# A few scheduling problems

Yves Robert
Ecole Normale Supérieure de Lyon
& Institut Universitaire de France

http://graal.ens-lyon.fr/~yrobert

University of Colorado Denver, November 2010

# Who needs a scheduler?

- Billions of (mostly idle) computers in the world
- All interconnected by these (mostly empty) network pipes
- Resources: abundant and cheap, not to say unlimited and free

- Greedy resource allocation
- Demand-driven: first-come first-serve, round-robin

Nobody needs a scheduler!!!

# Who needs a scheduler?

- Billions of (mostly idle) computers in the world
- All interconnected by these (mostly empty) network pipes
- Resources: abundant and cheap, not to say unlimited and free

- Greedy resource allocation
- Demand-driven: first-come first-serve, round-robin

**Let's prove this wrong?!**

# Evolution of parallel machines

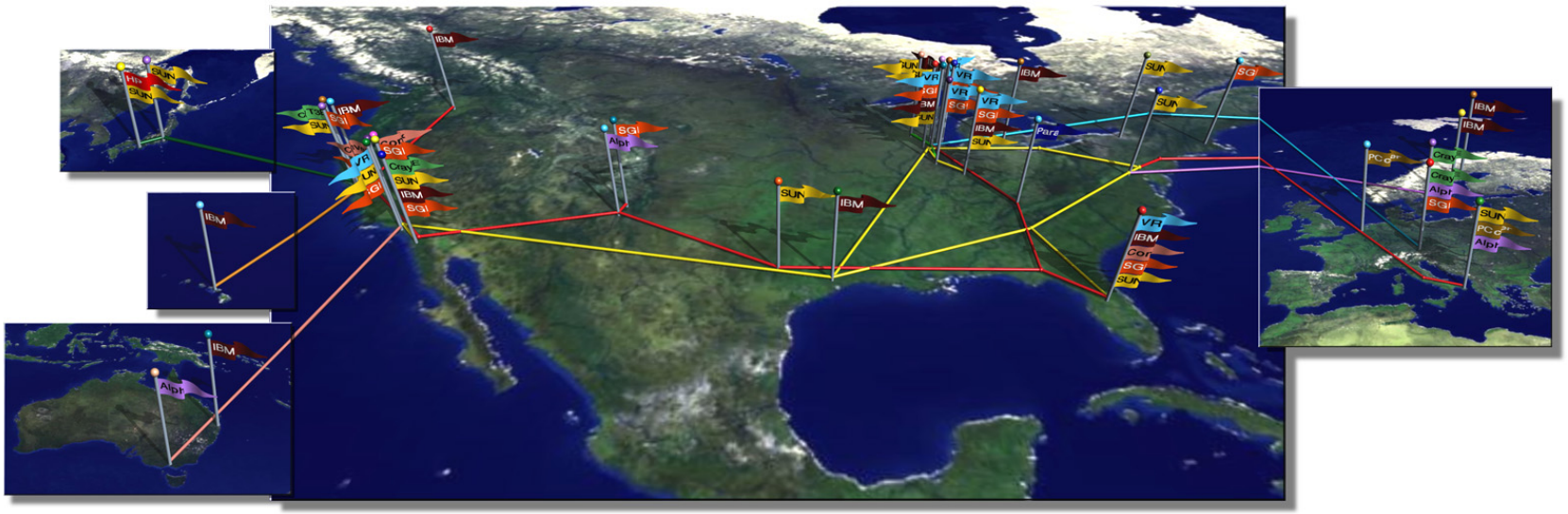
From (good old) parallel architectures . . .

# Evolution of parallel machines

. . . to heterogeneous clusters . . .

# Evolution of parallel machines

. . . to large-scale grid platforms . . .

