Rethinking HPC algorithms for Exascale: the case for Adjoint Computations

Guillaume Aupy

VANDERBILT UNIVERSITY

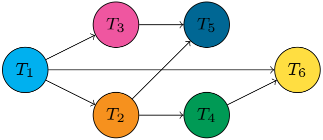
Scheduling 102,
Scheduling under constraints

Guillaume Aupy

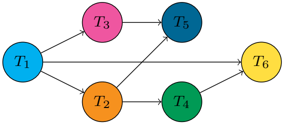
VANDERBILT
UNIVERSITY

Scheduling

- p processors (or nodes or cores or computing units).
- An application represented by a DAG $\mathcal{G} = (V, E)$:
  - Vertices are tasks (or functions to be computed)
  - Edges represent data dependency (need to be respected)

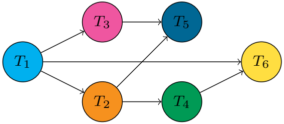


Motivational example

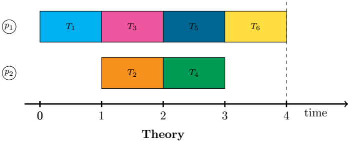


Assume tasks have unit size and there are two processors.

Motivational example

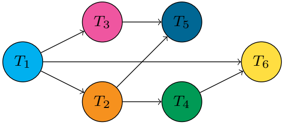


Assume tasks have unit size and there are two processors.

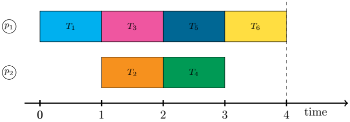


- In theory, optimal schedule ☺

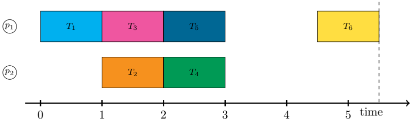
Motivational example



Assume tasks have unit size and there are two processors.



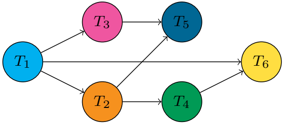
Theory



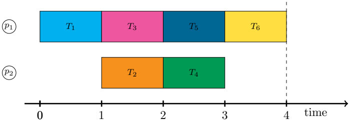
Practice

- In theory, optimal schedule ☺
- In practice, not what we expected ☹.

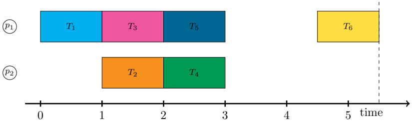
Motivational example



Assume tasks have unit size and there are two processors.



Theory

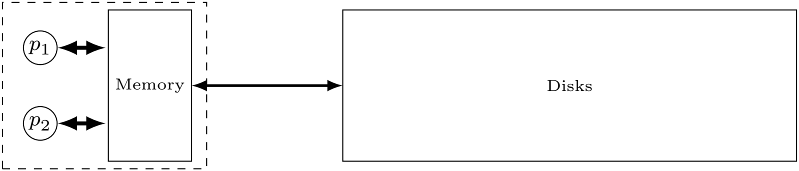


Practice

- In theory, optimal schedule ☺
- In practice, not what we expected ☹.

What happened?

Data is critical

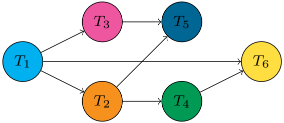


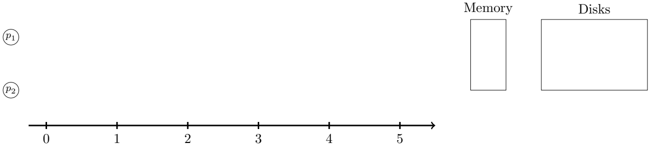
Overview of a computer.

In general,

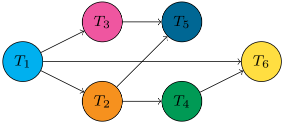
- Memory is small but accesses are fast;
- Disks are large but accesses are slow.

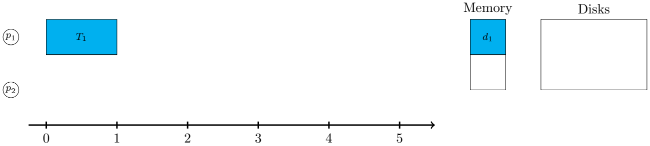
Back to our schedule



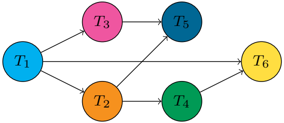


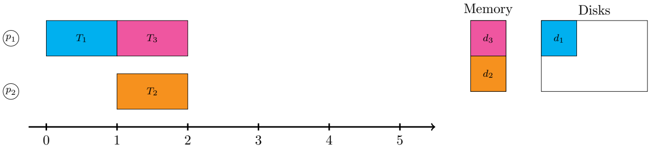
Back to our schedule



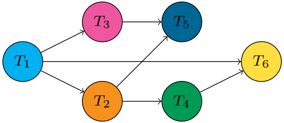


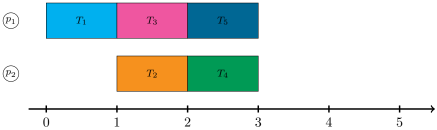
Back to our schedule

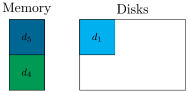




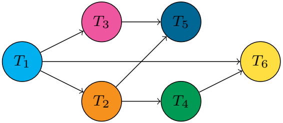
Back to our schedule

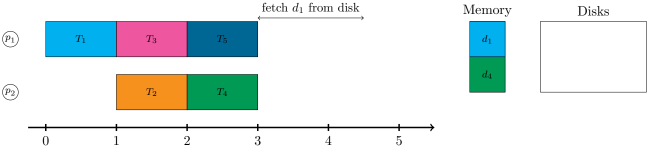




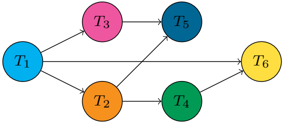


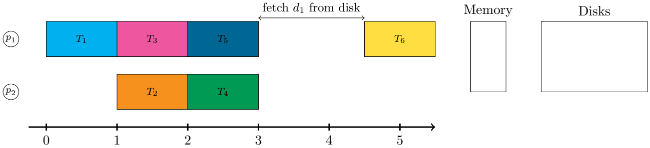
Back to our schedule



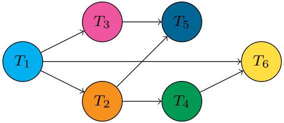


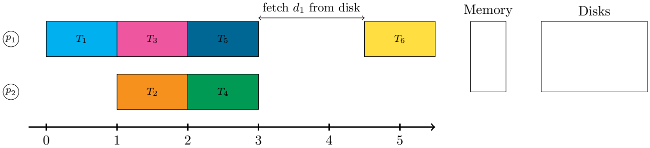
Back to our schedule





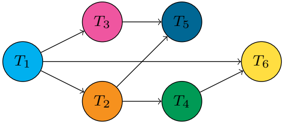
Back to our schedule

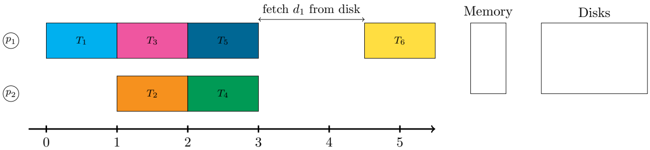




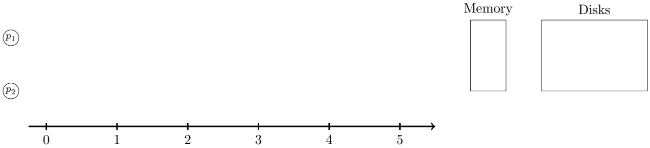
Can we do better (or prove that we cannot)?

