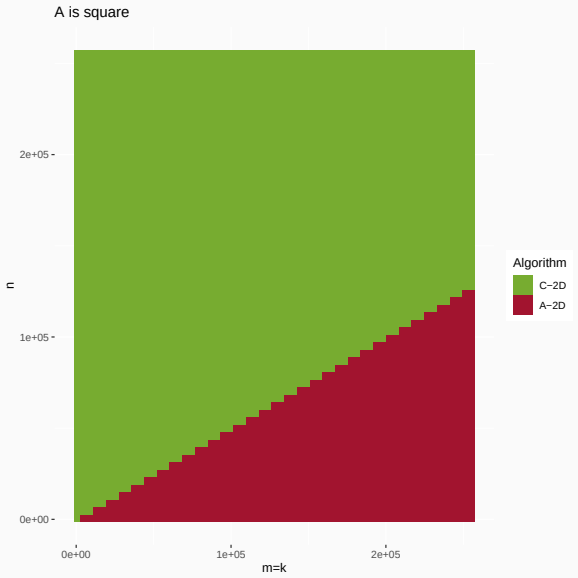
- **Data mapping**
  - 2D block-cyclic
- **Worload mapping**
  - owner of A computes
- **Communications**
  - column-wise scatter+broadcast of B
  - row-wise **reduction** of C

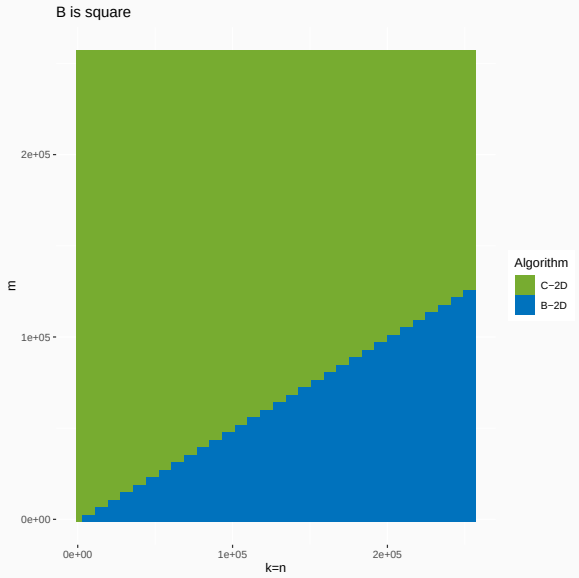
```
! I am r × c
do j=1, n
   forall l, l%p==r.and.l%q==c :
      recv(B_{l,j}, l%p×j%q)
   forall l, l%p==c :
      bcast(B_{l,j}, *×c)
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      reduce(C_{i,j}, i%r×j%q)
end do
```

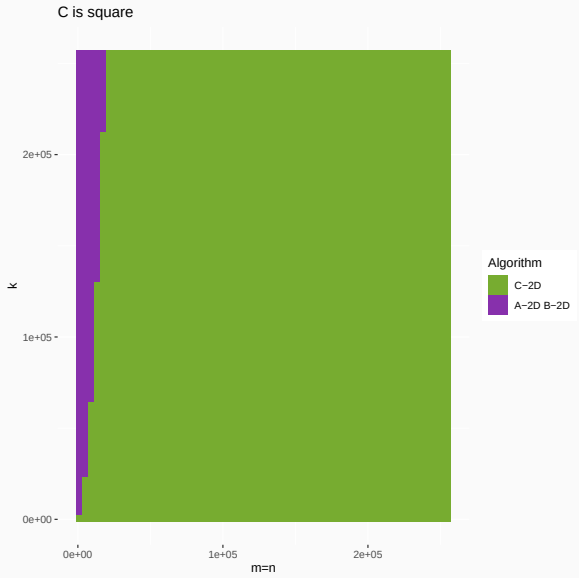
Geijn and Watts, 1997

Distributed-memory GEMM Algorithms

Each variant is best suited for a specific ratio in dimensions.

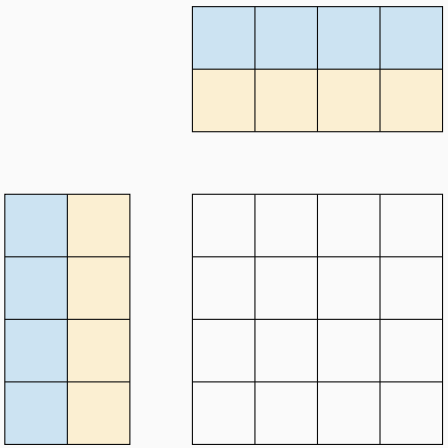




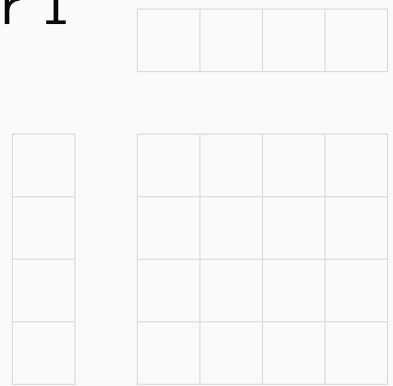


Distributed-memory GEMM Algorithms

Layer 0



Layer 1



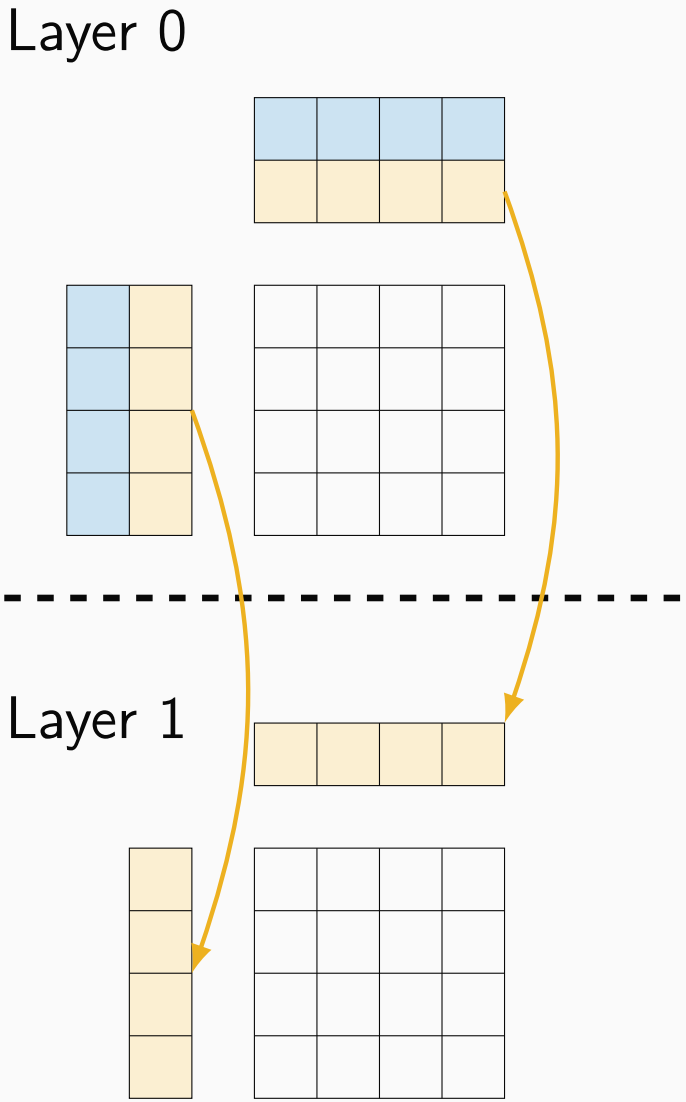
2.5D C-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic on 0^{th} layer
- **Workload mapping**
  - Not trivially related to data mapping
- **Communications**
  - row-wise scatter+broadcast of A
  - column-wise scatter+broadcast of B
  - aisle-wise reduction of C

```
! I am r × c × h
scatter A_{i,j} on layer j/h
scatter B_{i,j} on layer i/h

forall layer=1,h:
   call SUMMA_2DC(A[:, (layer : layer + 1) * k/h],
                  B[(layer : layer + 1) * k/h, :], C^h)

forall i, i%p==r :
  forall j, j%q==c :
    reduce(C^h_{i,j}, r×c×0)
```

Irony, Toledo, and Tiskin, 2004

Distributed-memory GEMM Algorithms



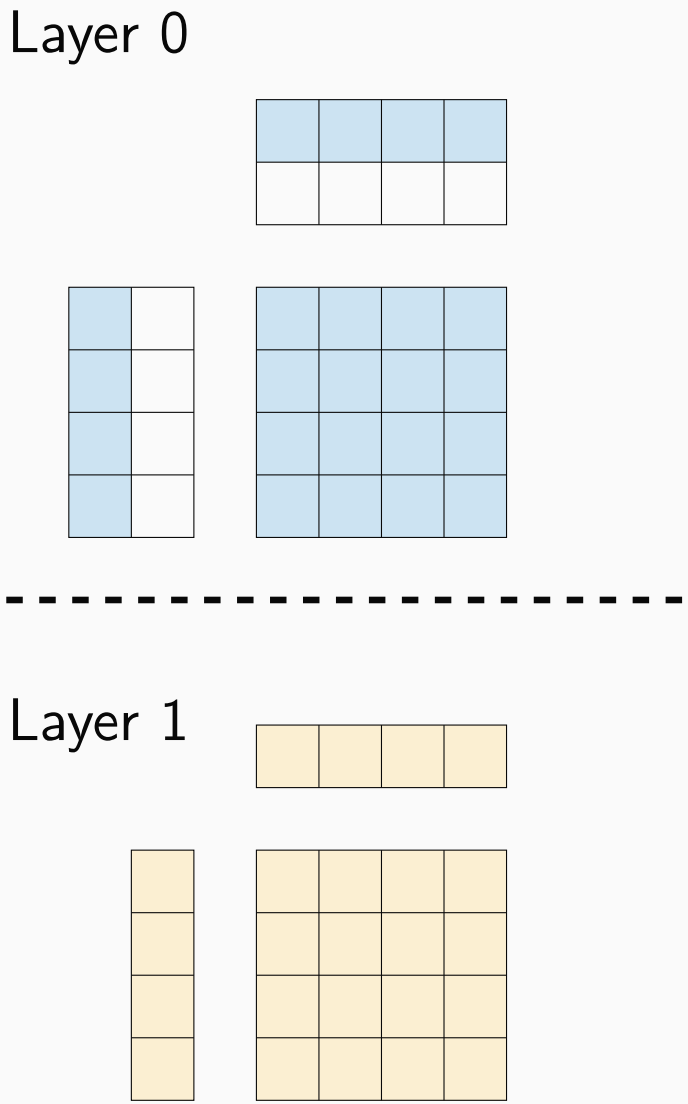
2.5D C-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic on 0^{th} layer
- **Workload mapping**
  - Not trivially related to data mapping
- **Communications**
  - row-wise **scatter**+broadcast of A
  - column-wise **scatter**+broadcast of B
  - aisle-wise reduction of C

```
! I am r × c × h
scatter A_{i,j} on layer j/h
scatter B_{i,j} on layer i/h

forall layer=1,h:
   call SUMMA_2DC(A[:, (layer : layer + 1) * k/h],
                  B[(layer : layer + 1) * k/h, :], C^h)

forall i, i%p==r :
  forall j, j%q==c :
    reduce(C^h_{i,j}, r×c×0)
```

Irony, Toledo, and Tiskin, 2004

Distributed-memory GEMM Algorithms



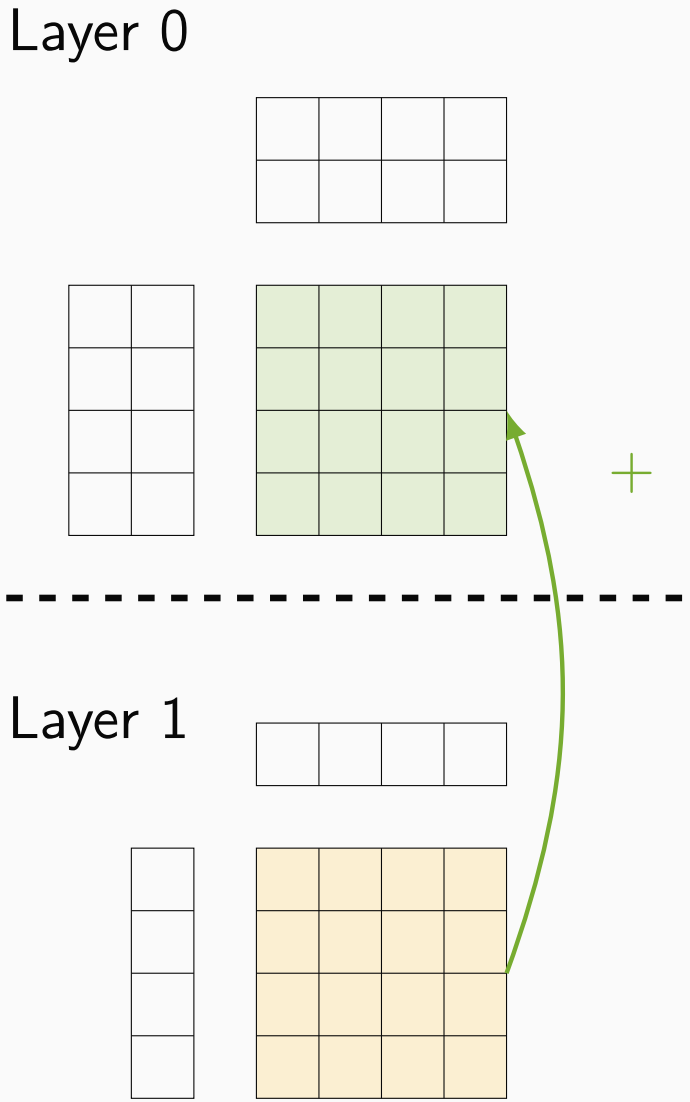
2.5D C-stationary SUMMA

- **Data mapping**
  - 2D block-cyclic on 0^{th} layer
- **Workload mapping**
  - Not trivially related to data mapping
- **Communications**
  - row-wise scatter+**broadcast** of A
  - column-wise scatter+**broadcast** of B
  - aisle-wise reduction of C

```
! I am r × c × h
scatter A_{i,j} on layer j/h
scatter B_{i,j} on layer i/h

forall layer=1,h:
   call SUMMA_2DC(A[:, (layer : layer + 1) * k/h],
                  B[(layer : layer + 1) * k/h, :], C^h)

forall i, i%p==r :
  forall j, j%q==c :
    reduce(C^h_{i,j}, r×c×0)
```