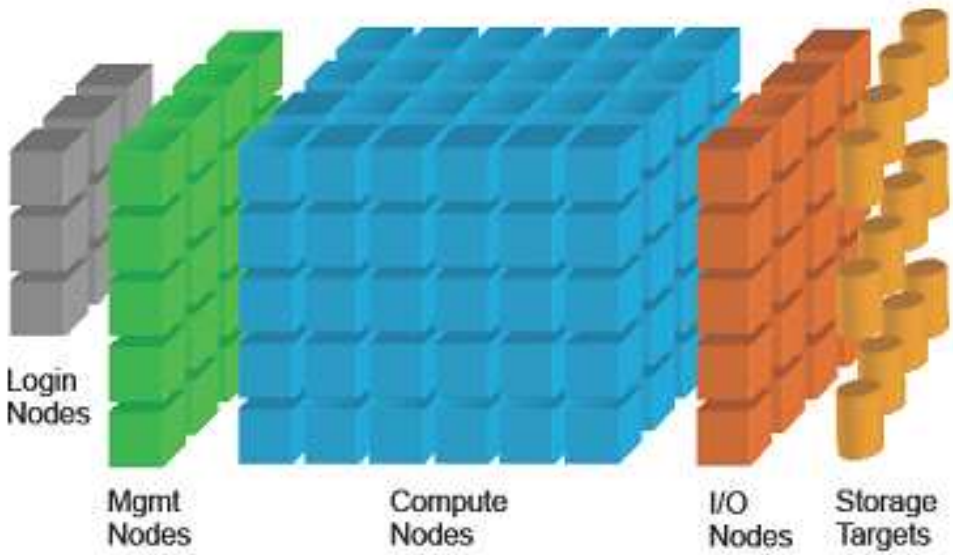


# Evolution of parallel machines

. . . everything hard-to-program . . . *(courtesy Rajeev Thakur)*

## Future Extreme Scale Platforms

- Hundreds of thousands of "nodes"
- Each node has large numbers of cores, including
  - Regular CPUs and accelerators (e.g., GPUs)



# Evolution of parallel machines

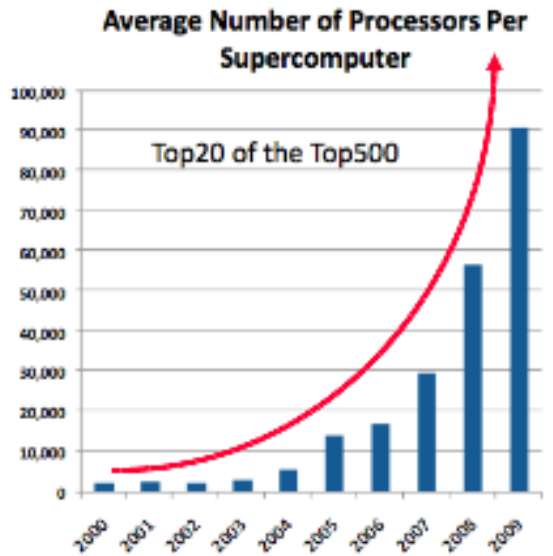
. . . and definitely prone to failure! *(courtesy Al Geist)*

## Fear of the Exponential Growth in Parallelism

Sequoia 1.5M cores 2011

- Fundamental assumptions of today's system software architecture did not anticipate exponential growth in parallelism
- Number of system components is increasing faster than component reliability, which is set by COTS needs
- Number of components and MTBF leading to a paradigm shift – Faults will be the norm rather than rare events. SW adaptation to frequent failures or

"be crazy and die like a dog"



# New platforms, new problems, new solutions

Parallel algorithm design and scheduling were already difficult tasks with homogeneous machines

**On heterogeneous multicore failure-prone platforms, it gets worse** ☹

Need to adapt algorithms and scheduling strategies:
new objective functions, new models

# New platforms, new problems, new solutions

Parallel algorithm design and scheduling were already difficult tasks with homogeneous machines

**On heterogeneous multicore failure-prone platforms, it gets worse** ☹

Need to adapt algorithms and scheduling strategies:
new objective functions, new models

# Outline


1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Independent chunks

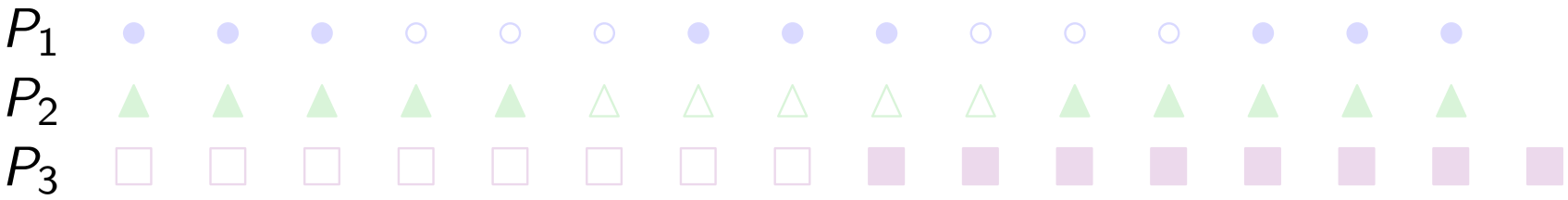
- $B$ independent equal-size tasks ● ● ● ● ● ● ● ● ● ●
- $p$ processors $P_1$, $P_2$, ..., $P_p$
- $w_i$ = time for $P_i$ to process a task ●