Back to our schedule

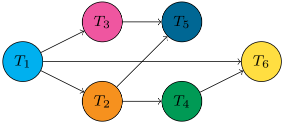


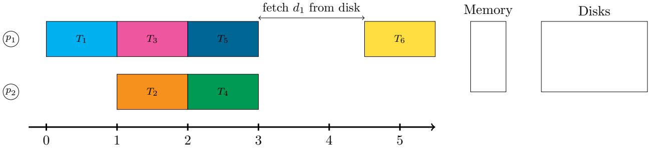


Can we do better (or prove that we cannot)?

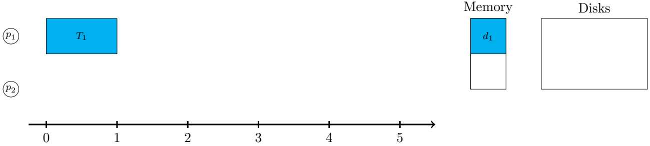


Back to our schedule

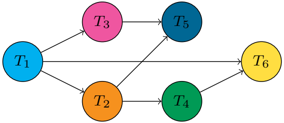


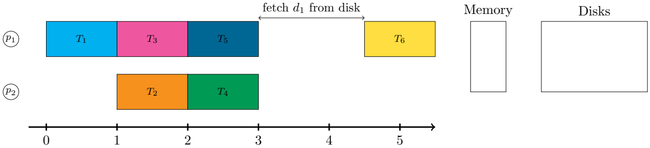


Can we do better (or prove that we cannot)?

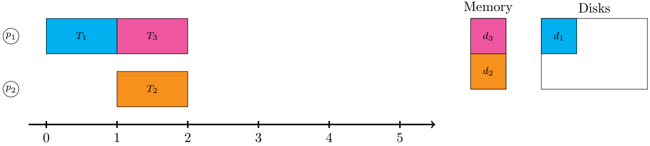


Back to our schedule

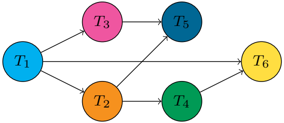


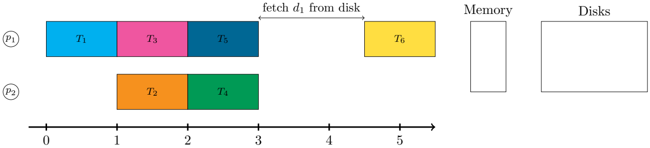


Can we do better (or prove that we cannot)?

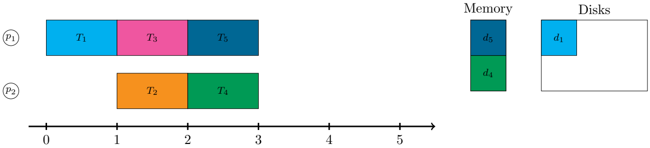


Back to our schedule

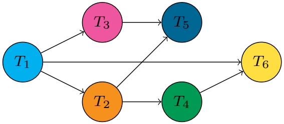


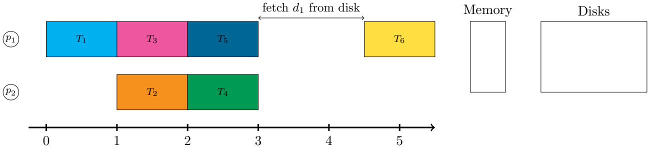


Can we do better (or prove that we cannot)?

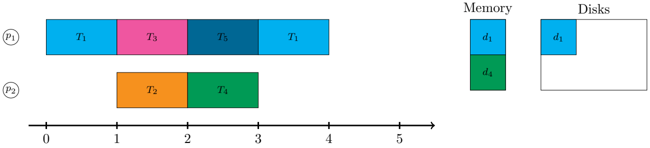


Back to our schedule

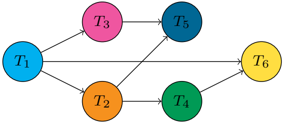


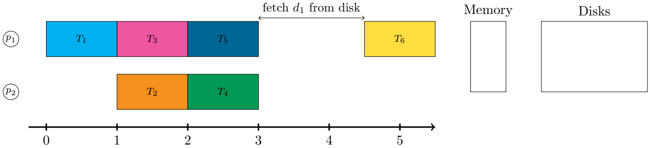


Can we do better (or prove that we cannot)?

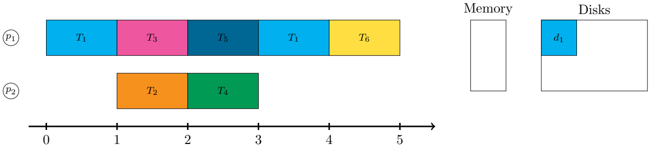


Back to our schedule





Can we do better (or prove that we cannot)?



Some numbers

(Brief) history of supercomputers at Argonne National Lab:

	Intrepid	Mira
Year	2008-2013	2013-
Peak Perf	0.557 PFlops	10 PFlops
Peak I/O Throughput	88 GB/s	240 GB/s

Some numbers

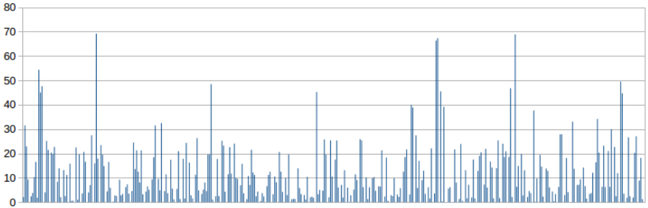
(Brief) history of supercomputers at Argonne National Lab:

	Intrepid	Mira
Year	2008-2013	2013-
Peak Perf	0.557 PFlops	10 PFlops
Peak I/O Throughput	88 GB/s	240 GB/s
I/O Tp per Flops	157 GB/PFlops/s	16.8 GB/PFlops/s

Some numbers

(Brief) history of supercomputers at Argonne National Lab:

	Intrepid	Mira
Year	2008-2013	2013-
Peak Perf	0.557 PFlops	10 PFlops
Peak I/O Throughput	88 GB/s	240 GB/s
I/O Tp per Flops	157 GB/PFlops/s	16.8 GB/PFlops/s



Analysis of the Intrepid system @Argonne: I/O throughput decrease (percentage per application, over 400 applications).