Irony, Toledo, and Tiskin, 2004

Distributed-memory GEMM Algorithms



2.5D C-stationary SUMMA

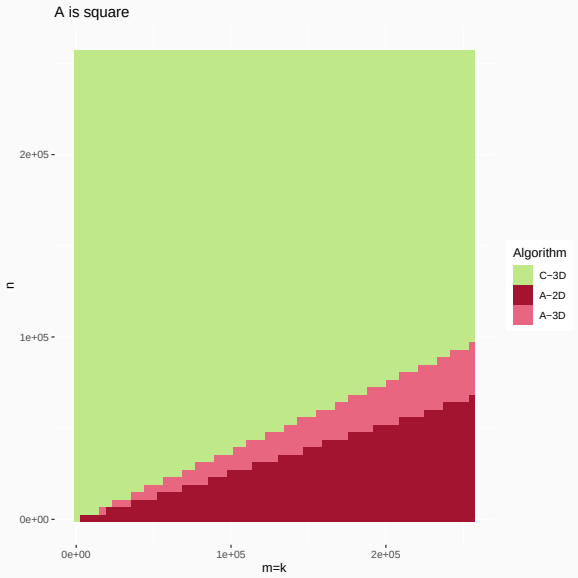
- **Data mapping**
  - 2D block-cyclic on 0^{th} layer
- **Workload mapping**
  - Not trivially related to data mapping
- **Communications**
  - row-wise scatter+broadcast of A
  - column-wise scatter+broadcast of B
  - aisle-wise **reduction** of C

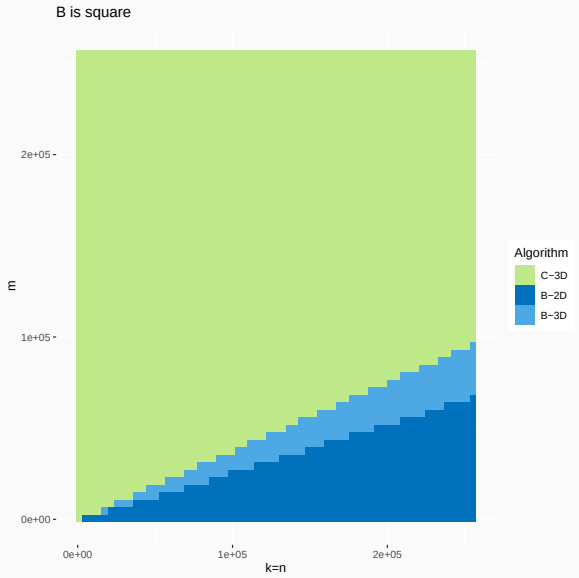
```
! I am r × c × h
scatter A_{i,j} on layer j/h
scatter B_{i,j} on layer i/h

forall layer=1,h:
   call SUMMA_2DC(A[:, (layer : layer + 1) * k/h],
                  B[(layer : layer + 1) * k/h, :], C^h)

forall i, i%p==r :
  forall j, j%q==c :
    reduce(C^h_{i,j}, r×c×0)
```

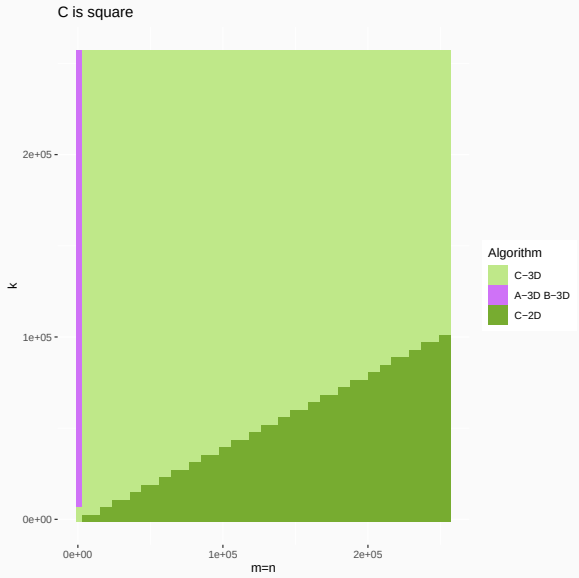
Irony, Toledo, and Tiskin, 2004

Distributed-memory GEMM Algorithms

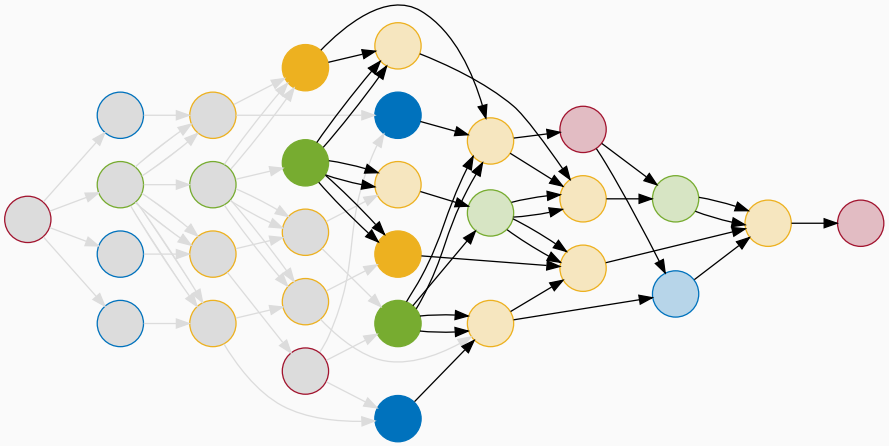
Each variant is best suited for a specific ratio in dimensions.







Task-based programming model



Directed acyclic graph (DAG)

- **Node** Elementary task
- **Edge** Dependencies between task

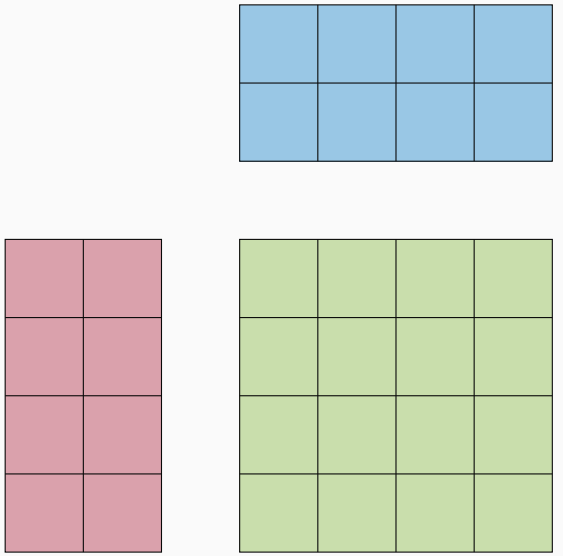
Several libraries expose a task-based programming model

- OpenMP (since 4.0)
- StarPU – developed in Bordeaux
- ParSEC
- ...

The sequential task flow (STF) Model

Sequential GEMM

```
do i=1,m
  do j=1,n
    do l=1,k
      call gemm(
                    A_{i,l},
                    B_{l,j},
                    C_{i,j} )
    end do
  end do
end do
```



Shared-memory STF GEMM

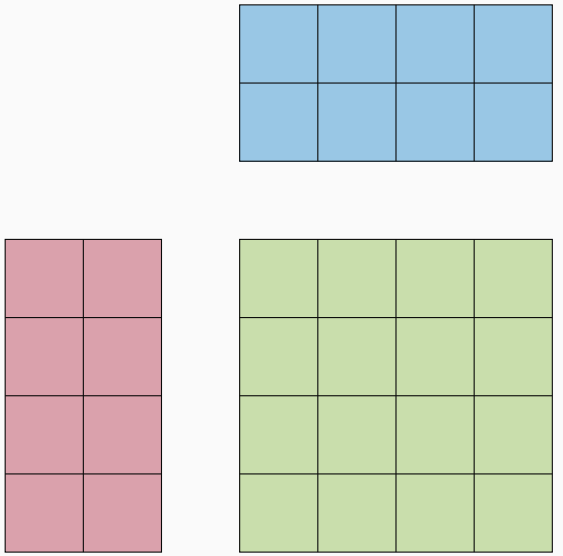
```
do i=1,m
  do j=1,n
    do l=1,k
      call insert_task( gemm,
                    A_{i,l}:R,
                    B_{l,j}:R,
                    C_{i,j}:RW)
    end do
  end do
end do
```


DAG of GEMM

The sequential task flow (STF) Model

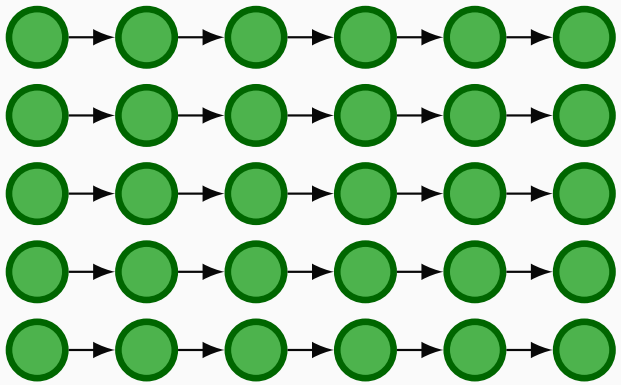
Sequential GEMM

```
do i=1,m
  do j=1,n
    do l=1,k
      call gemm(
                  A(i,l),
                  B(l,j),
                  C(i,j) )
    end do
  end do
end do
```



Shared-memory STF GEMM

```
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):RW)
    end do
  end do
end do
```



DAG of GEMM

Objectives

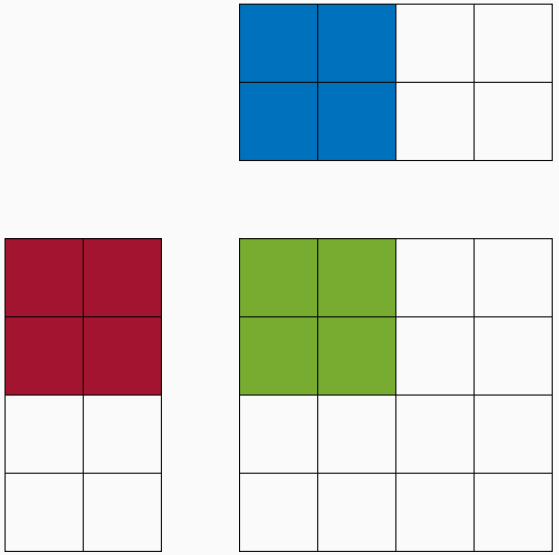
- Can we use the **STF model** to express a **scalable distributed-memory GEMM** algorithm ?

- How **efficient** is the resulting expression compared to reference libraries ?

DISTRIBUTED-MEMORY STF GEMM ALGORITHM

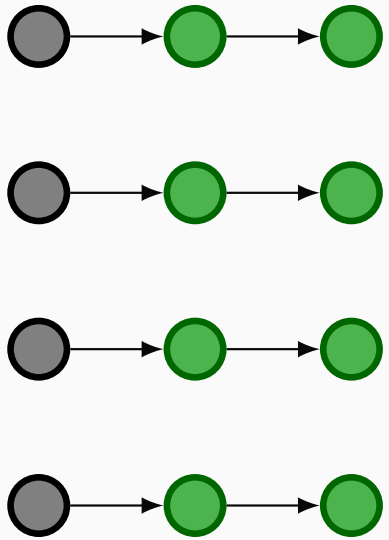
Baseline STF model

```
!
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}:RW)
    end do
  end do
end do
```



Baseline model (M0)[a]

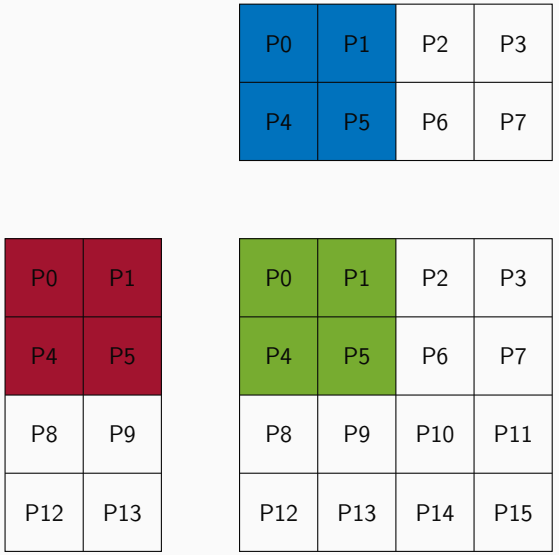
- M0-0 Data mapping
- M0-1 Task mapping **inferred** from DM (RW)
- M0-2 Point-to-point comms. **inferred** from TM



[a] Agullo, Aumage, Faverge, Furmento, Pruvost, Sergent, and Thibault, 2017.

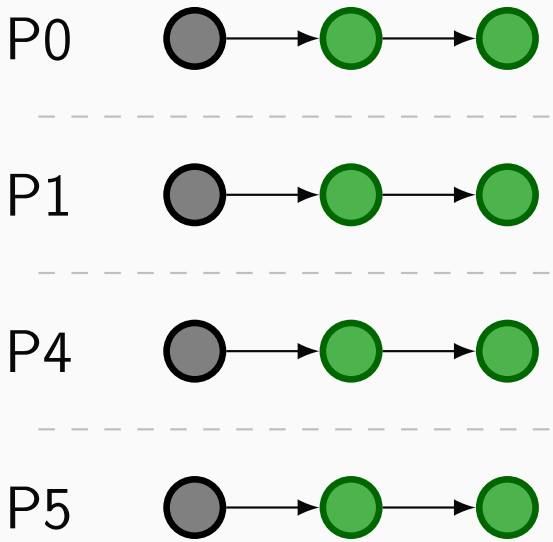
Baseline STF model

```
! data_map: A,B,C distributed on a grid
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}:RW)
    end do
  end do
end do
```



Baseline model (M0)[a]

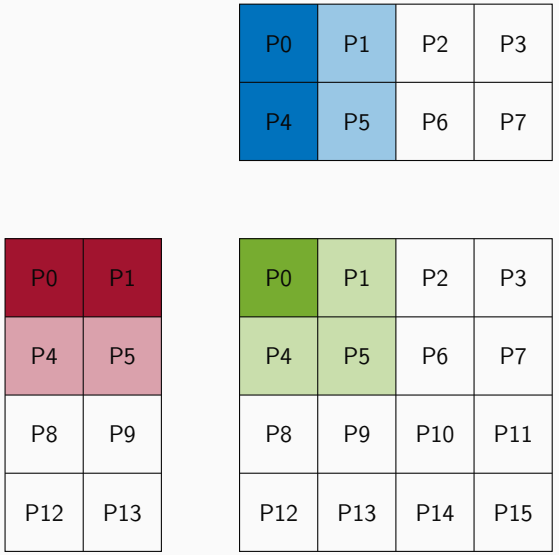
- M0-0 Data mapping
- M0-1 Task mapping **inferred** from DM (RW)
- M0-2 Point-to-point comms. **inferred** from TM



[a] Agullo, Aumage, Faverge, Furmento, Pruvost, Sergent, and Thibault, 2017.