- **Intuition:** load of $P_i$ proportional to its speed $1/w_i$
- Assign $n_i$ tasks to $P_i$

$$\textbf{Objective: minimize } \mathrm{T}_{\mathrm{exe}} = \max_{\sum_{i=1}^{p} n_i = B} (n_i \times w_i)$$

# Dynamic programming

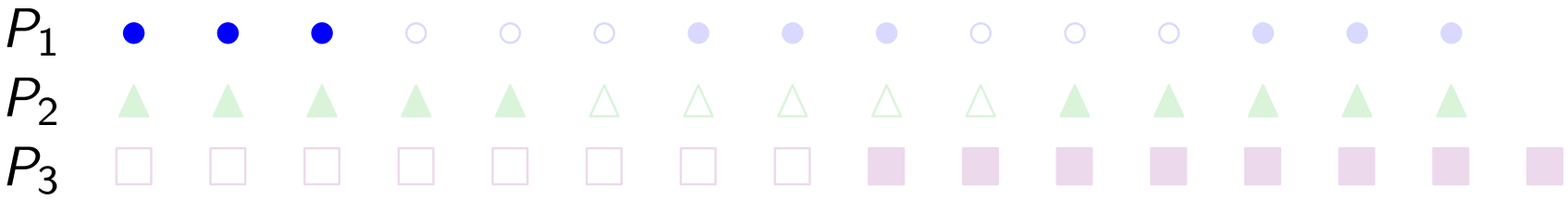
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

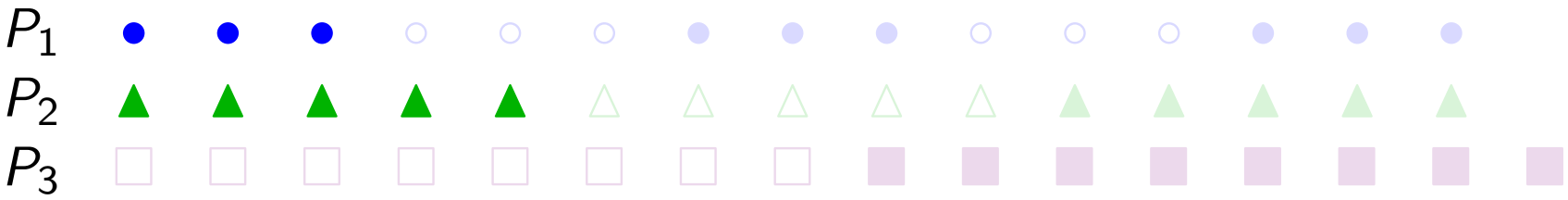
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

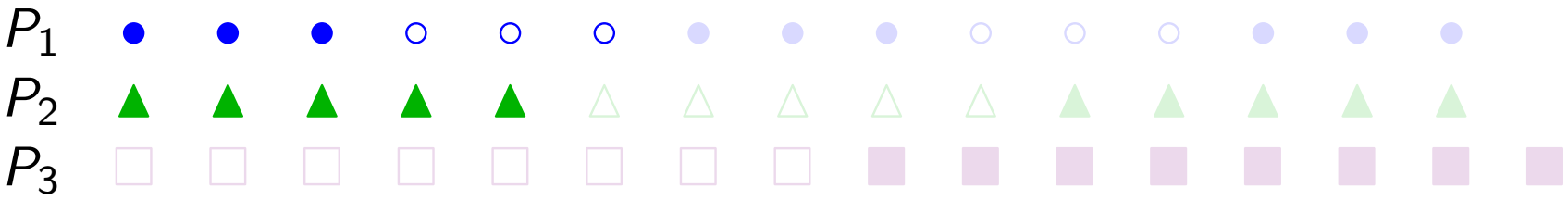
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