What next

Some directions to solve this problem:

- Better I/O Management?
- Rethinking I/O intensive applications:
  from computation-oriented thinking to I/O-oriented thinking.

Optimal multistage algorithms for adjoint computation

Guillaume Aupy,
work with Julien Herrmann, Paul Hovland & Yves Robert

VANDERBILT
UNIVERSITY

Ice-sheet model (I)

"In climate modelling, Ice-sheet models use numerical methods to simulate the evolution, dynamics and thermodynamics of ice sheets." (wikipedia)

Model Algorithm (single timestep)

1. Evaluate driving stress $\tau_d = \rho g h \nabla s$
2. Solve for velocities

   DO i = 1, max_iter

   i. Evaluate nonlinear viscosity v_i from iterate $\boldsymbol{u}_i$

   ii. Construct stress matrix $A\{v_i\}$

   iii. Solve linear system $A\ \boldsymbol{u}_{i+1} = \tau_d$

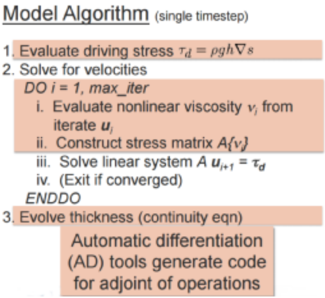
   iv. (Exit if converged)

   ENDDO
3. Evolve thickness (continuity eqn)

Automatic differentiation (AD) tools generate code for adjoint of operations

Credit: Daniel Goldberg

“In climate modelling, Ice-sheet models use numerical methods to simulate the evolution, dynamics and thermodynamics of ice sheets.” (wikipedia)



Simpler Version:

proc Model Algorithm($u_0, \boldsymbol{y}$)
begin
  Do stuff;
  for $i = 0$ *to* n **do**
    $u_{i+1} = f_i(u_i)$;
    Do stuff;
  end
  /* $F(u_0) = f_n \circ f_{n-1} \circ \ldots \circ f_0(u_0)$ */
  Compute $\boldsymbol{\nabla} F(u_0)\boldsymbol{y}$;
end

Credit: Daniel Goldberg

Ice-sheet model (II)

$$F(u_0) = f_n \circ f_{n-1} \circ \ldots \circ f_1 \circ f_0(u_0)$$

$$\begin{aligned} \boldsymbol{\nabla} F(u_0)\boldsymbol{y} &= Jf_0(u_0)^T \cdot \boldsymbol{\nabla}(f_n \circ f_1)(u_1) \cdot \boldsymbol{y} \\ &= \boxed{Jf_0(u_0)^T \cdot Jf_1(u_1)^T \cdot \ldots \cdot Jf_{n-1}(u_{n-1})^T \cdot Jf_n(u_n)^T \cdot \boldsymbol{y}} \end{aligned}$$

$$\begin{aligned} Jf^T &= \text{ Transpose Jacobian matrix of } f; \\ u_{i+1} &= f_i(u_i) = f_i\left(f_{i-1} \circ \ldots \circ f_0(u_0)\right). \end{aligned}$$

Ice-sheet model (II)

$$F(u_0) = f_n \circ f_{n-1} \circ \ldots \circ f_1 \circ f_0(u_0)$$

$$\begin{aligned} \boldsymbol{\nabla} F(u_0)\boldsymbol{y} &= Jf_0(u_0)^T \cdot \boldsymbol{\nabla}(f_n \circ f_1)(u_1) \cdot \boldsymbol{y} \\ &= \boxed{Jf_0(u_0)^T \cdot Jf_1(u_1)^T \cdot \ldots \cdot Jf_{n-1}(u_{n-1})^T \cdot Jf_n(u_n)^T \cdot \boldsymbol{y}} \end{aligned}$$

$$\begin{aligned} Jf^T &= \text{ Transpose Jacobian matrix of } f; \\ u_{i+1} &= f_i(u_i) = f_i\left(f_{i-1} \circ \ldots \circ f_0(u_0)\right). \end{aligned}$$

But then, isn't there a faster algorithm?

A better solution?

$$\boldsymbol{\nabla} F(u_0)\boldsymbol{y} = Jf_0(u_0)^T \cdot Jf_1(u_1)^T \cdot \ldots \cdot Jf_{n-1}(u_{n-1})^T \cdot Jf_n(u_n)^T \cdot \boldsymbol{y}$$

```
proc Algo A(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
    end
    Compute ∇F(u_0)y;
end
```

```
proc Algo B(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
        v_{i+1} = v_i · Jf_{i+1}(u_{i+1})^T;
    end
end
```

A better solution?

$$\nabla F(u_0)\boldsymbol{y} = Jf_0(u_0)^T \cdot Jf_1(u_1)^T \cdot \ldots \cdot Jf_{n-1}(u_{n-1})^T \cdot Jf_n(u_n)^T \cdot \boldsymbol{y}$$

```
proc Algo A(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
    end
    Compute ∇F(u_0)y;
end
```

```
proc Algo B(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
        v_{i+1} = v_i · Jf_{i+1}(u_{i+1})^T;
    end
end
```

What is the problem with Algo B?

A BETTER SOLUTION?

$$\nabla F(u_0)\boldsymbol{y} = Jf_0(u_0)^T \cdot Jf_1(u_1)^T \cdot \ldots \cdot Jf_{n-1}(u_{n-1})^T \cdot Jf_n(u_n)^T \cdot \boldsymbol{y}$$

```
proc Algo A(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
    end
    Compute ∇F(u_0)y;
end
```

```
proc Algo B(u_0, y)
begin
    Do stuff;
    for i = 0 to n do
        u_{i+1} = f_i(u_i);
        Do stuff;
        v_{i+1} = v_i · Jf_{i+1}(u_{i+1})^T;
    end
end
```

What is the problem with Algo B?

$$\nabla F(u_0)\boldsymbol{y} = \left(\left(\ldots\left(Jf_0{}^T \cdot Jf_1{}^T\right) \cdot \ldots \cdot Jf_{n-1}{}^T\right) \cdot Jf_n{}^T\right) \cdot \boldsymbol{y} \quad n \text{ MatMat ops}$$

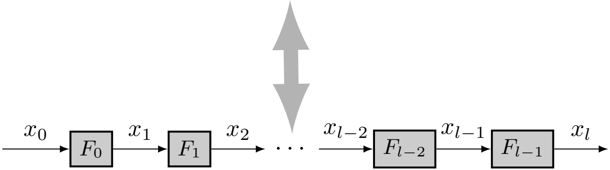
$$\nabla F(u_0)\boldsymbol{y} = Jf_0{}^T \cdot \left(Jf_1{}^T \cdot \ldots \cdot \left(Jf_{n-1}{}^T \cdot \left(Jf_n{}^T \cdot \boldsymbol{y}\right) \ldots\right)\right) \quad n \text{ MatVec ops}$$

Adjoint computation

$$\begin{aligned} F_i(x_i) &= x_{i+1} \quad && i < l \\ \bar{F}_i(x_i, \bar{x}_{i+1}) &= \bar{x}_i && i \le l \end{aligned}$$