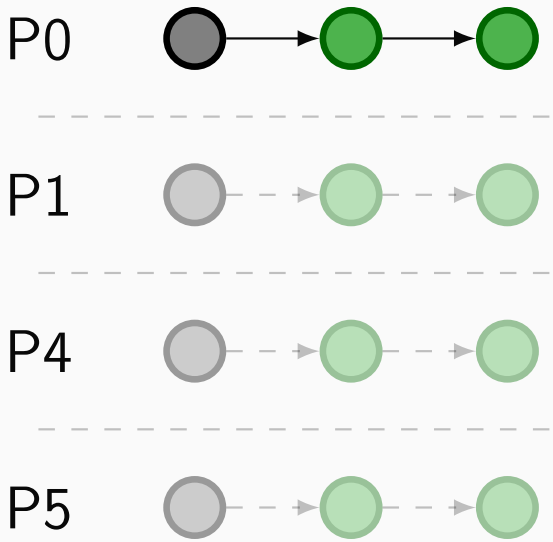
Baseline STF model

```
! data_map: A,B,C distributed on a grid
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}:RW)
    end do
  end do
end do
```



Baseline model (M0)[a]

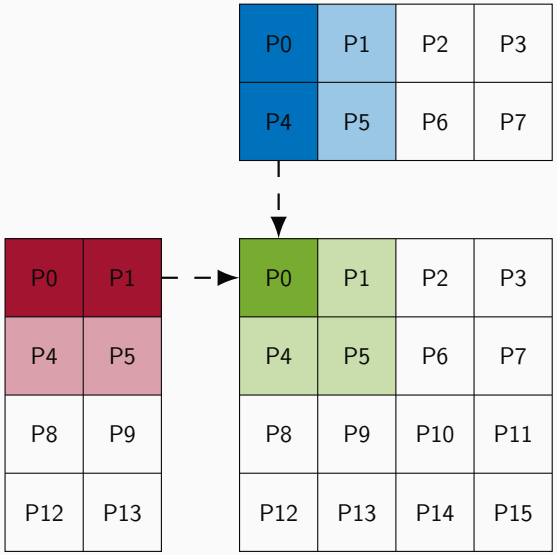
- M0-0 Data mapping
- M0-1 Task mapping **inferred** from DM (RW)
- M0-2 Point-to-point comms. **inferred** from TM



[a] Agullo, Aumage, Faverge, Furmento, Pruvost, Sergent, and Thibault, 2017.

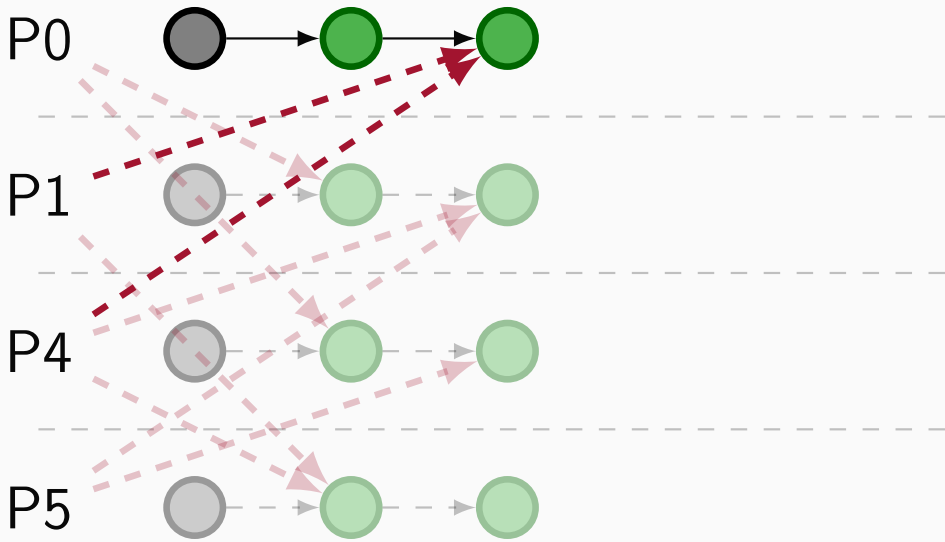
Baseline STF model

```
! data_map: A,B,C distributed on a grid
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}:RW)
    end do
  end do
end do
```



Baseline model (M0)[a]

- M0-0 Data mapping
- M0-1 Task mapping **inferred** from DM (RW)
- M0-2 Point-to-point comms. **inferred** from TM



[a] Agullo, Aumage, Faverge, Furmento, Pruvost, Sergent, and Thibault, 2017.

Limits to the baseline model

The baseline STF model `M0` does not allow the expression of SUMMA algorithms

- Task mapping is bound to data mapping (`M0-1`)
- Reduction patterns are hard to make
- Point-to-point communications patterns are inefficient (`M0-2`)

We have identified key functionalities required in a STF model `MX`, extended from `M0`, to allow the expression of SUMMA algorithms.

Limits to the baseline model

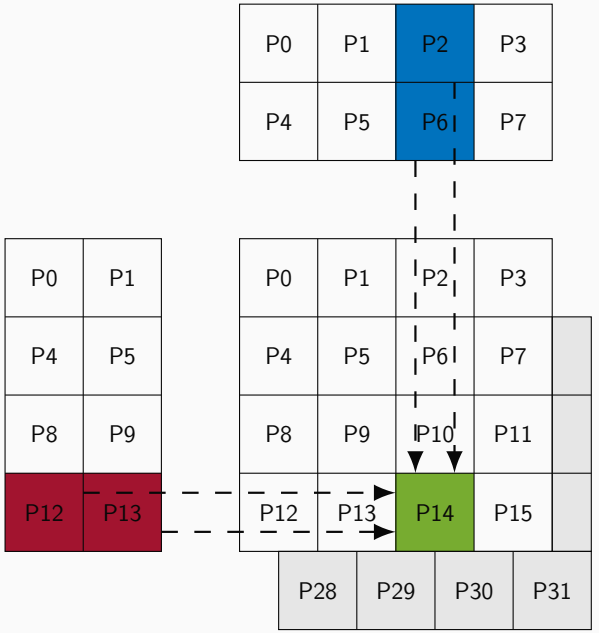
The baseline STF model `M0` does not allow the expression of SUMMA algorithms

- Task mapping is bound to data mapping (`M0-1`)
- Reduction patterns are hard to make
- Point-to-point communications patterns are inefficient (`M0-2`)

We have identified key functionalities required in a STF model `MX`, extended from `M0`, to allow the expression of SUMMA algorithms.

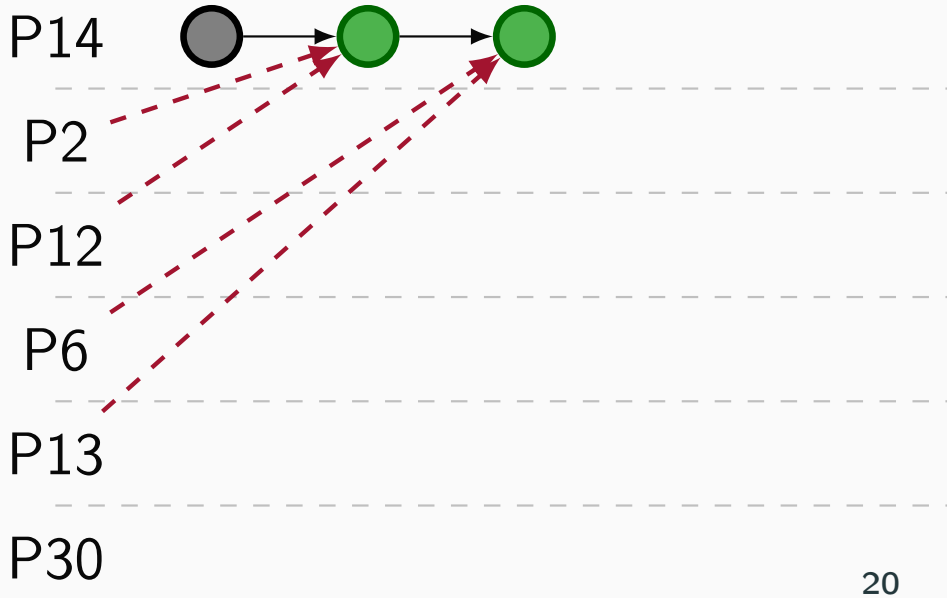
Key 1 – Task mapping

```
!data_map: A,B,C distributed on one
      layer
!
!

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}:RW)
    end do
  end do
end do
```



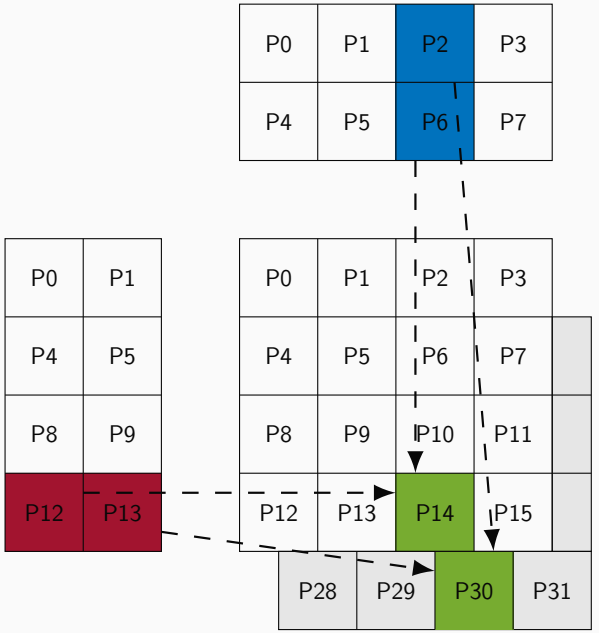
Advanced model MX

- M0-0 Data mapping
- M0-1 Implicit task mapping
- M0-2 Comms. inferred



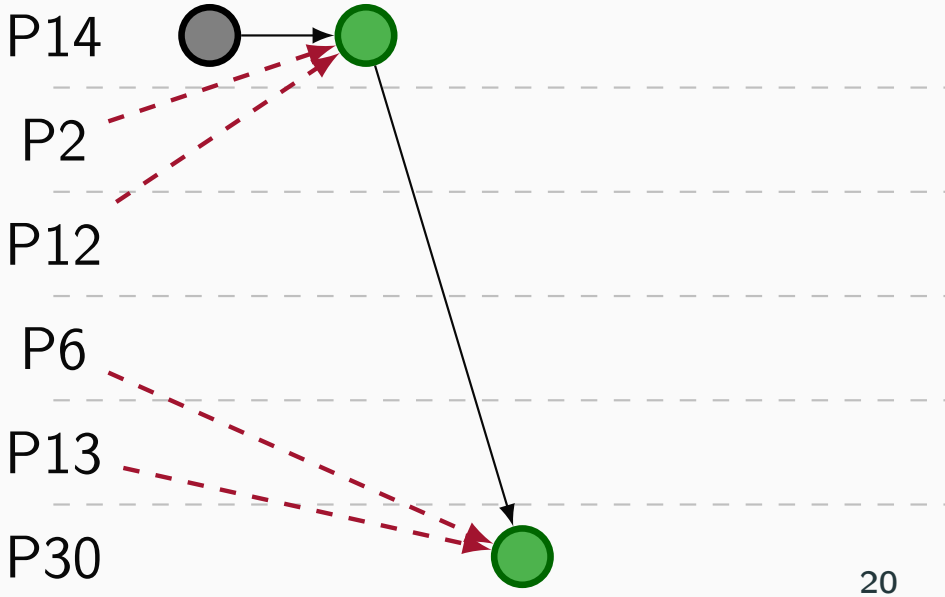
Key 1 – Task mapping

```
!data_map: A,B,C distributed on one
     layer
!task_map: executing node of A_{i,l}B_{l,j}
!

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}:RW,
                       task_map[i,j,l])
    end do
  end do
end do
```



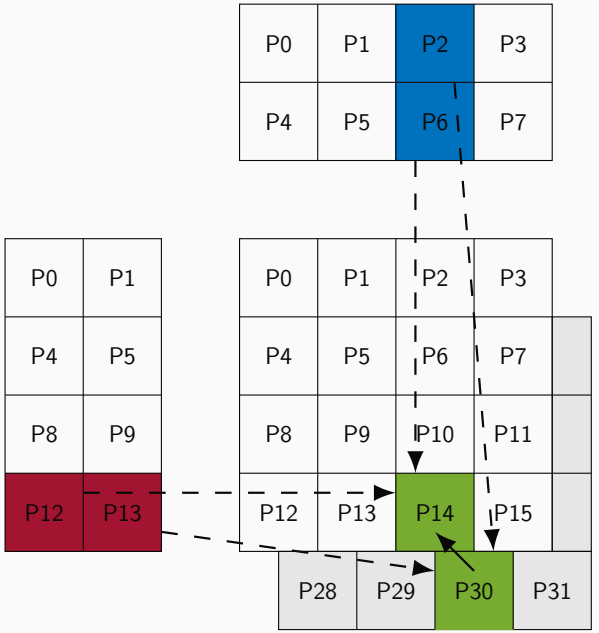
Advanced model MX

- MO-0 Data mapping
- MX-1 **Explicit** task mapping
- MO-2 Comms. inferred



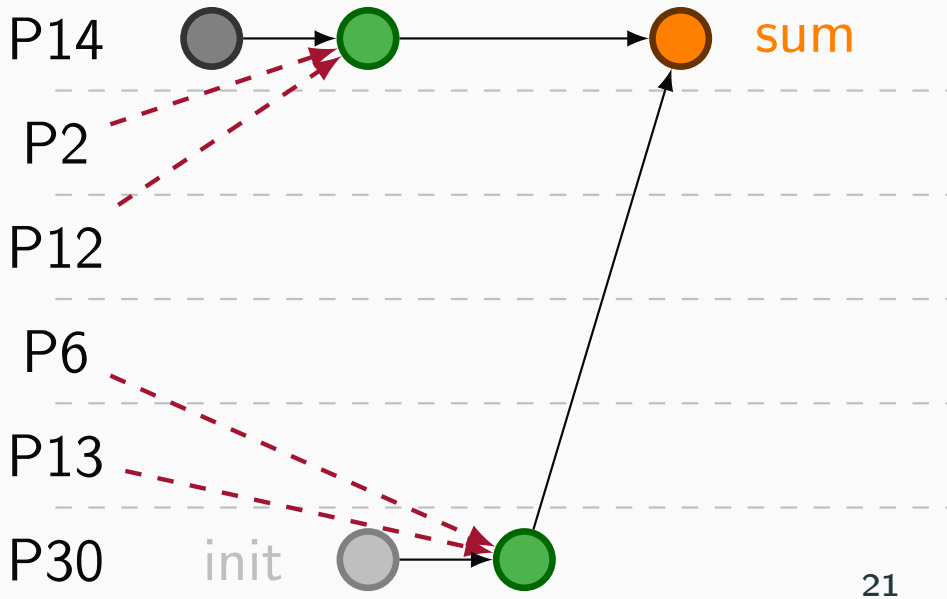
Key 2 – Reduction pattern

```
!data_map: A,B,C distributed on one
     layer
!task_map: executing node of A_{i,l}B_{l,j}
!
call instantiate_local(C)
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}:RW,
                       task_map[i,j,l])
    end do
  end do call reduce(C_{i,j})
end do
```