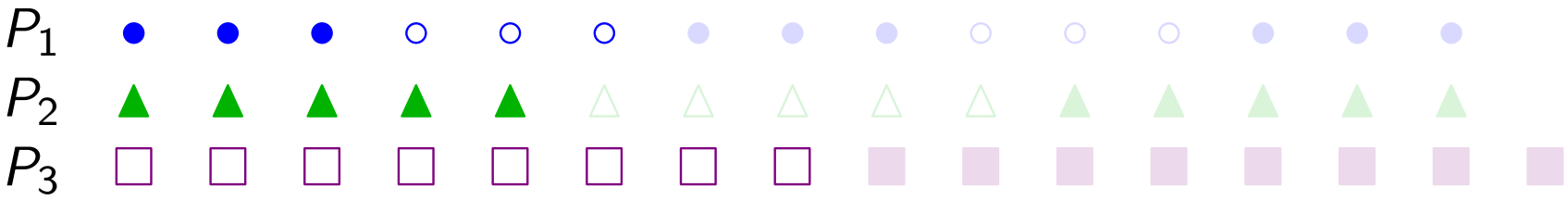
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

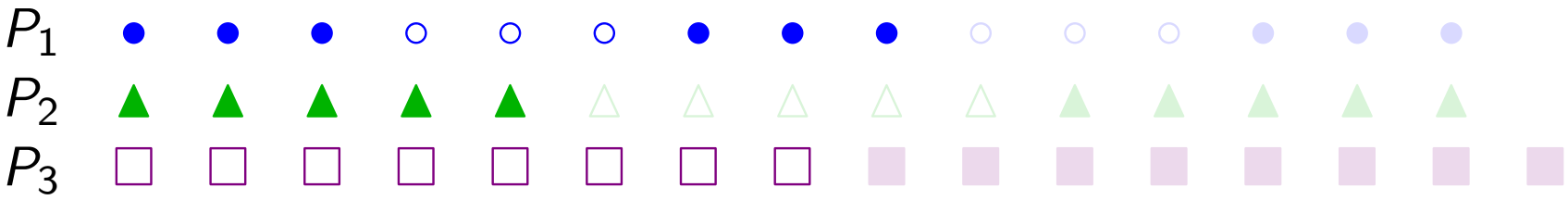
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

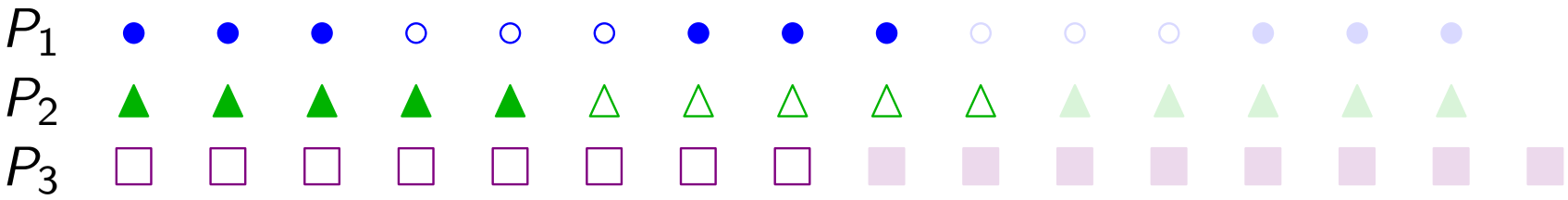
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

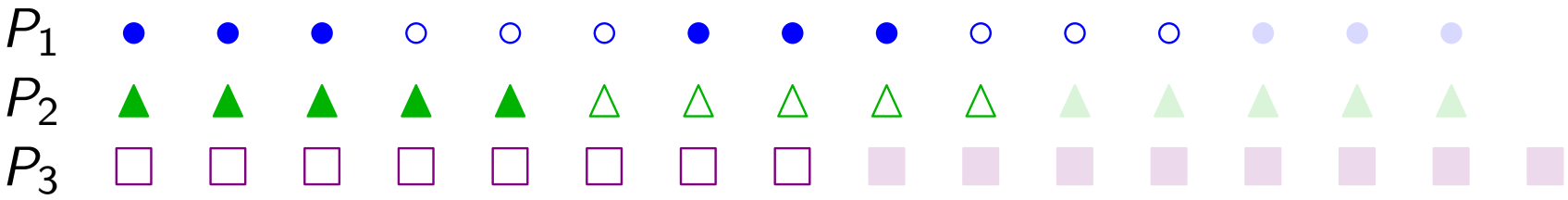
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