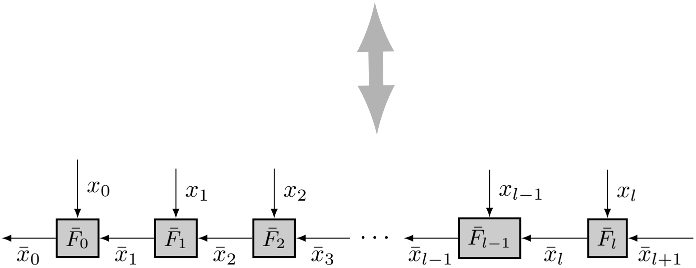
Adjoint computation

$$\boldsymbol{F_i(x_i) = x_{i+1} \quad i < l}$$
$$\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \qquad i \leq l$$



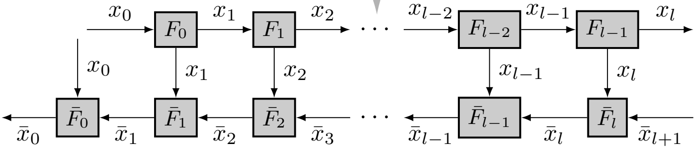
Adjoint computation

$$F_i(x_i) = x_{i+1} \quad i < l$$
$$\boldsymbol{\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \qquad i \leq l}$$



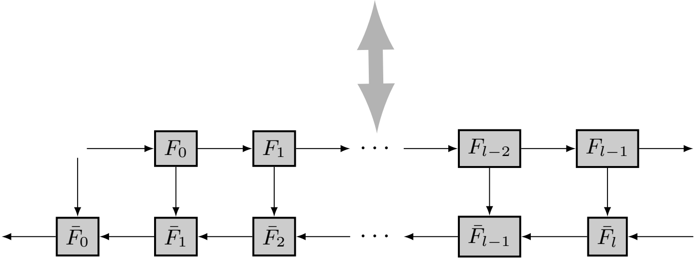
Adjoint computation

$$
\begin{aligned}
F_i(x_i) &= x_{i+1} \quad & i < l \\
\bar{F}_i(x_i, \bar{x}_{i+1}) &= \bar{x}_i \quad & i \leq l
\end{aligned}
$$

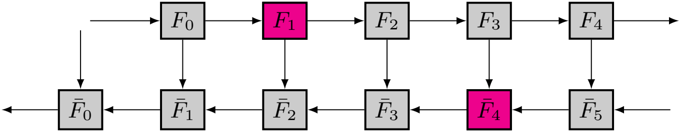


Adjoint computation

$$\begin{aligned} F_i(x_i) &= x_{i+1} & i < l \\ \bar{F}_i(x_i, \bar{x}_{i+1}) &= \bar{x}_i & i \le l \end{aligned}$$

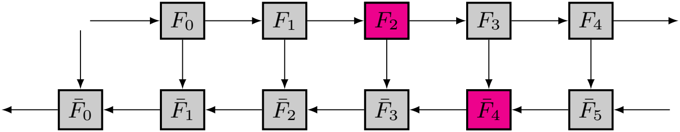


Model of Computation



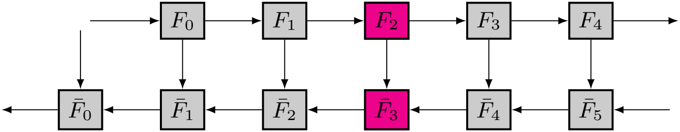
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



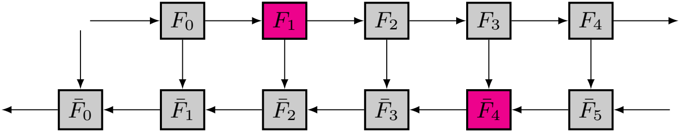
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



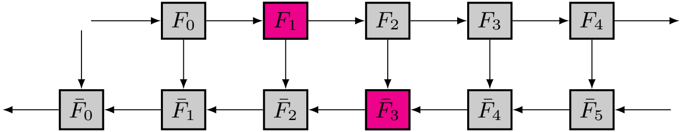
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



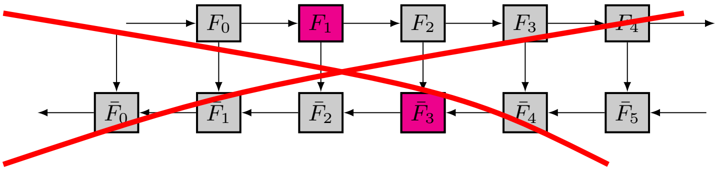
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



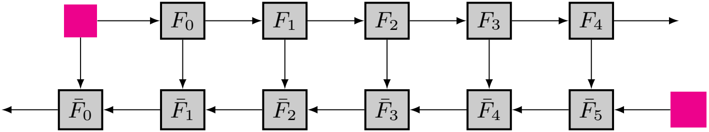
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



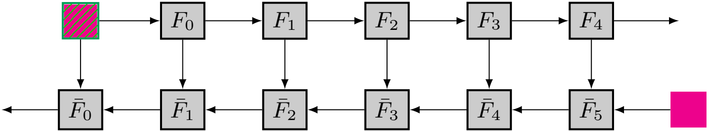
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



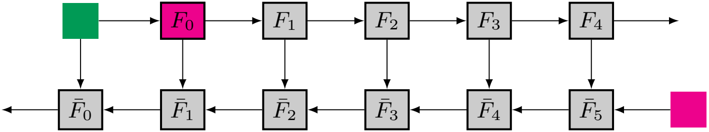
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.

Model of Computation



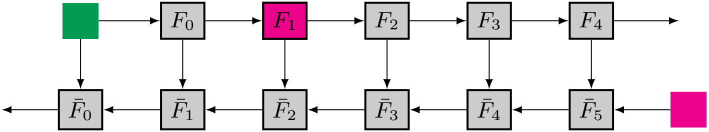
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
    - Cost to write: $w_m = 0$,
    - Cost to read: $r_m = 0$.