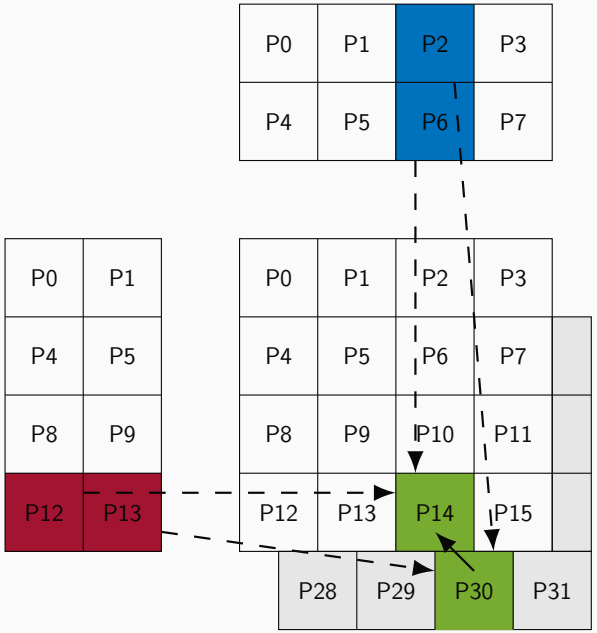
Advanced model MX

- MO-0 Data mapping
- MX-1 Explicit task mapping
- MO-2 Comms. inferred
- MO-3 Explicit reduction pattern



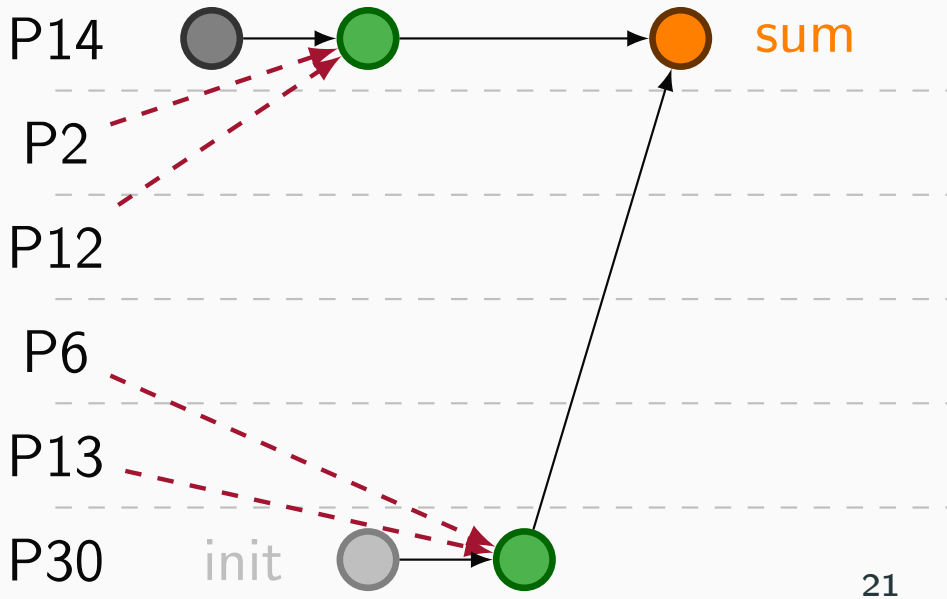
Key 2 – Reduction pattern

```
!data_map: A,B,C distributed on one
     layer
!task_map: executing node of A_{i,l}B_{l,j}
!
call instantiate_local(C)
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 call reduce(C_{i,j})
end do
```



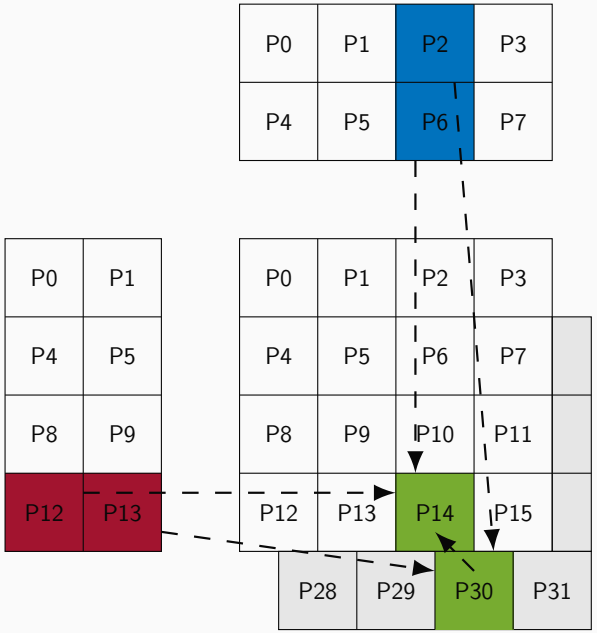
Advanced model MX

- MO-0 Data mapping
- MX-1 Explicit task mapping
- MO-2 Comms. inferred
- MO-3 Explicit reduction pattern



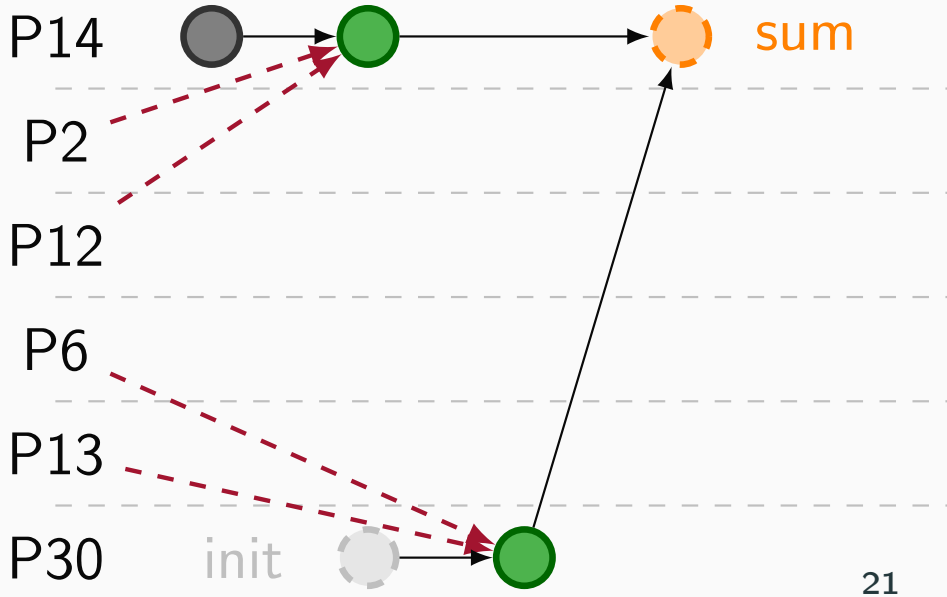
Key 2 – Reduction pattern

```
!data_map: A,B,C distributed on one
     layer
!task_map: executing node of A_{i,l}B_{l,j}
!
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
```



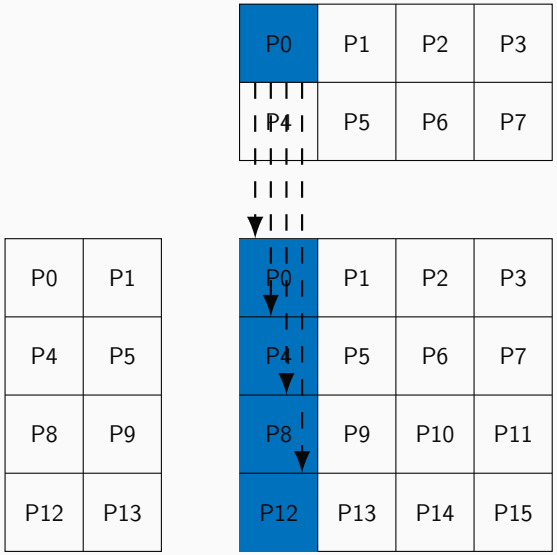
Advanced model MX

- MO-0 Data mapping
- MX-1 Explicit task mapping
- MO-2 Comms. inferred
- MX-3 **Implicit** reduction pattern



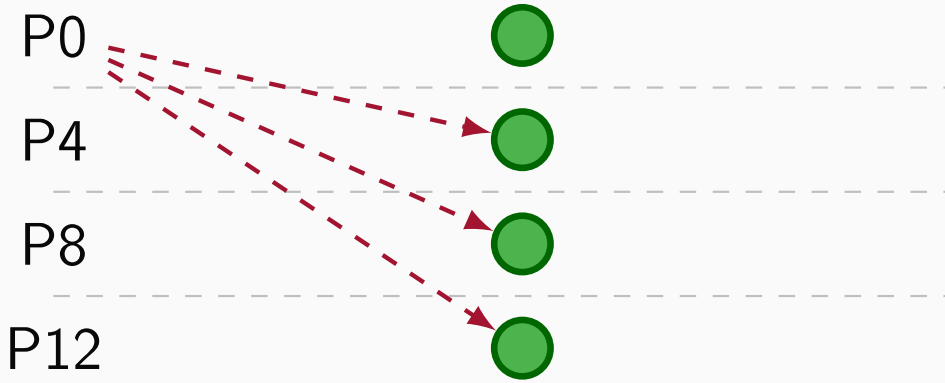
Key 3 – Communications

```
!data_map: A,B,C distributed on one
     layer
!task_map: executing node of A_{i,l}B_{l,j}
!
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
```



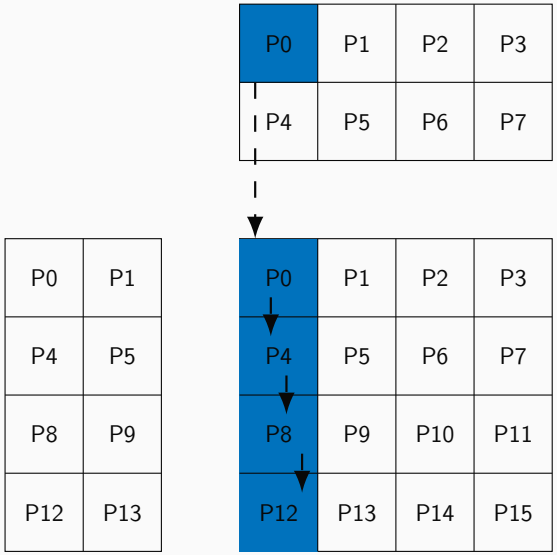
Advanced model MX

- MO-0 Data mapping
- MX-1 Explicit task mapping
- MO-2 Comms. inferred
- MX-3 Implicit reduction pattern



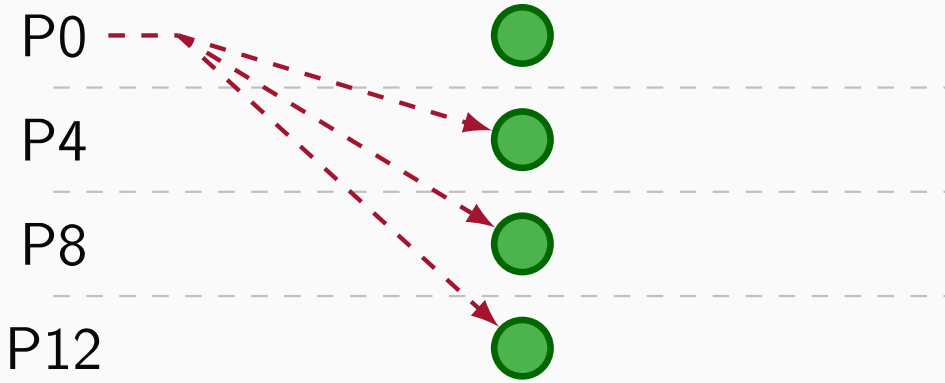
Key 3 – Communications

```
!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
```



Advanced model MX

- MO-0 Data mapping
- MX-1 Explicit task mapping
- MX-2 **Coll. comms.** inferred [1]
- MX-3 Implicit reduction pattern



[1] Denis, Jeannot, Swartvagher, and Thibault, 2020.

Ideal expression

```
!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
```

- `task_map` can be set to cover any SUMMA variant
- Agnostic of:
  - the data distribution
  - the scheduling
  - the processing units

Advanced model `MX`

- `M0-0` Data mapping
- `MX-1` **Explicit** task mapping
- `MX-2` **Collective** communications **inferred** from task mapping
- `MX-3` **Implicit** reduction pattern

Ideal expression - task mappings

- 2D-C-Stat owner of C computes
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RW`

- 2D-A-Stat owners of A compute
  - `task_map[i,j,l]` $= l \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RANK_REDUX`

- 2.5D-C-Stat several nodes compute
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r) + rc \frac{l}{k/h}$
  - `access_mode` $=$ `RANK_REDUX`

The advanced model allows to **cover all variants** of SUMMA algorithms with three nested loops.
In a **distribution-agnostic** fashion.

Ideal expression - task mappings

- 2D-C-Stat owner of C computes
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RW.and.COMMUTE`

- 2D-A-Stat owners of A compute
  - `task_map[i,j,l]` $= l \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RANK_REDUX`

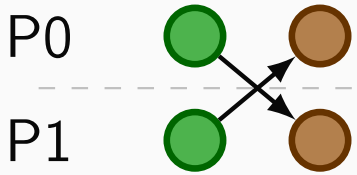
- 2.5D-C-Stat several nodes compute
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r) + rc \frac{l}{k/h}$
  - `access_mode` $=$ `RANK_REDUX`

The advanced model allows to **cover all variants** of SUMMA algorithms with three nested loops.
In a **distribution-agnostic** fashion.

Ideal expression - task mappings

- 2D-C-Stat owner of C computes
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RW.and.COMMUTE`

- 2D-A-Stat owners of A compute
  - `task_map[i,j,l]` $= l \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RANK_REDUX`



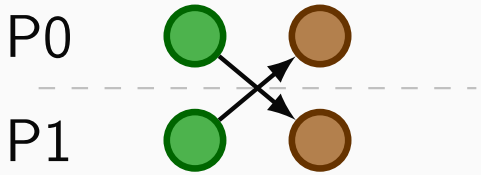
- 2.5D-C-Stat several nodes compute
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r) + rc \frac{l}{k/h}$
  - `access_mode` $=$ `RANK_REDUX`

The advanced model allows to **cover all variants** of SUMMA algorithms with three nested loops.
In a **distribution-agnostic** fashion.

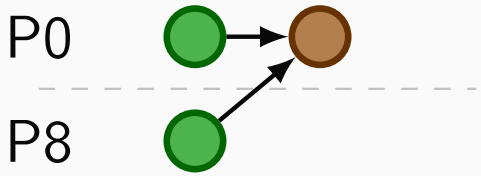
Ideal expression - task mappings

- 2D-C-Stat owner of C computes
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RW.and.COMMUTE`

- 2D-A-Stat owners of A compute
  - `task_map[i,j,l]` $= l \,\%\, c + c * (i \,\%\, r)$
  - `access_mode` $=$ `RANK_REDUX`



- 2.5D-C-Stat several nodes compute
  - `task_map[i,j,l]` $= j \,\%\, c + c * (i \,\%\, r) + rc \frac{l}{k/h}$
  - `access_mode` $=$ `RANK_REDUX`

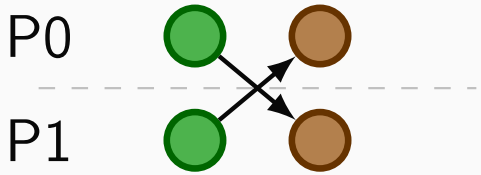


The advanced model allows to **cover all variants** of SUMMA algorithms with three nested loops.
In a distribution-agnostic fashion.