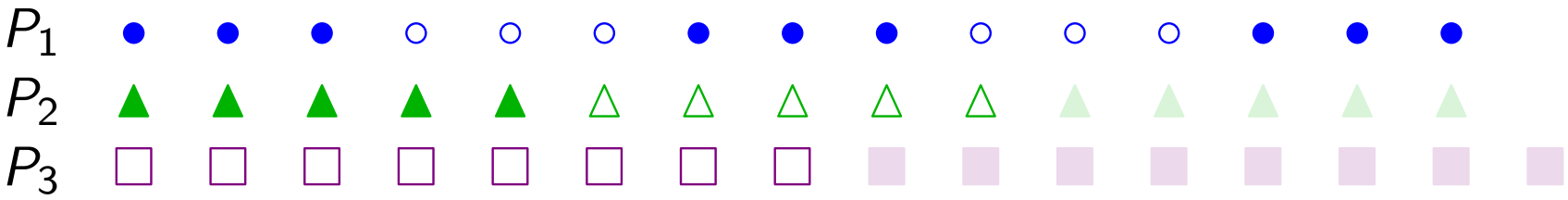
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

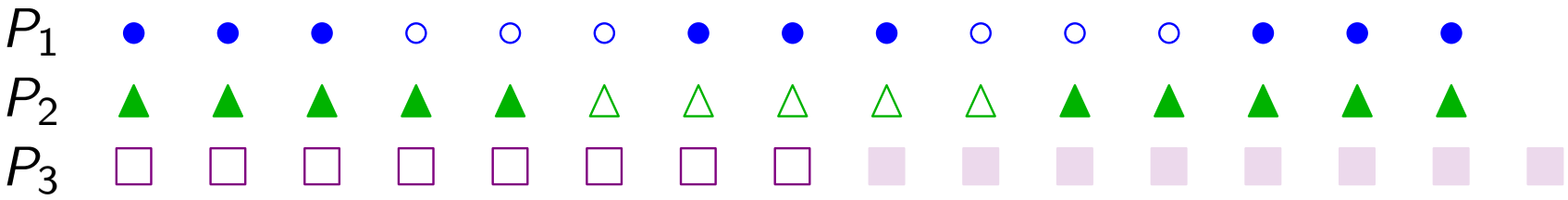
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

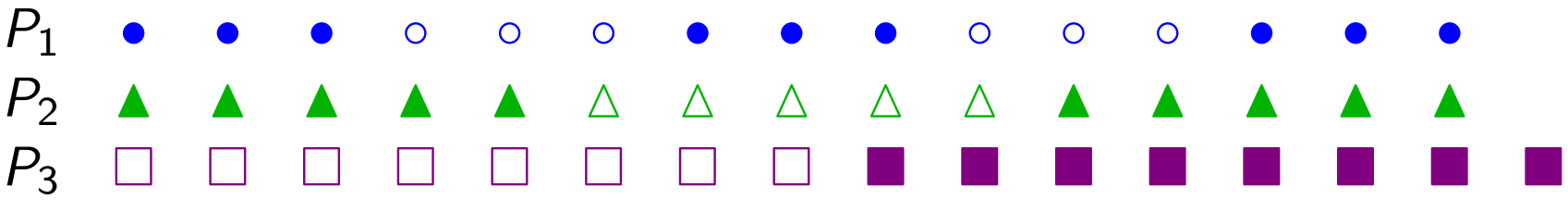
**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Dynamic programming

**With 3 processors: $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$**




| Task | $n_1$ | $n_2$ | $n_3$ | $T_{exe}$ | Selected proc. |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 3 | 2 |
| 2 | 1 | 1 | 0 | 5 | 1 |
| 3 | 2 | 1 | 0 | 6 | 3 |
| 4 | 2 | 1 | 1 | 8 | 1 |
| 5 | 3 | 1 | 1 | 9 | 2 |
| 6 | 3 | 2 | 1 | 10 | 1 |
| 7 | 4 | 2 | 1 | 12 | 1 |
| 8 | 5 | 2 | 1 | 15 | 2 |
| 9 | 5 | 3 | 1 | 15 | 3 |
| 10 | 5 | 3 | 2 | 16 | |