Model of Computation



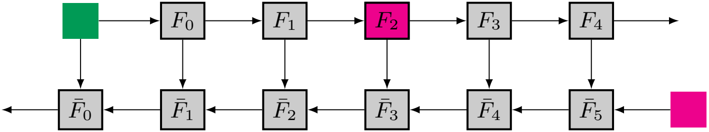
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



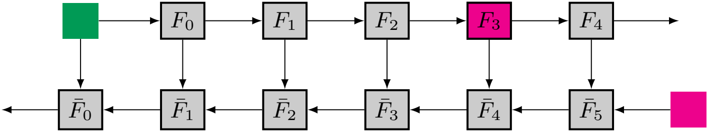
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



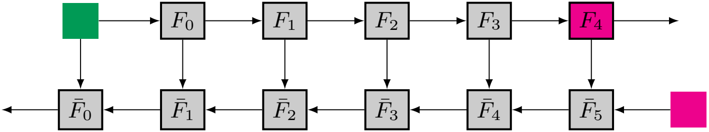
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



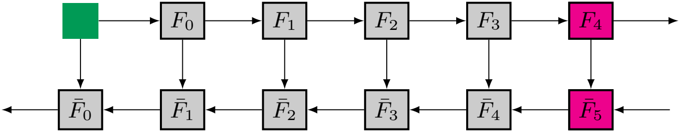
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



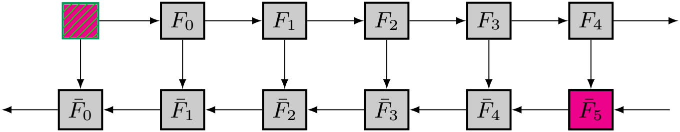
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



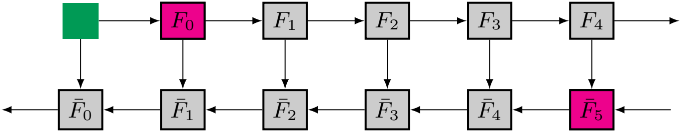
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation

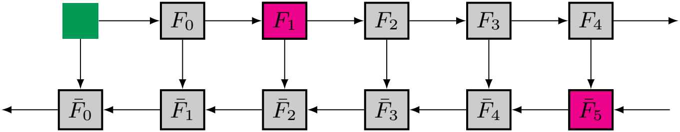


- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation

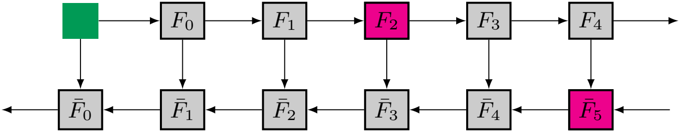


- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.



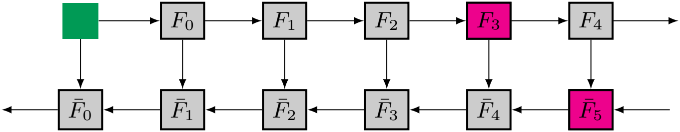
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



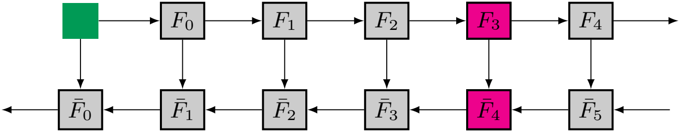
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



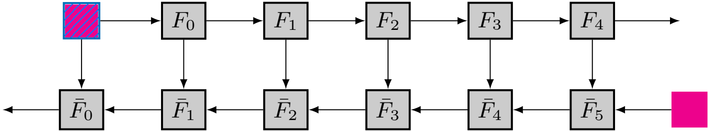
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



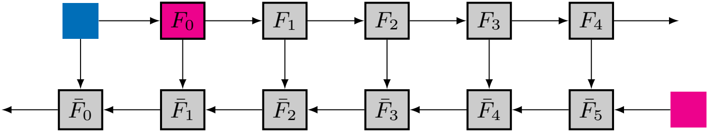
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.

Model of Computation



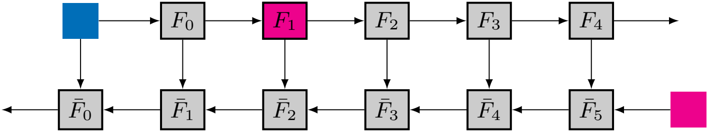
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



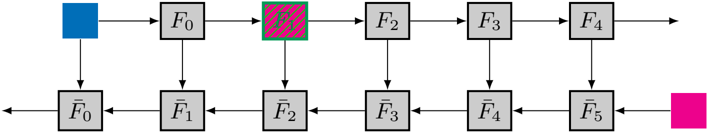
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



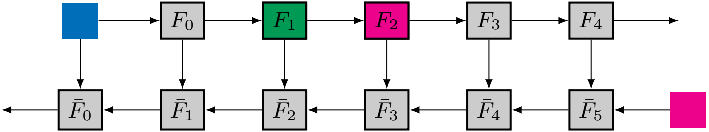
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



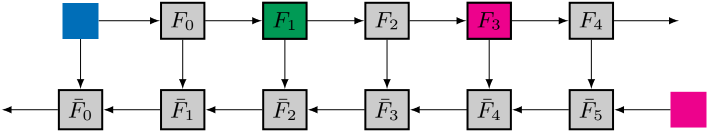
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



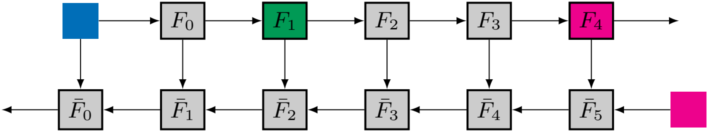
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



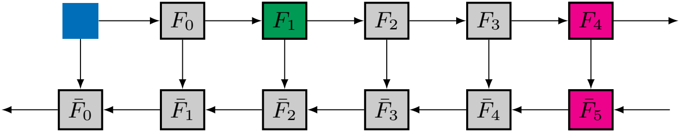
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



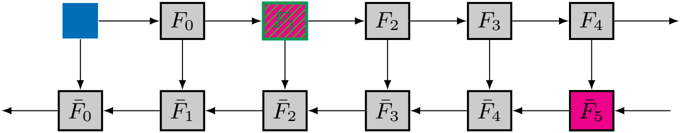
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



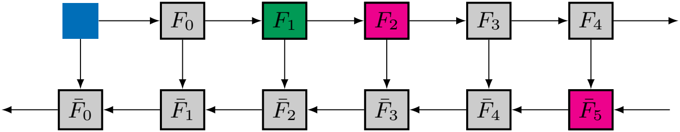
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



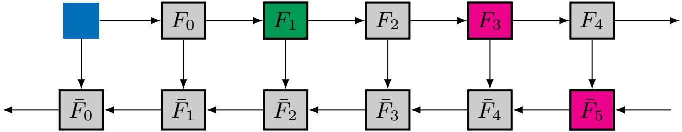
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



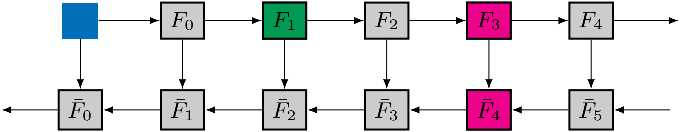
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



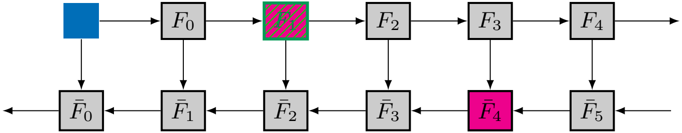
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



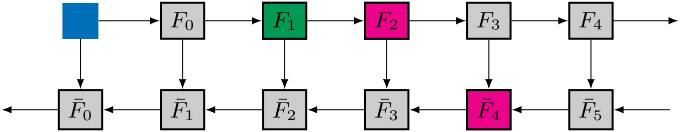
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



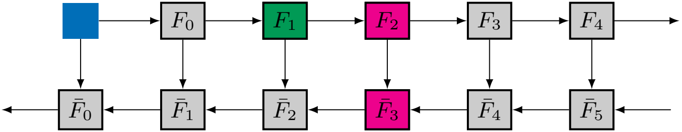
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
    - Cost to write: $w_m = 0$,
    - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
    - Cost to write: w_d,
    - Cost to read: r_d.