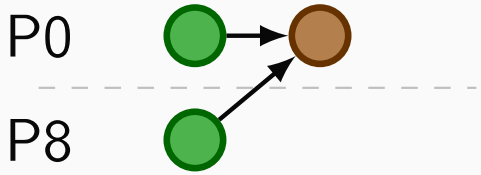
Ideal expression - task mappings

- 2D-C-Stat owner of C computes
  - `task_map[i,j,l]` $= C_{i,j}.owner\_rank$
  - `access_mode` $=$ `RW.and.COMMUTE`

- 2D-A-Stat owners of A compute
  - `task_map[i,j,l]` $= A_{i,l}.owner\_rank$
  - `access_mode` $=$ `RANK_REDUX`



- 2.5D-C-Stat several nodes compute
  - `task_map[i,j,l]` $= C_{i,j}.owner\_rank + rc\frac{l}{k/h}$
  - `access_mode` $=$ `RANK_REDUX`



The advanced model allows to **cover all variants** of SUMMA algorithms with three nested loops.
In a **distribution-agnostic** fashion.

EXPERIMENTAL RESULTS

Experimental setup

- **Hardware**: Très Grand Centre de Calcul (TGCC)
  - **Irène - Skylake**
    - Dual-processor Intel Skylake 8168 @ 2.7 GHz
    - 24 cores per processor, 192 GB DDR4 memory
    - Infiniband EDR, 4x links, «fat tree» topology
  - **Irène - Rome**
    - Dual-processor AMD Rome (Epyc) @ 2.6 GHz
    - 64 cores per processor, 256 GB DDR4 memory
    - Infiniband HDR100, 2x links, «dragonfly» topology

Experimental setup

- **Software**: GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model `MX` + seq. BLAS
    - 2.5D A,B,C-stationary version
  - `scalapack` – OpenMPI 4.0.5, 2 cores / MPI
    - Synchronous MPI + seq. BLAS
    - 2D A,B,C-stationary
  - `slate` – OpenMPI 4.0.5, 1 MPI / node
    - MPI + OpenMP with `task` + seq. BLAS
    - 2D A,C-stationary with lookahead
  - `elemental` – OpenMPI 4.0.5, 2 cores / MPI
    - Flame runtime + seq. BLAS
    - 2D A,B,C-stationary
  - `chameleon` – newMadeleine, 1 MPI / node
    - StarPU with explicit comms. management + seq. BLAS
    - 2D C-stationary pipelined with lookahead

Experimental setup

- **Software**: GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model `MX` + seq. BLAS
    - 2.5D A,B,C-stationary version
  - `scalapack` – OpenMPI 4.0.5, 2 cores / MPI
    - Synchronous MPI + seq. BLAS
    - 2D A,B,C-stationary
  - `slate` – OpenMPI 4.0.5, 1 MPI / node
    - MPI + OpenMP with `task` + seq. BLAS
    - 2D A,C-stationary with lookahead
  - `elemental` – OpenMPI 4.0.5, 2 cores / MPI
    - Flame runtime + seq. BLAS
    - 2D A,B,C-stationary
  - `chameleon` – newMadeleine, 1 MPI / node
    - StarPU with explicit comms. management + seq. BLAS
    - 2D C-stationary pipelined with lookahead

Experimental setup

- **Software**: GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model `MX` + seq. BLAS
    - 2.5D A,B,C-stationary version
  - `scalapack` – OpenMPI 4.0.5, 2 cores / MPI
    - Synchronous MPI + seq. BLAS
    - 2D A,B,C-stationary
  - `slate` – OpenMPI 4.0.5, 1 MPI / node
    - MPI + OpenMP with `task` + seq. BLAS
    - 2D A,C-stationary with lookahead
  - `elemental` – OpenMPI 4.0.5, 2 cores / MPI
    - Flame runtime + seq. BLAS
    - 2D A,B,C-stationary
  - `chameleon` – newMadeleine, 1 MPI / node
    - StarPU with explicit comms. management + seq. BLAS
    - 2D C-stationary pipelined with lookahead

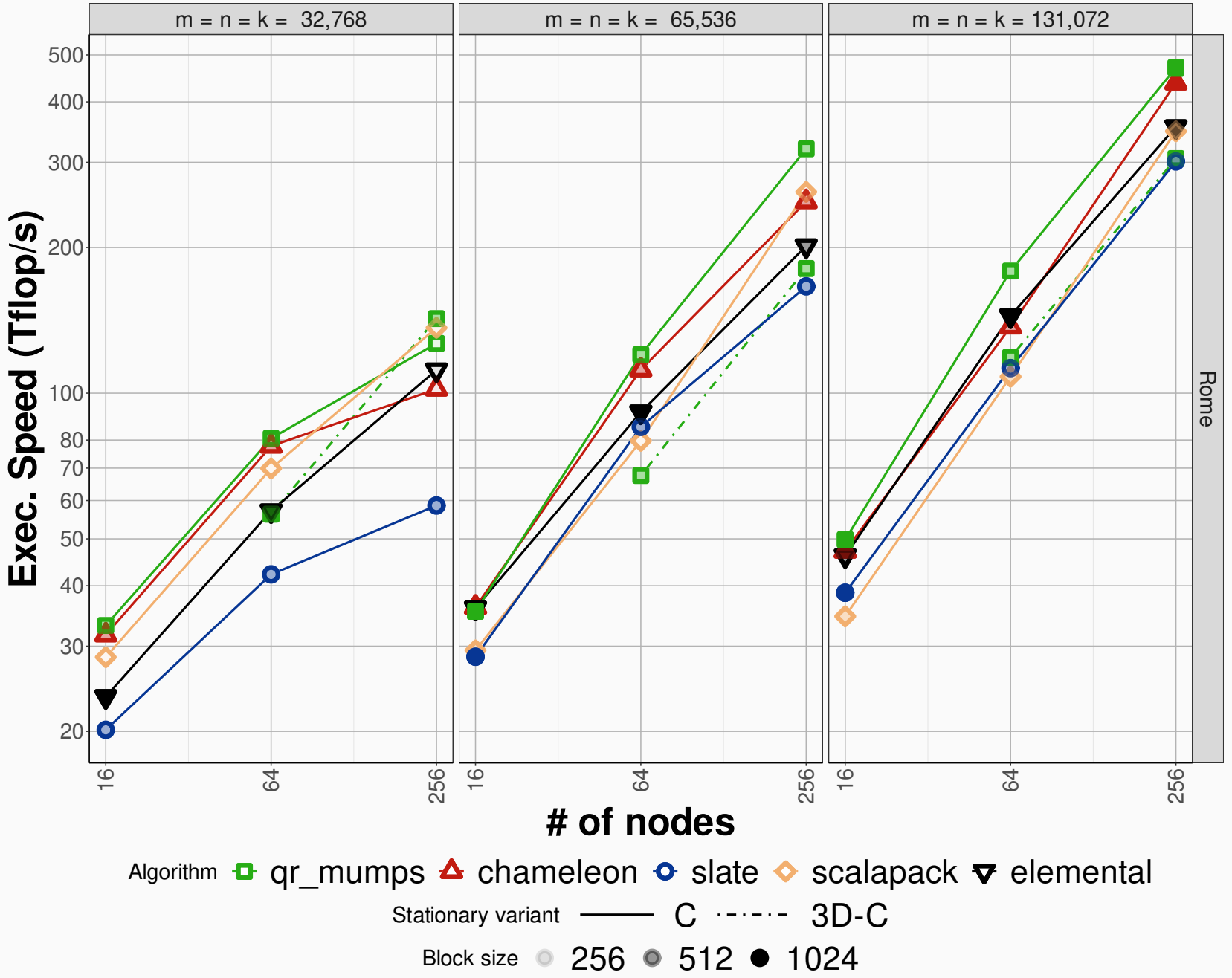
Experimental setup

- **Software**: GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model `MX` + seq. BLAS
    - 2.5D A,B,C-stationary version
  - `scalapack` – OpenMPI 4.0.5, 2 cores / MPI
    - Synchronous MPI + seq. BLAS
    - 2D A,B,C-stationary
  - `slate` – OpenMPI 4.0.5, 1 MPI / node
    - MPI + OpenMP with `task` + seq. BLAS
    - 2D A,C-stationary with lookahead
  - `elemental` – OpenMPI 4.0.5, 2 cores / MPI
    - Flame runtime + seq. BLAS
    - 2D A,B,C-stationary
  - `chameleon` – newMadeleine, 1 MPI / node
    - StarPU with explicit comms. management + seq. BLAS
    - 2D C-stationary pipelined with lookahead

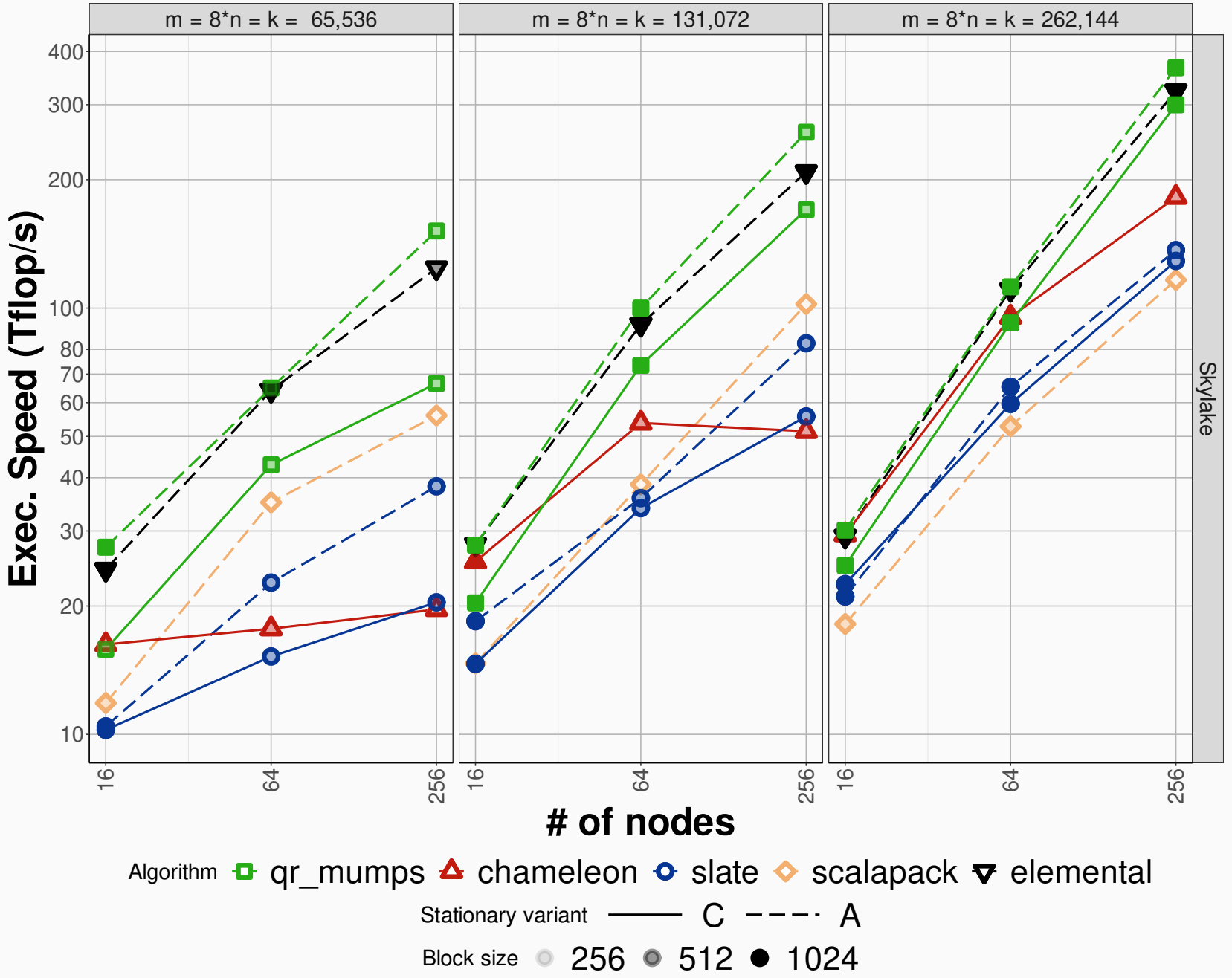
Experimental setup

- **Software**: GNU 9.3.0, MKL 21.3.0
  - `qr_mumps` **ours** – newMadeleine, 1 MPI / node
    - StarPU with advanced model `MX` + seq. BLAS
    - 2.5D A,B,C-stationary version
  - `scalapack` – OpenMPI 4.0.5, 2 cores / MPI
    - Synchronous MPI + seq. BLAS
    - 2D A,B,C-stationary
  - `slate` – OpenMPI 4.0.5, 1 MPI / node
    - MPI + OpenMP with `task` + seq. BLAS
    - 2D A,C-stationary with lookahead
  - `elemental` – OpenMPI 4.0.5, 2 cores / MPI
    - Flame runtime + seq. BLAS
    - 2D A,B,C-stationary
  - `chameleon` – newMadeleine, 1 MPI / node
    - StarPU with explicit comms. management + seq. BLAS
    - 2D C-stationary pipelined with lookahead

Rome (up to 32,768 cores) – square matrices, DGEMM



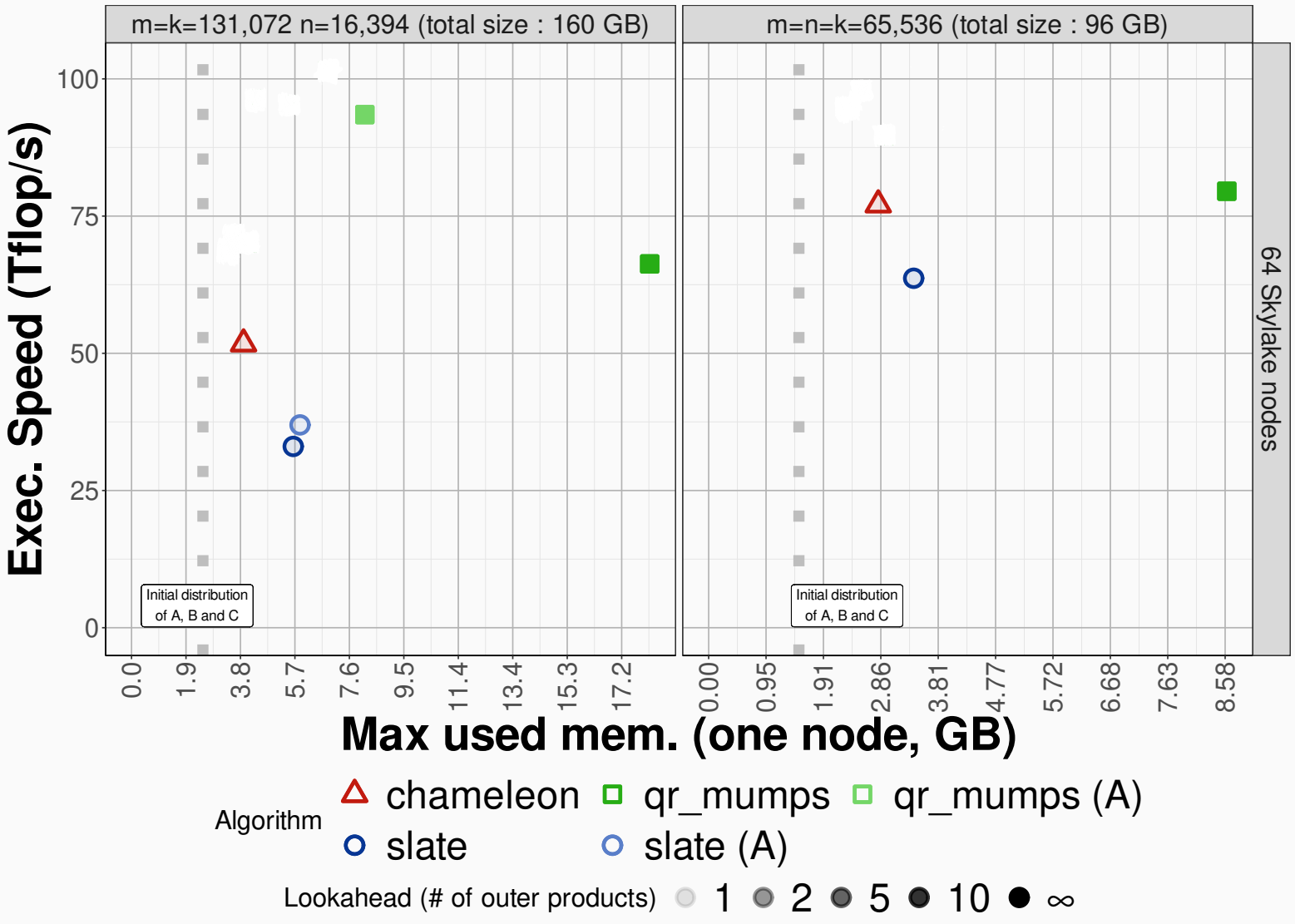
Skylake (up to 12,288 cores) – large A, DGEMM