# Static versus dynamic

- Greedy (demand-driven) would have done a perfect job
- Would even be better (possible variations in processor speeds)

Static assignment required **useless thinking** ☹

# Static versus dynamic

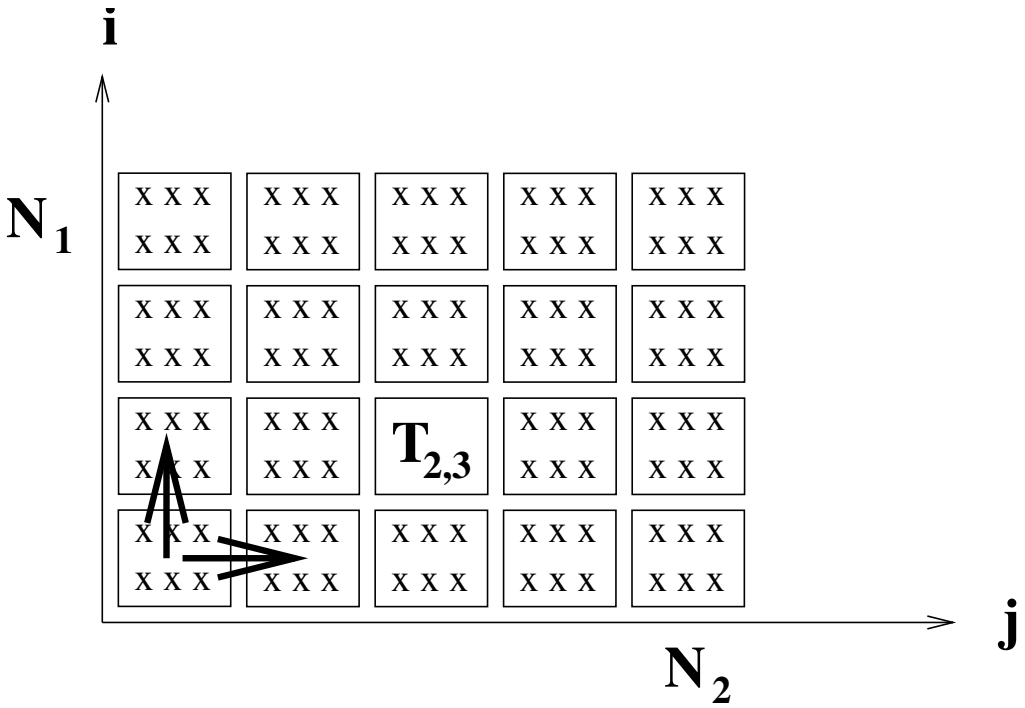
- Greedy (demand-driven) would have done a perfect job
- Would even be better (possible variations in processor speeds)

Static assignment required **useless thinking** ☹

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Coping with dependences




**A simple finite difference problem**

- Iteration space: 2D rectangle of size $N_1 \times N_2$
- Dependences between tiles $\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$

# Allocation strategy (1/3)

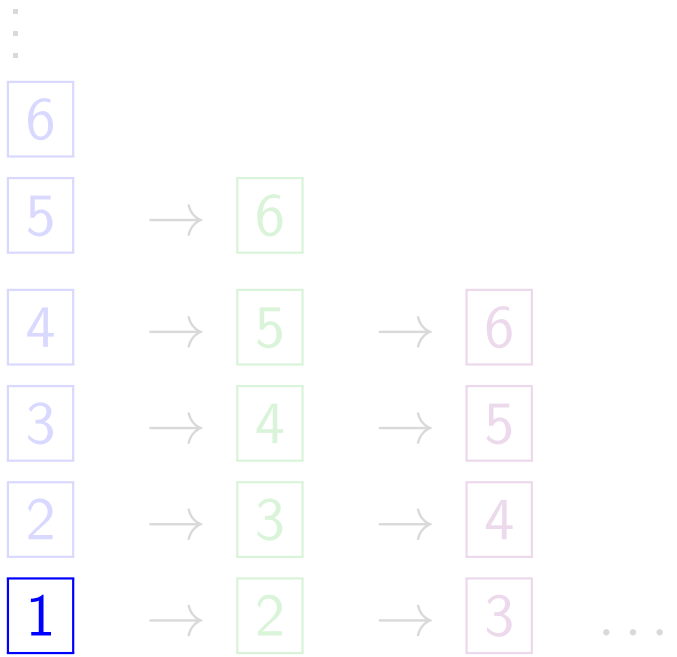
Use column-wise allocation to enhance locality