Model of Computation



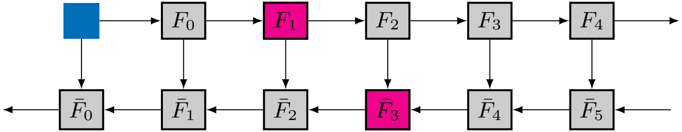
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



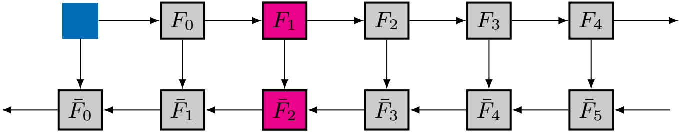
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



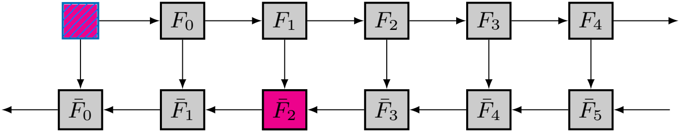
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



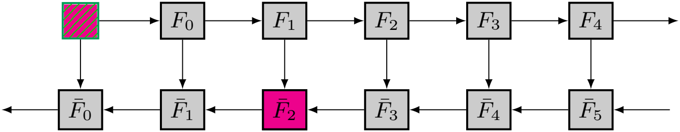
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



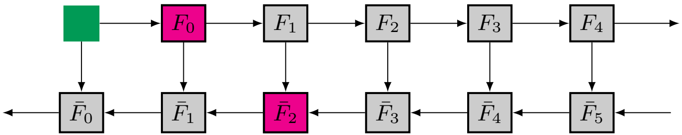
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



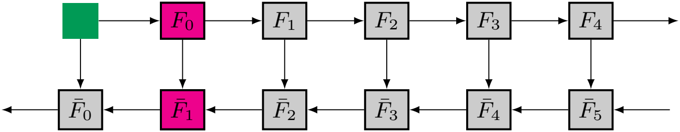
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



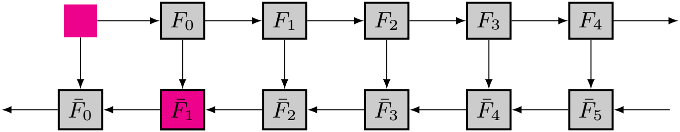
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



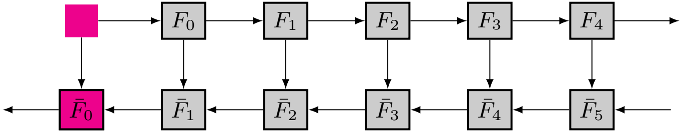
- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation



- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Model of Computation

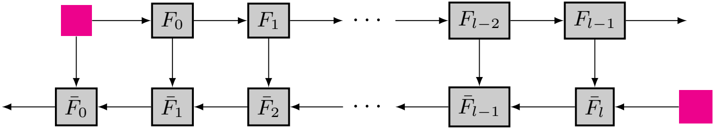


- Two buffers: store output of computations (x_i or $\bar{x}_i$). Initial state: contains x_0 and $\bar{x}_{l+1}$.
- $c_m < +\infty$ in-core slots (memory).
  - Cost to write: $w_m = 0$,
  - Cost to read: $r_m = 0$.
- $c_d = +\infty$ out-of-core slots (disk).
  - Cost to write: w_d,
  - Cost to read: r_d.

Problem formulation

We want to minimize the makespan of:

		Initial state:
AC graph:	size l	
Steps:	u_f, u_b	
Memory:	c_m, $w_m = r_m = 0$,	$\mathcal{M}_{\text{ini}} = \emptyset$
Disks:	$c_d = +\infty, w_d, r_d,$	$\mathcal{D}_{\text{ini}} = \emptyset$
Buffers:	$\mathcal{B}^{\top}$, $\mathcal{B}^{\perp}$	$\mathcal{B}^{\top}_{\text{ini}} = \{x_0\}$, $\mathcal{B}^{\perp}_{\text{ini}} = \{\bar{x}_{l+1}\}$



GW00: Revolve(l, c_m), optimal algorithm with c_m memory slots and no disk slots.

SW09: SWA*, an algorithm based on Revolve that takes disk storage into acount.

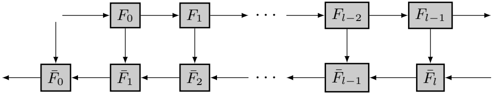
(i) $\mathrm{SWA}(l, c_m, c_d, w_d, r_d) \approx \textsc{Revolve}(l, c_d + c_m)$[1]

(ii) $\mathrm{SWA}^\star(l, c_m, w_d, r_d) = \min_{c_d=0\ldots l-c_m} \mathrm{SWA}(l, c_m, c_d, w_d, r_d)$

This work: optimal algorithm with disk storage.

[1] out of the $c_d + c_m$ slots used by Revolve, the c_d slots the least used are considered disk slots.

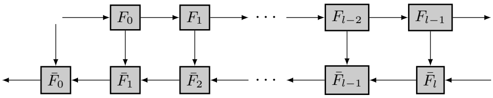
A grasp of the proof



Any algorithm works in two phases:

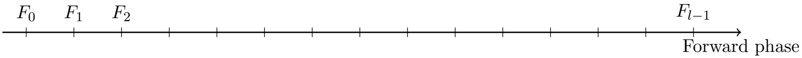
- The forward phase (before executing $\bar{F}_l$);
- The backward phase (that starts when executing $\bar{F}_l$);

The forward phase

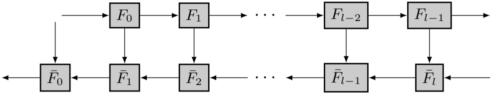


In this phase:

- We execute all F-operations;

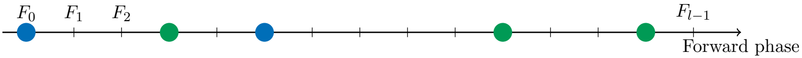


The forward phase

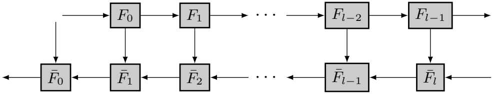


In this phase:

- We execute all F-operations;
- We write some data to disk and/or to memory.



The backward phase



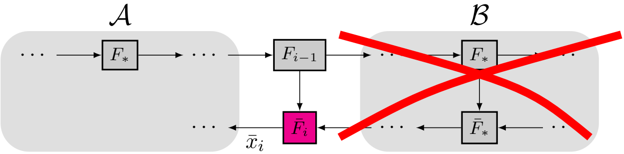
In this phase:

- We DO NOT write any data to disk (could have been done in the forward phase);
- All other operations are allowed.