Skylake – memory consumption

Submitting the **full DAG** allows **greedy** memory consumption

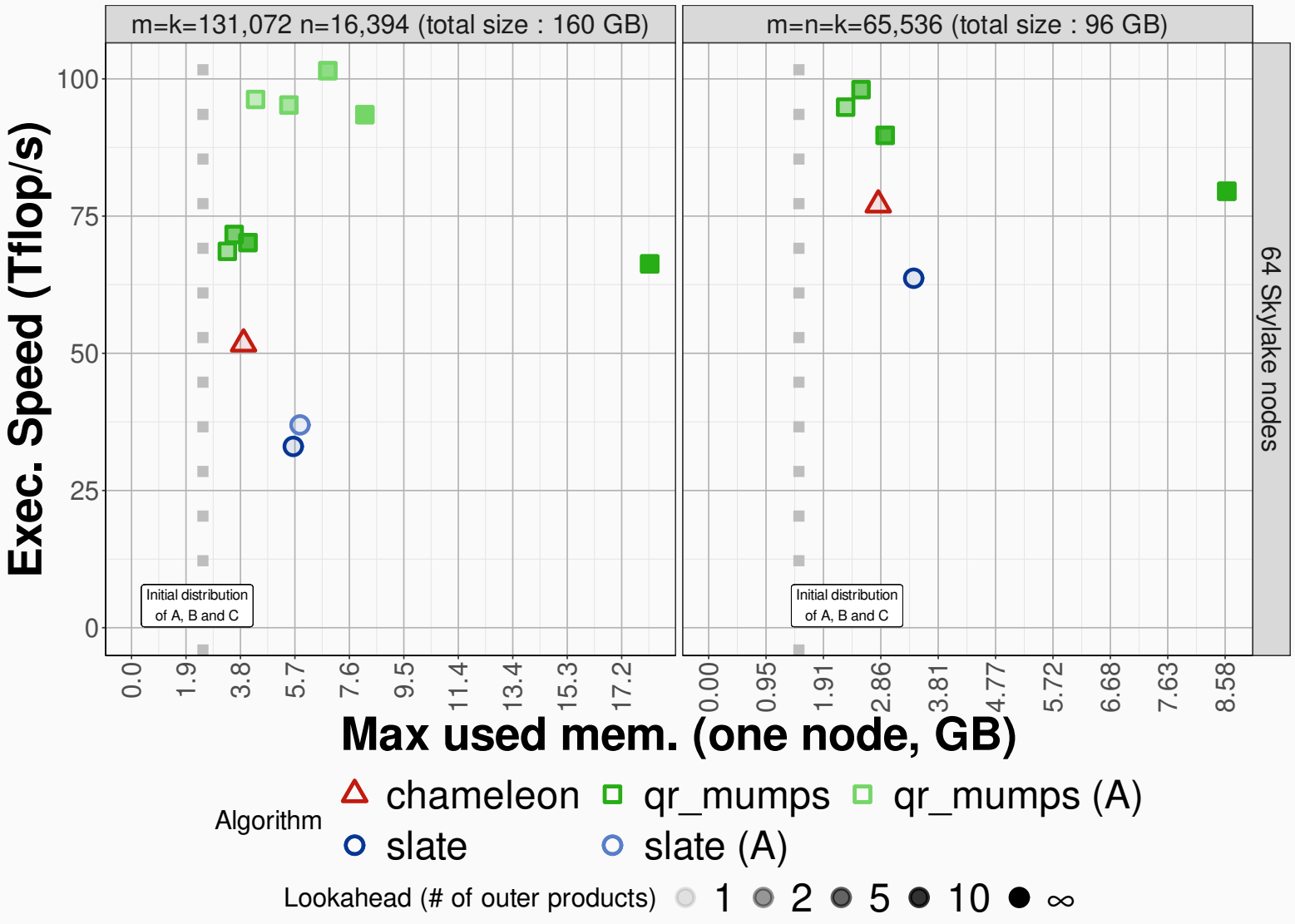
Task window mechanisms can help limit memory consumption



Skylake – memory consumption

Submitting the **full DAG** allows **greedy** memory consumption

Task window mechanisms can help limit memory consumption



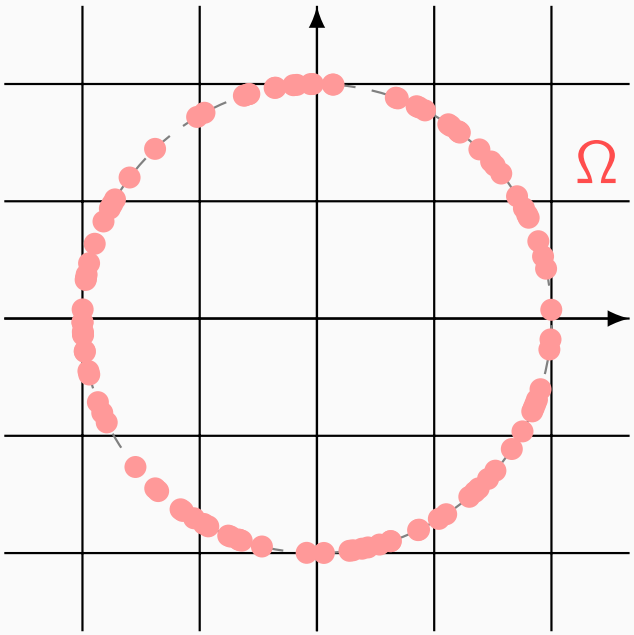
APPLICATION TO MACHINE LEARNING

Random linear algebra – background

Unsupervised learning:
We search for **eigencouples** of A, $n \times n$ matrix.

Two *limitations*:

- SVD is very costly $\sim O(n^3)$
- SVD is complex to parallelize



$A \times \Omega \sim O(n^2 k)$ if Ω is $n \times k$.

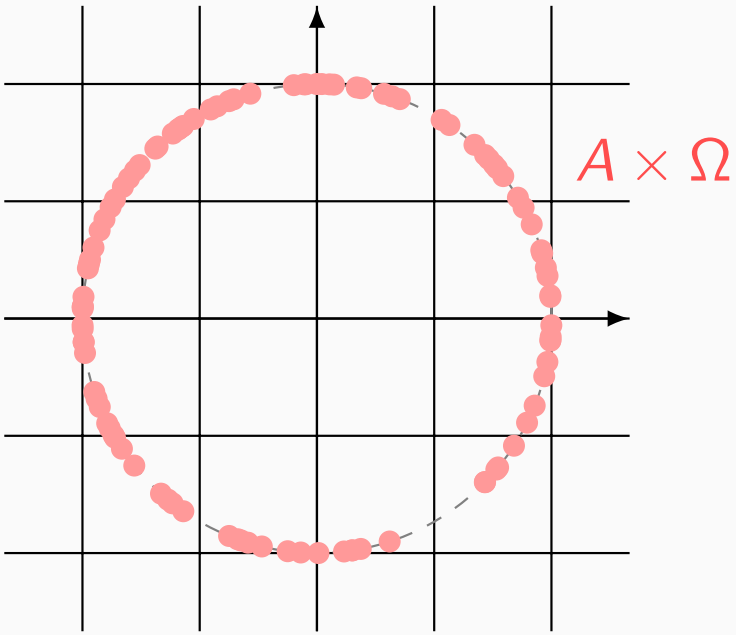
$\Rightarrow$ Approximating eigenvectors is *relatively cheap*.

Random linear algebra – background

Unsupervised learning:
We search for **eigencouples** of A, $n \times n$ matrix.

Two *limitations*:

- SVD is very costly $\sim O(n^3)$
- SVD is complex to parallelize



$A \times \Omega \sim O(n^2 k)$ if Ω is $n \times k$.

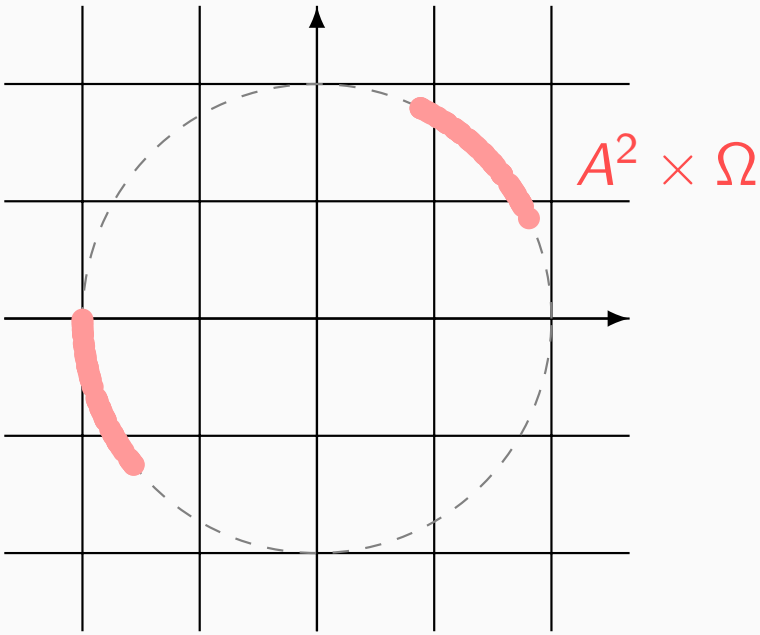
$\Rightarrow$ Approximating eigenvectors is *relatively cheap*.

Random linear algebra – background

Unsupervised learning:
We search for **eigencouples** of A, $n \times n$ matrix.

Two *limitations*:

- SVD is very costly $\sim O(n^3)$
- SVD is complex to parallelize



$A \times \Omega \sim O(n^2 k)$ if Ω is $n \times k$.

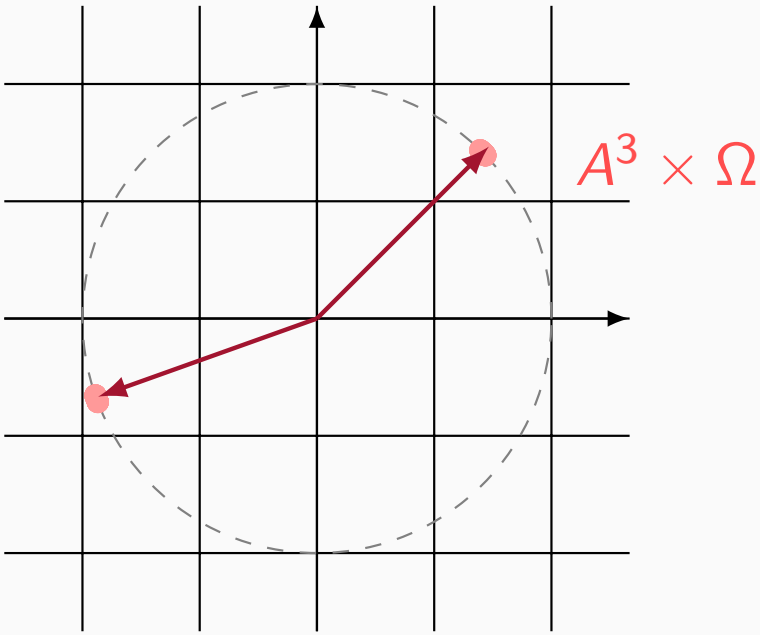
$\Rightarrow$ Approximating eigenvectors is *relatively cheap*.

Random linear algebra – background

Unsupervised learning:
We search for **eigencouples** of A, $n \times n$ matrix.

Two *limitations*:

- SVD is very costly $\sim O(n^3)$
- SVD is complex to parallelize



$A \times \Omega \sim O(n^2 k)$ if Ω is $n \times k$.

$\Rightarrow$ Approximating eigenvectors is *relatively cheap*.

Random linear algebra – background

Randomized SVD algorithm

```
D <- random_sample(n)
G <- gramian_matrix(D)
Ω <- random_matrix(n,k)
A <- GΩ
Q, R <- qr(A)
Y <- Q^T G
U, Σ, V <- svd(Y)
```

Some properties

G is a symmetric matrix.

The most costly operations are

- $G\Omega \sim O(n^2 k)$
- $Q^T G \sim O(n^2 k)$

Tradeoffs between `GEMM` and `SYMM`:

- `SYMM` communicates twice more than `GEMM`
  - A given column of G is stored over $p + q$ nodes instead of p with 2DBC.
- `GEMM` consumes twice more memory than `SYMM`
  - The whole matrix G is allocated.

Unless ...

- New distributions are discovered (they have been!)
- Our programming model adapts to any distributions (it does !)

Random linear algebra – background

Randomized SVD algorithm

```
D <- random_sample(n)
G <- gramian_matrix(D)
Ω <- random_matrix(n,k)
A <- GΩ
Q, R <- qr(A)
Y <- Q^T G
U, Σ, V <- svd(Y)
```

Some properties

G is a symmetric matrix.

The most costly operations are

- $G\Omega \sim O(n^2 k)$
- $Q^T G \sim O(n^2 k)$

Tradeoffs between `GEMM` and `SYMM`:

- `SYMM` communicates twice more than `GEMM`
  - A given column of G is stored over $p + q$ nodes instead of p with 2DBC.
- `GEMM` consumes twice more memory than `SYMM`
  - The whole matrix G is allocated.

Unless ...

- New distributions are discovered (they have been!)
- Our programming model adapts to any distributions (it does !)

Random linear algebra – background

Randomized SVD algorithm

```
D <- random_sample(n)
G <- gramian_matrix(D)
Ω <- random_matrix(n,k)
A <- GΩ
Q, R <- qr(A)
Y <- Q^T G
U, Σ, V <- svd(Y)
```

Some properties

G is a symmetric matrix.