Stepwise execution

# Allocation strategy (1/3)

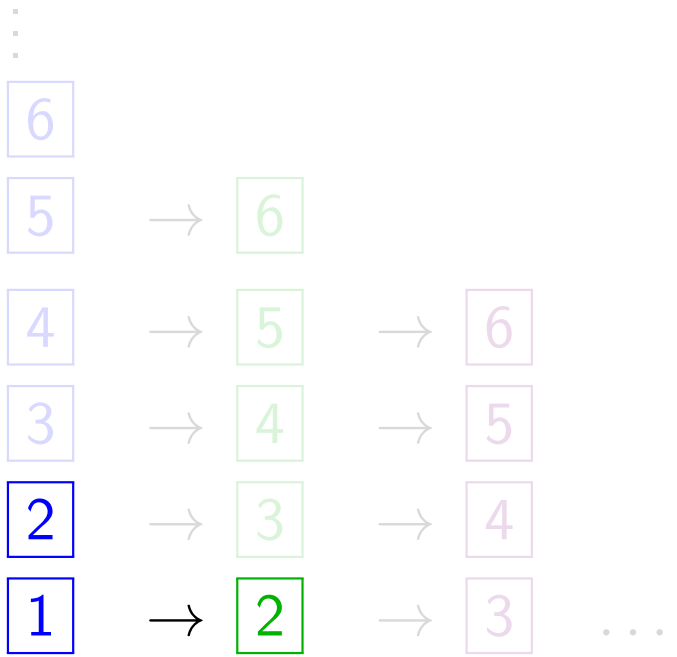
Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (1/3)

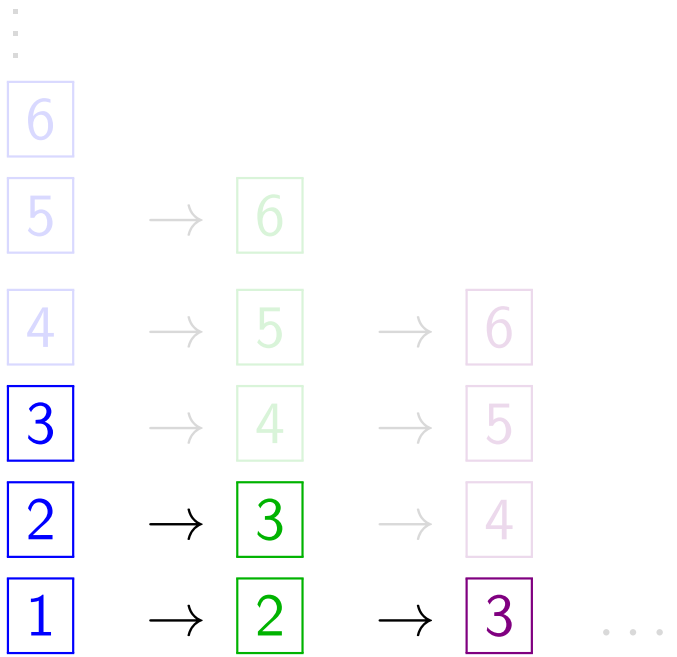
Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (1/3)