No going back

Lemma

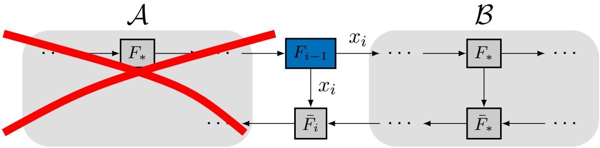
If $\bar{x}_i$ is computed, then there are no F_j for $i \leq j$ (or operations involving $\mathcal{B}$).



Checkpoint persistence (I)

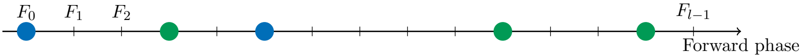
Lemma

If x_i is written (to disk or memory), until we have executed $\bar{F}_i$ there are no F_j for $j < i$ (or operations involving $\mathcal{A}$).



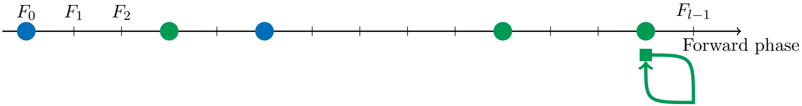
Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:



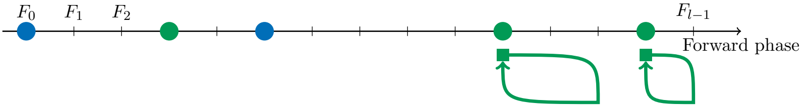
Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:



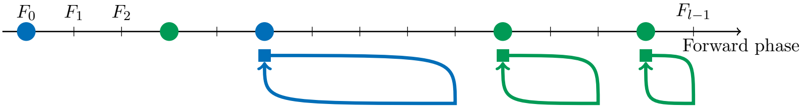
Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:



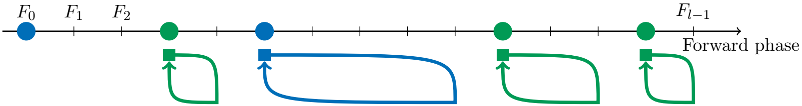
Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:



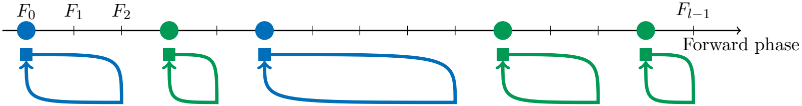
Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:

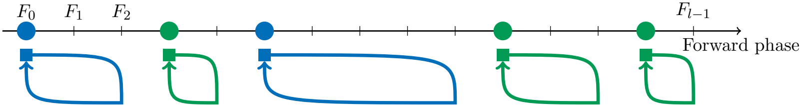


Checkpoint persistence (II)

In this case, for a given forward phase, we get a multi-phase backward phase:



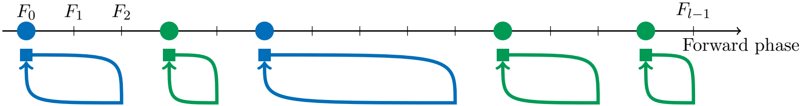
Characterizing the backward phase



- m backward steps to execute;
- No disk writes or reads;
- c memory checkpoints available

$\Longrightarrow r_m + \text{Revolve}(m, c)$

Characterizing the backward phase



- m backward steps to execute;
- No disk writes or reads;
- c memory checkpoints available

$\Longrightarrow r_m + \text{Revolve}(m, c)$

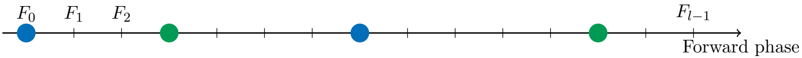
- m backward steps to execute;
- No disk writes;
- c memory checkpoints available

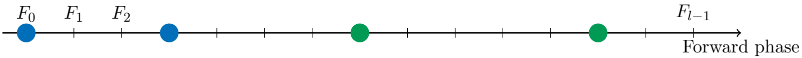
$\Longrightarrow r_d + \text{1D-Revolve}(m, c)$

Characterizing the forward phase

Theorem

During the forward phase, first we write to disks, then we write to memory.

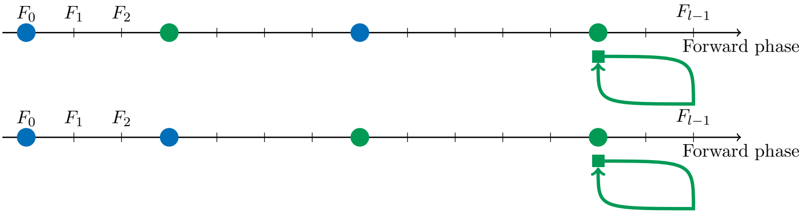




Characterizing the forward phase

Theorem

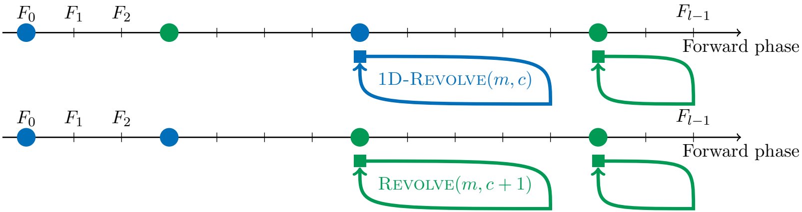
During the forward phase, first we write to disks, then we write to memory.



Characterizing the forward phase

Theorem

During the forward phase, first we write to disks, then we write to memory.

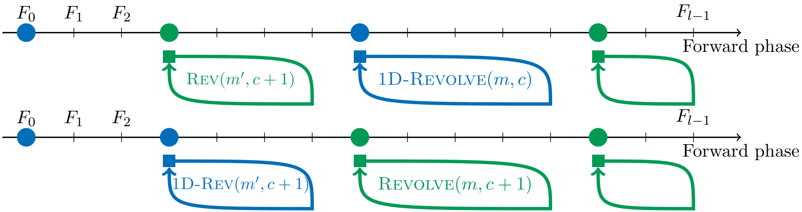


- $\mathcal{E}xe(\text{Revolve}(m, c+1)) \leq \mathcal{E}xe(\text{1D-Revolve}(m, c))$

Characterizing the forward phase

Theorem

During the forward phase, first we write to disks, then we write to memory.

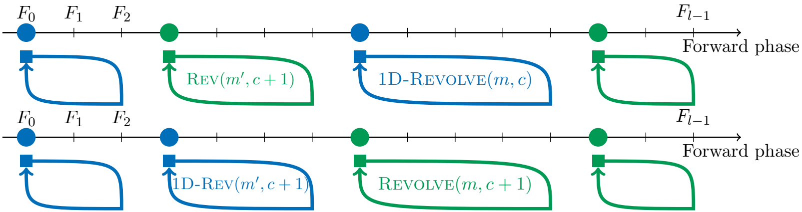


- $\mathcal{E}xe(\text{Revolve}(m, c+1)) \leq \mathcal{E}xe(\text{1D-Revolve}(m, c))$
- $\mathcal{E}xe(\text{1D-Revolve}(m', c+1)) \leq \mathcal{E}xe(\text{Revolve}(m', c+1))$

Characterizing the forward phase

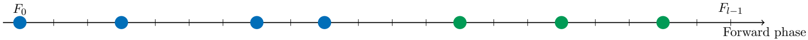
Theorem

During the forward phase, first we write to disks, then we write to memory.