The most costly operations are

- $G\Omega \sim O(n^2 k)$
- $Q^T G \sim O(n^2 k)$

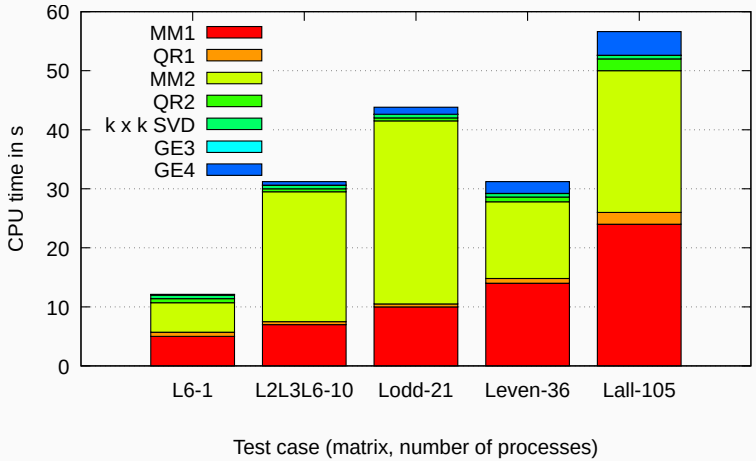
Tradeoffs between `GEMM` and `SYMM`:

- `SYMM` communicates twice more than `GEMM`
  - A given column of G is stored over $p + q$ nodes instead of p with 2DBC.
- `GEMM` consumes twice more memory than `SYMM`
  - The whole matrix G is allocated.

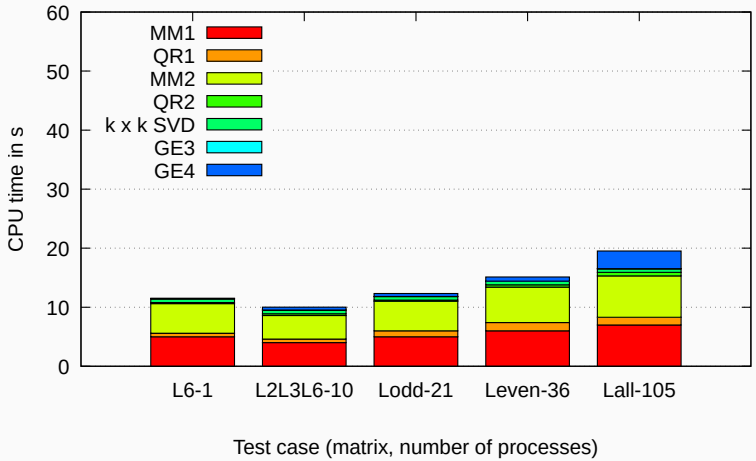
Unless ...

- New distributions are discovered (they have been!)
- Our programming model adapts to any distributions (it does !)

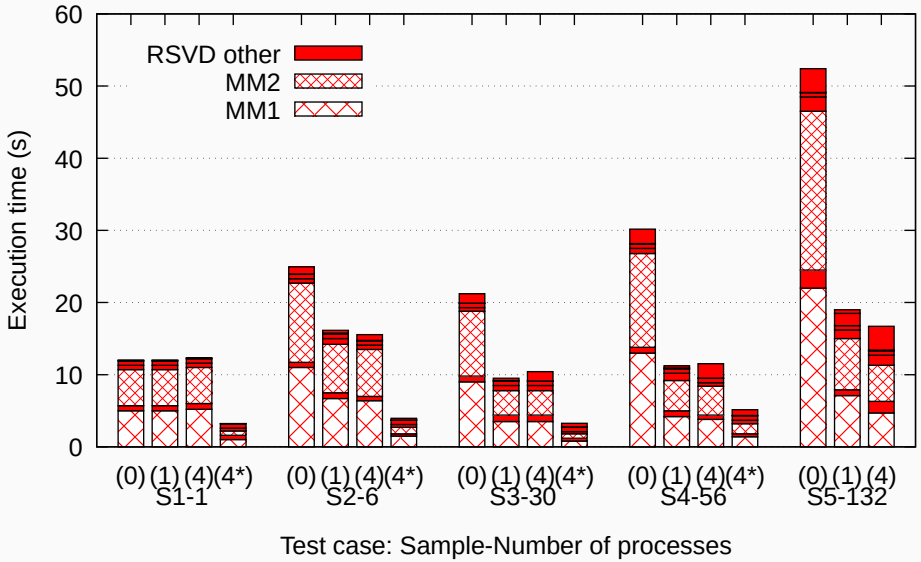
Random linear algebra – experiments



Legacy algorithm



A-stationary SYMM



- (0) Legacy algorithm
- (1) A-stationary GEMM
- (4) A-stationary SYMM
- (4*) A-stationary SYMM + GPU

CONCLUSIONS

Contributions

- Show **the STF programming model** can be suitably **extended** to **express** a complex distributed-memory algorithm
  - **Portable**, easily **maintainable**
  - Independant of matrices' distribution, workflow's scheduling
  - **Competitive** with reference libraries

Technical report: `https://hal.inria.fr/hal-03588491/`

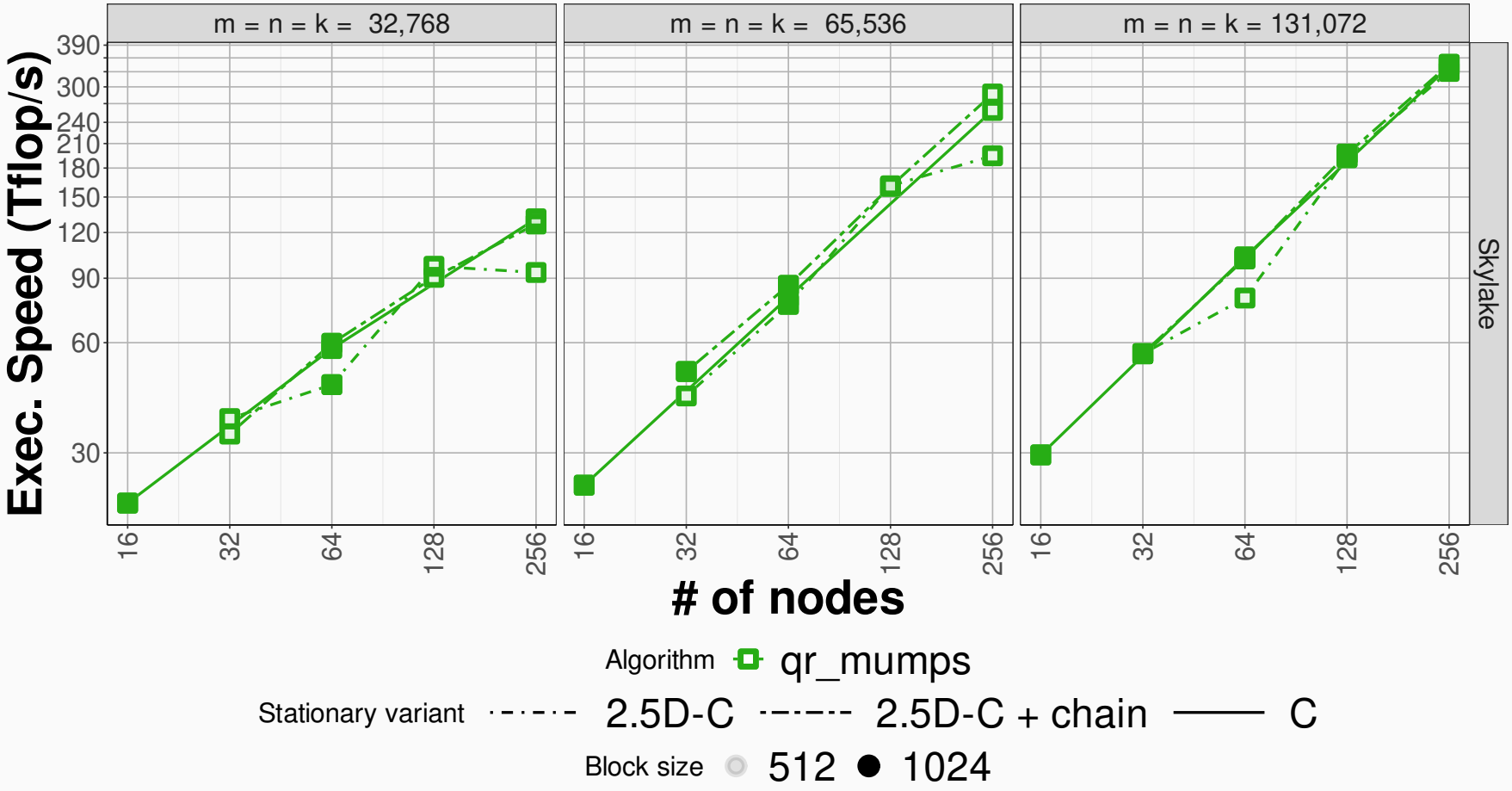
On-going work

- Implementing factorizations algorithms
  - How to replicate tasks ? How to setup alternative sources for a block ?
- Studying the scheduling of communication patterns in GEMM
  - In collaboration with O. Beaumont, A. Denis, L. Eyraud-Dubois, E. Jeannot, M. Vérité, P. Swartvagher

ON-GOING WORK

Extensive study on communication ordering for GEMM

Experiments questionning the scheduling of communications



⇒ In collaboration with:

Olivier Beaumont, Alexandre Denis, Lionel Eyraud-Dubois, Emmanuel Jeannot, Mathieu Vérité, Philippe Swartvagher

Dense Factorizations (LU, Cholesky)

2.5D factorization algorithms have **more complex** critical path.

They require more advanced key functionalities:

1. Accumulation and reduction of **updates over layers**
2. **Replication** of factorization tasks
3. **All-reduce pattern** to implement tournament pivoting

The programming model currently lacks support for 2. and 3.

Dense Factorizations (LU, Cholesky)

2.5D factorization algorithms have **more complex** critical path.

They require more advanced key functionalities:

1. Accumulation and reduction of **updates over layers**
2. **Replication** of factorization tasks
3. **All-reduce pattern** to implement tournament pivoting

The programming model currently lacks support for 2. and 3.

BACKUP SLIDES

Practical expression

Some common operations are done before switching on the `stat` parameter:

- Binding C to the `init` and `sum` codelets
- (submitting) the scaling of C by β

2.5D-A-Stat

```
do jh=1,n/h; do layer=1,h
  j = jh * layer
  do i=1,m
    do l=1,k
Ail => transa ? A_{l,i} : A_{i,l}
Blj => transb ? B_{j,l} : B_{l,j}
Cij => C_{i,j}
work_node = Ail%owner
          + r*c*layer
call insert_task( gemm,
  work_node:ON_NODE,
  Ail:R,Blj:R,
  Cij:RANK_REDUX )
    end do
    call mpi_redux_tree(C_{i,j})
  end do
end do; end do
```

2.5D-B-Stat

```
do ih=1,m/h; do layer=1,h
  i = ih * layer
  do j=1,n
    do l=1,k
Ail => transa ? A_{l,i} : A_{i,l}
Blj => transb ? B_{j,l} : B_{l,j}
Cij => C_{i,j}
work_node = Blj%owner
          + r*c*layer
call insert_task( gemm,
  work_node:ON_NODE,
  Ail:R,Blj:R,
  Cij:RANK_REDUX )
    end do
    call mpi_redux_tree(C_{i,j})
  end do
end do; end do
```

2.5D-C-Stat

```
do lh=1,k/h; do layer=1,h
  l = lh * layer
  do i=1,m
    do j=1,n
Ail => transa ? A_{l,i} : A_{i,l}
Blj => transb ? B_{j,l} : B_{l,j}
Cij => C_{i,j}
work_node = Cij%owner
          + r*c*layer
call insert_task( gemm,
  work_node:ON_NODE,
  Ail:R,Blj:R,
  Cij:RANK_REDUX )
    end do
  end do
end do; end do
do i=1,m; do j=1,n
  call mpi_redux_tree(C_{i,j})
end do; end do
```

StarPU – additional features ?

Some additional features to StarPU have been considered.

- Automatically insert reduction tasks
  - Further simplify factorization implementations

```
call starpu_mpi_redux(A_{I,I})
call insert_task(potrf, A_{I,I}:RW)
```

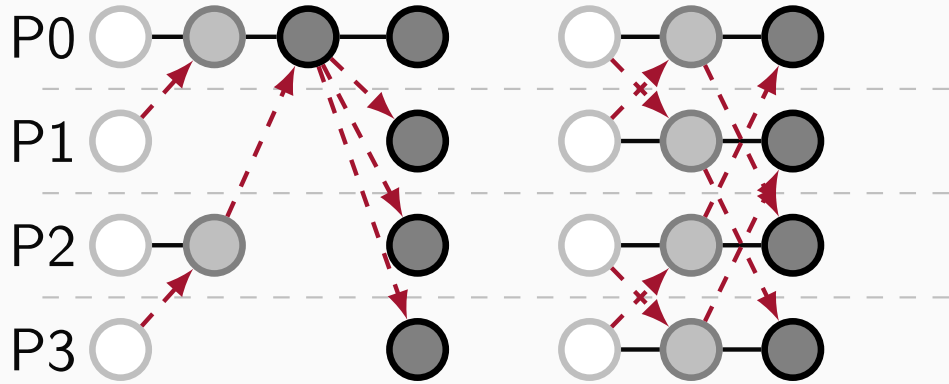
⇓

```
! implicit reduction
call insert_task(potrf, A_{I,I}:RW)
```

- Allow tasks replication across nodes
  - Required by state-of-the-art algorithms
  - Each resulting tile can be sent to different subsets of nodes

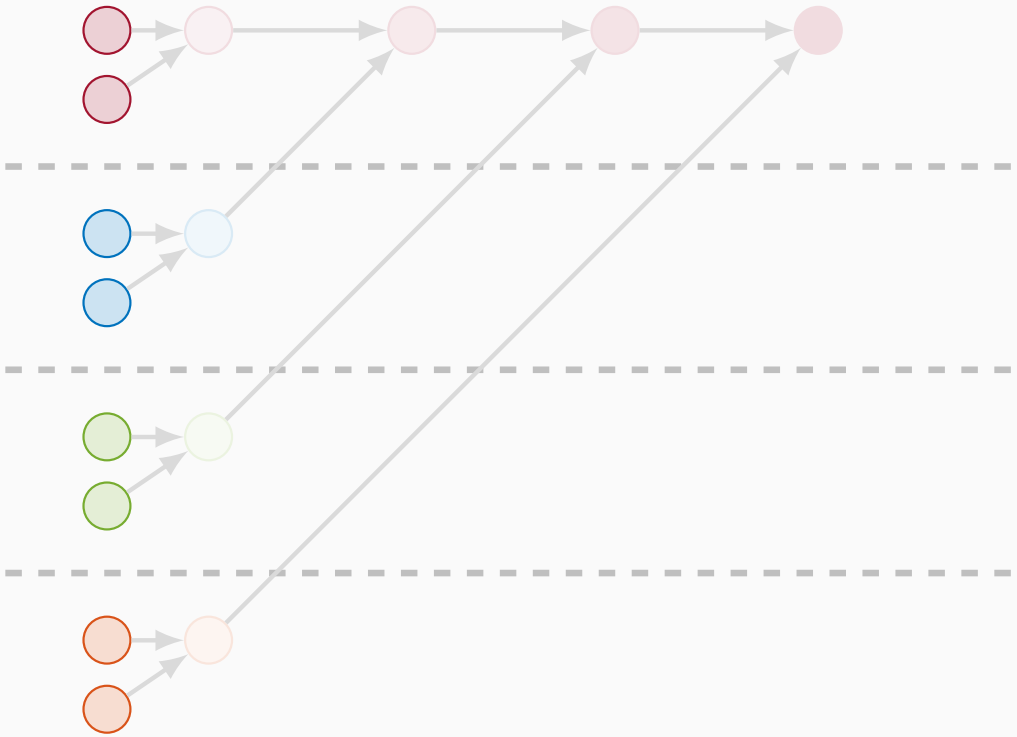
```
call insert_task(potrf, A_{I,I}:RW,
                 ON_NODES:task_map[I,I,:])
call insert_task(trsm, A_{I,I}:R, A_{i,I}:RW,
                 FROM_NODE:task_map[I,I,i%h])
```