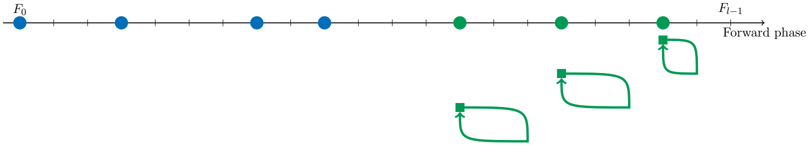


- $\mathcal{E}xe(\text{Revolve}(m, c+1)) \leq \mathcal{E}xe(\text{1D-Revolve}(m, c))$
- $\mathcal{E}xe(\text{1D-Revolve}(m', c+1)) \leq \mathcal{E}xe(\text{Revolve}(m', c+1))$

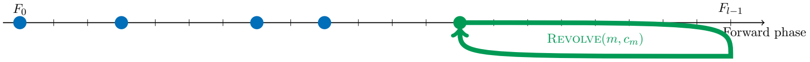
Computing the optimal schedule



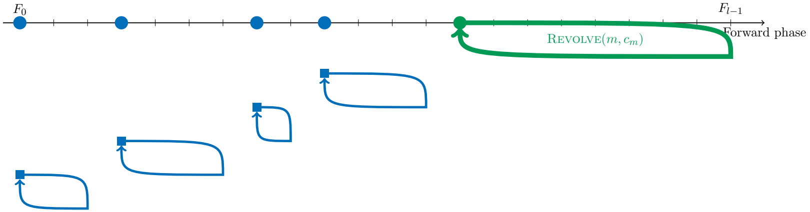
Computing the optimal schedule



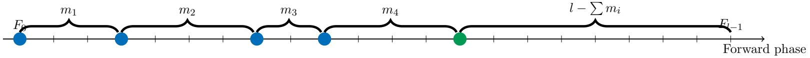
Computing the optimal schedule



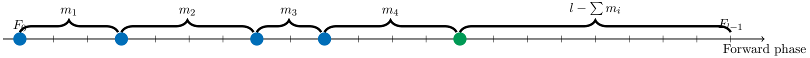
Computing the optimal schedule



Computing the optimal schedule



Computing the optimal schedule

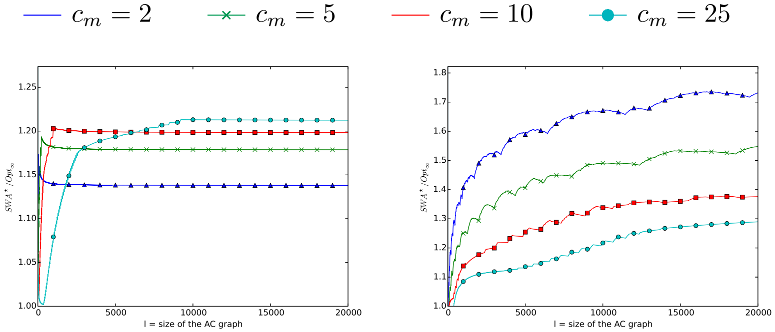


Theorem

We can compute the optimal number of disk checkpoints needed and the space between them in $O(l^2)$ with a dynamic programming algorithm to minimize execution time.

In practice?

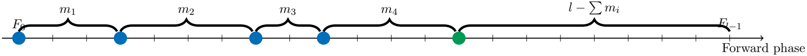
In realistic scenarios we expect to divide the execution time by 2 or 3.



(a) $w_d = r_d = 1$ (b) $w_d = r_d = 5$

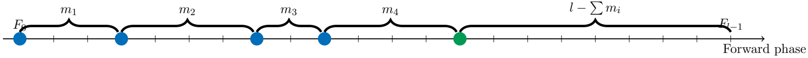
Ratio $\mathrm{SWA}^\star(l, c_m, w_d, r_d)/\mathrm{Opt}_\infty(l, c_m, w_d, r_d)$ as a function of l.

Going further



We know how to compute the m_i's. But the cost of computing them is non-negligible (l^2). Can we do better?

Going further

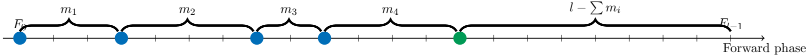


We know how to compute the m_i's. But the cost of computing them is non-negligible (l^2). Can we do better?

Theorem (Weak Periodicity)

Except for a bounded number of them (the bound depends on $X = (c_m, w_d, r_d)$), all the m_i's are equal (to m_X).

Going further



We know how to compute the m_i's. But the cost of computing them is non-negligible (l^2). Can we do better?

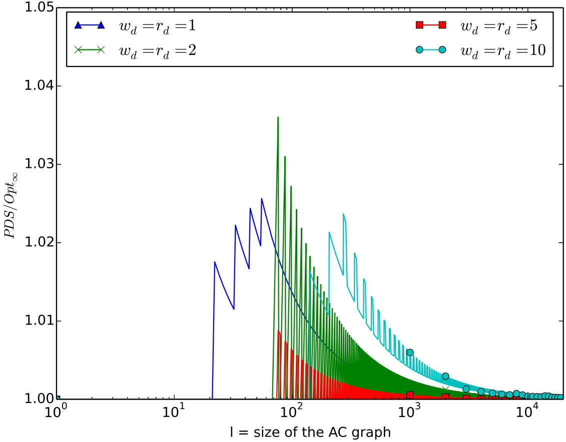
Theorem (Weak Periodicity)

Except for a bounded number of them (the bound depends on $X = (c_m, w_d, r_d)$), all the m_i's are equal (to m_X).

Corollary

Writing disk checkpoint every m_X forward steps is asymptotically optimal.

Additional data



Makespan of periodic algo over optimal: function of l, $c_m = 10$.

Conclusions on adjoint computations

Some numbers:

- The adjoint computation in MITgcm runs in O(days), the gain induced by our optimal algorithm would be non negligible!
- Nek5000 runs on 500K cores, two processes/core on Mira. We need to take reliability into account (future work).

References:

GW00 Griewank and Walther, *Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation*, TOMS, 2000

SW09 Stumm and Walther, *Multistage approaches for optimal offline checkpointing*, SISC, 2009

Perspectives on I/O management

Scalable I/O management is a critical problem for Exascale.

Some directions that need to be solved:

- Models are missing!
  Understanding applications is necessary to design better solutions.
- The energy cost of I/O management is barely studied!
  Energy is also one of the limiting factor for the next scale.
- Applications need to be redesigned!
  Some data may not be as important as other, can we find new strategies to deal with them?