- Allow all-reduce pattern
  - Shorten the critical path as replicated data is already available



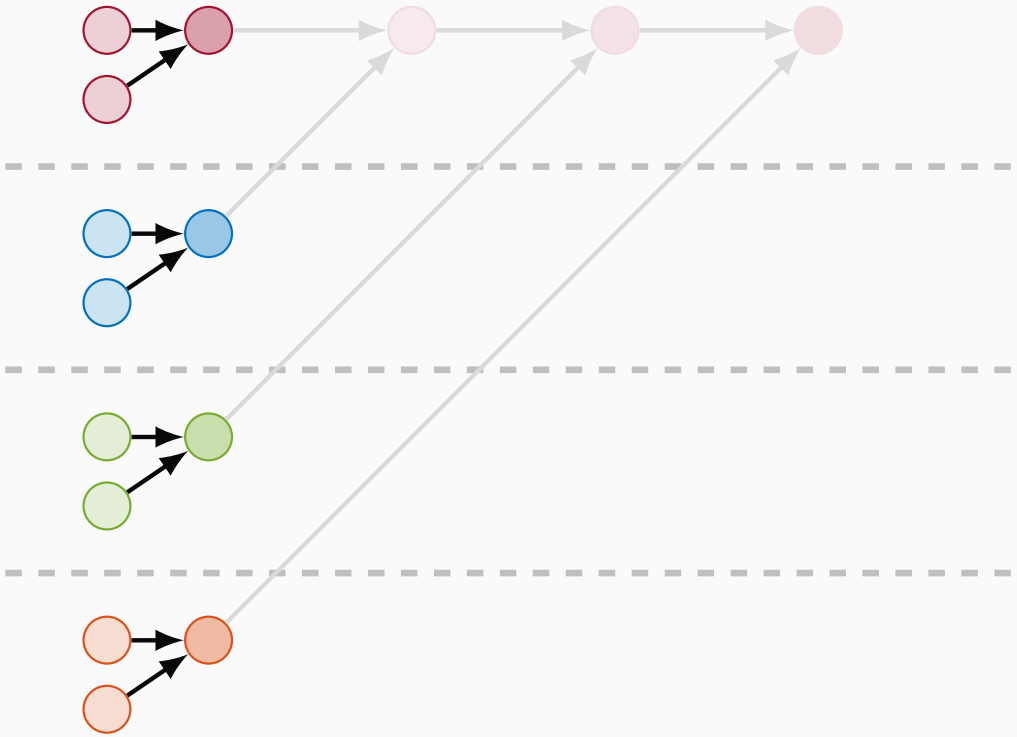
Features of StarPU – 1.3

- `STARPU_REDUX`
  - One contribution per **core**
    - Intra-node reduction through binary tree
  - Remain available in StarPU (master)
- `starpu_mpi_redux_data`
  - **All** nodes in `MPI_COMM_World` contribute through a **flat** tree
  - **Blocking** method



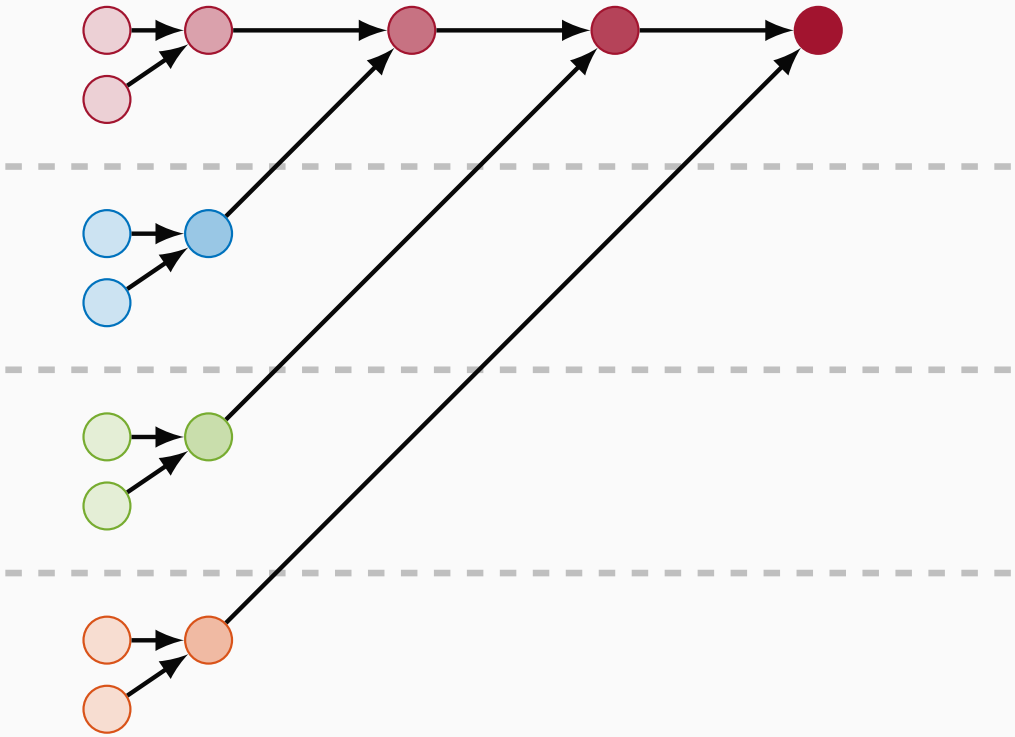
Features of StarPU – 1.3

- `STARPU_REDUX`
  - One contribution per **core**
    - Intra-node reduction through binary tree
  - Remain available in StarPU (master)
- `starpu_mpi_redux_data`
  - **All** nodes in `MPI_COMM_World` contribute through a **flat** tree
  - **Blocking** method



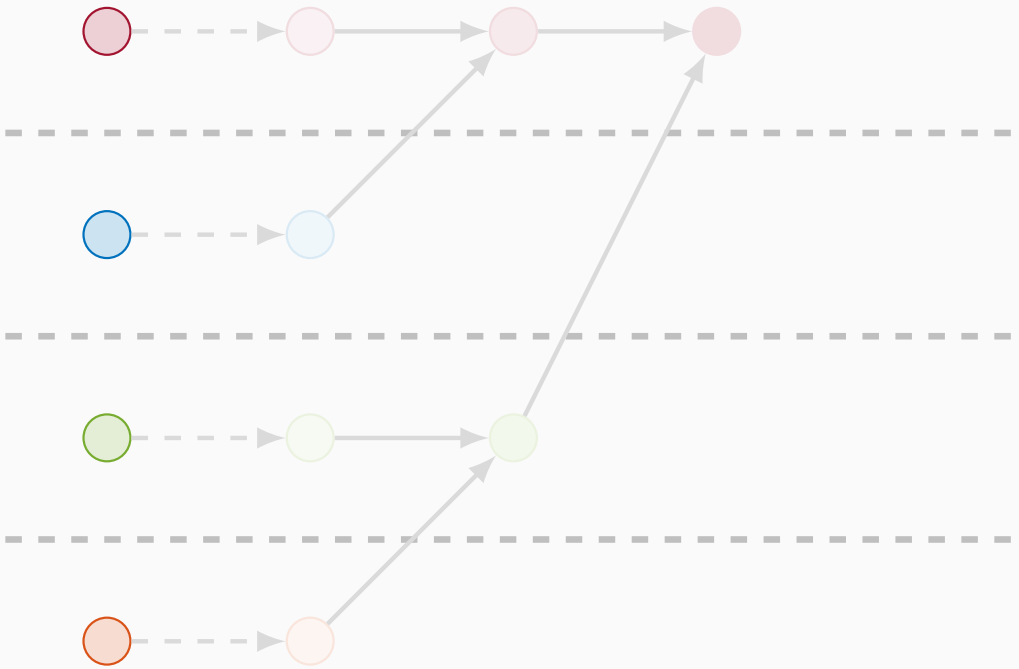
Features of StarPU – 1.3

- `STARPU_REDUX`
  - One contribution per **core**
    - Intra-node reduction through binary tree
  - Remain available in StarPU (master)
- `starpu_mpi_redux_data`
  - **All** nodes in `MPI_COMM_World` contribute through a **flat** tree
  - **Blocking** method



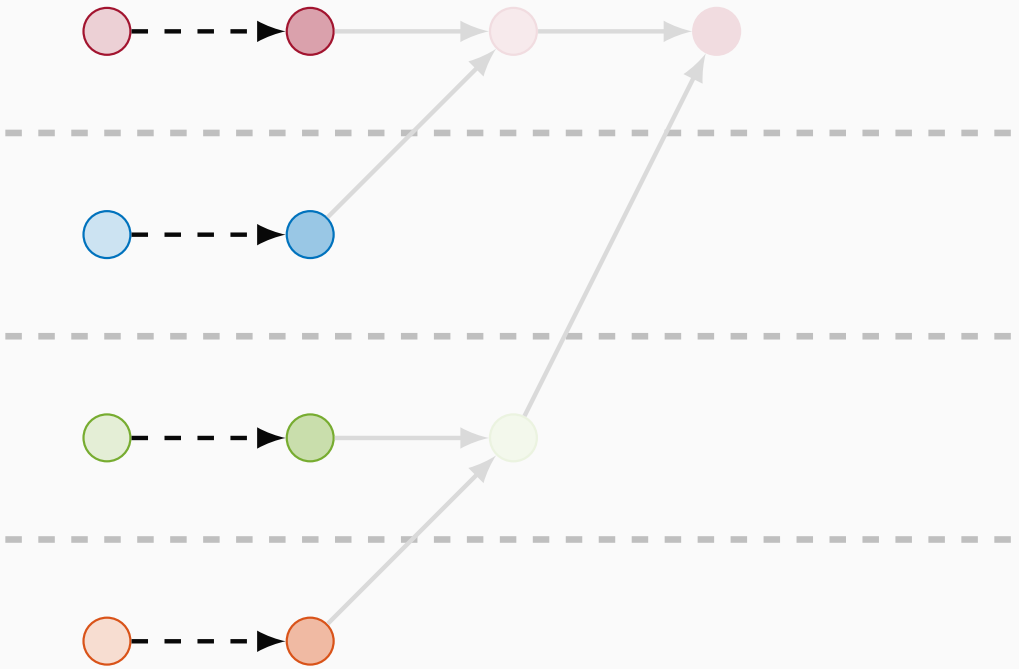
Features of StarPU – our contribution

- `STARPU_MPI_REDUX` (**Feature!** issue 3)
  - One contribution per **contributing node**
  - Interpreted as `STARPU_RW|STARPU_COMMUTE`
- `starpu_mpi_redux_data_tree` (**Enhancement!** merge req. 21)
  - All **contributing** nodes join in the reduction through a **n-ary** tree
  - **Non-blocking** method



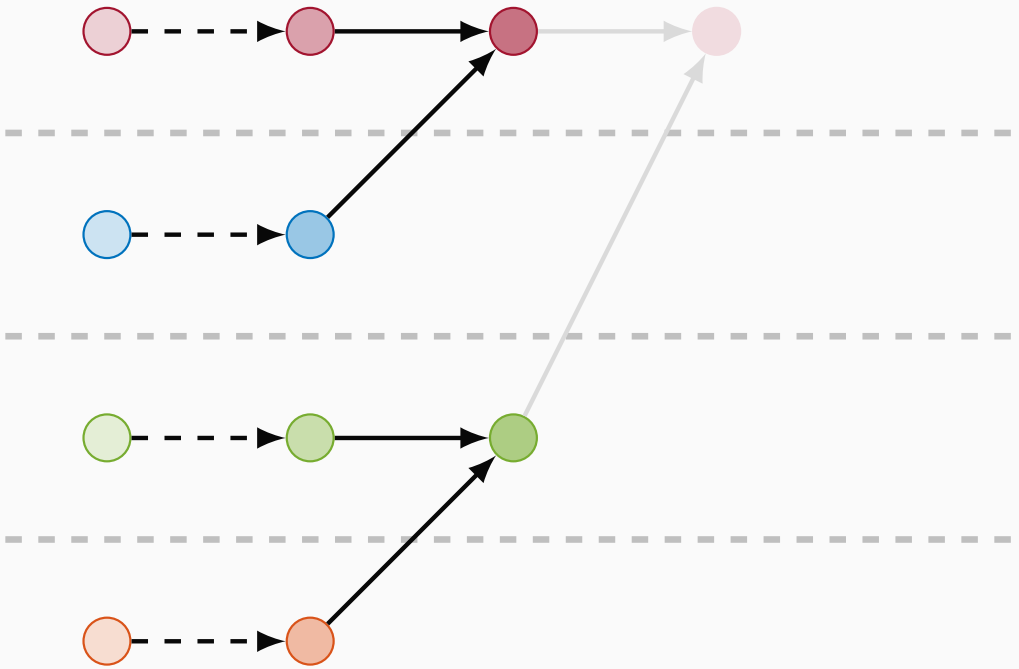
Features of StarPU – our contribution

- `STARPU_MPI_REDUX` (**Feature!** issue 3)
  - One contribution per **contributing node**
  - Interpreted as `STARPU_RW|STARPU_COMMUTE`
- `starpu_mpi_redux_data_tree` (**Enhancement!** merge req. 21)
  - All **contributing** nodes join in the reduction through a **n-ary** tree
  - **Non-blocking** method



Features of StarPU – our contribution

- `STARPU_MPI_REDUX` (**Feature!** issue 3)
  - One contribution per **contributing node**
  - Interpreted as `STARPU_RW|STARPU_COMMUTE`
- `starpu_mpi_redux_data_tree` (**Enhancement!** merge req. 21)
  - All **contributing** nodes join in the reduction through a **n-ary** tree
  - **Non-blocking** method



Features of StarPU – our contribution

- `STARPU_MPI_REDUX` (**Feature!** issue 3)
  - One contribution per **contributing node**
  - Interpreted as `STARPU_RW|STARPU_COMMUTE`
- `starpu_mpi_redux_data_tree` (**Enhancement!** merge req. 21)
  - All **contributing** nodes join in the reduction through a **n-ary** tree
  - **Non-blocking** method