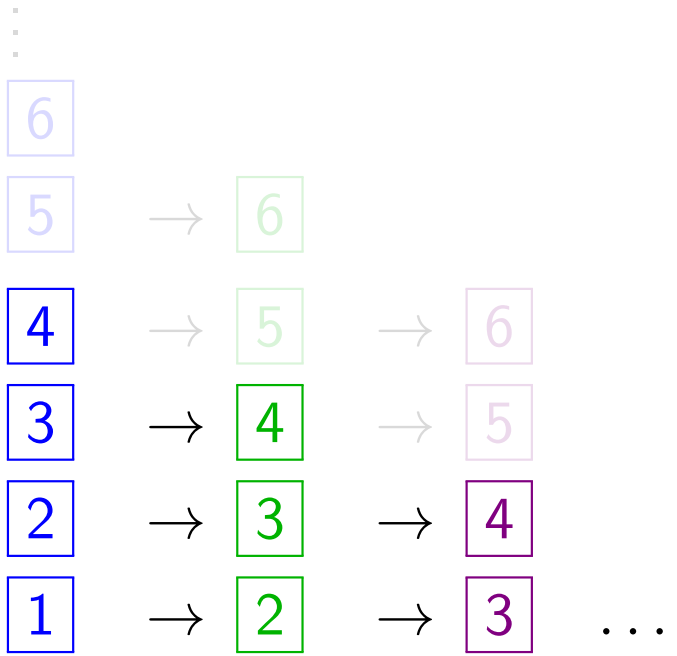
Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (1/3)

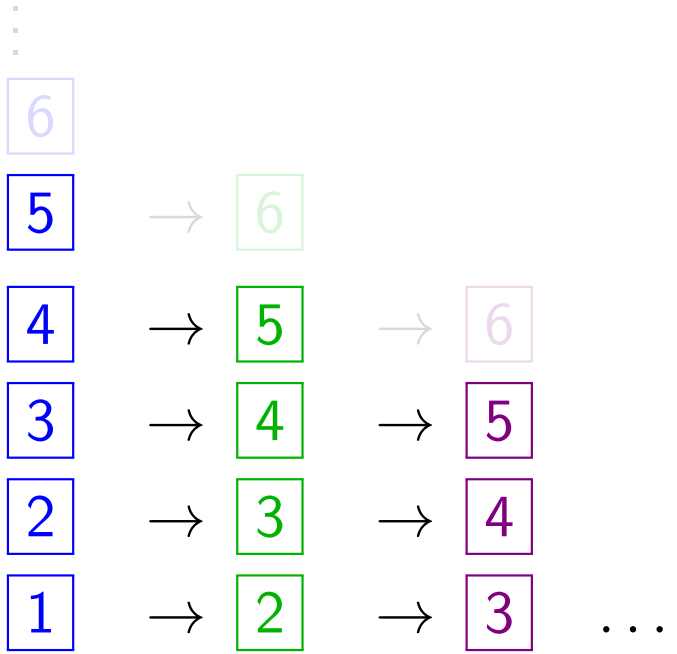
Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (1/3)

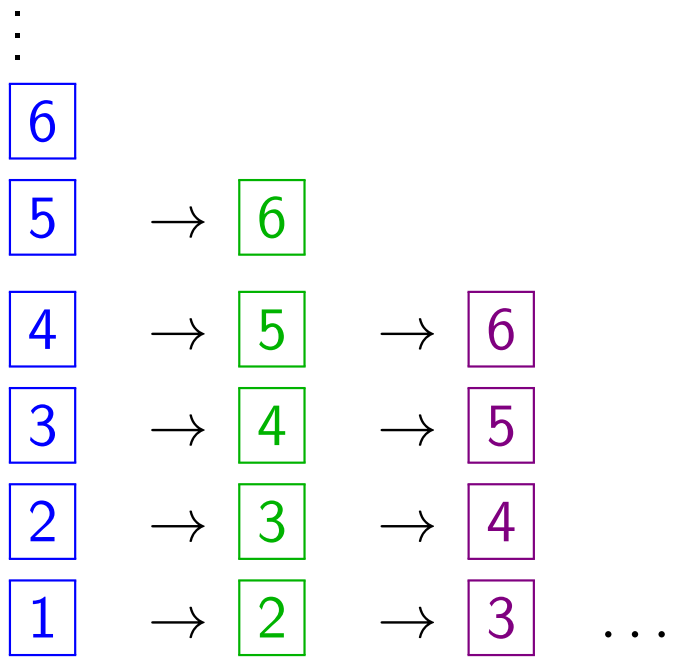
Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (1/3)

Use column-wise allocation to enhance locality




**Stepwise execution**

# Allocation strategy (2/3)

- With column-wise allocation,

$$T_{opt} \approx \frac{N_1 \times N_2}{\sum_{i=1}^{p} \frac{1}{w_i}}.$$

- Greedy (demand-driven) allocation $\Rightarrow$ **slowdown ?!**
- Execution progresses at the pace of the slowest processor ☹

# Allocation strategy (2/3)

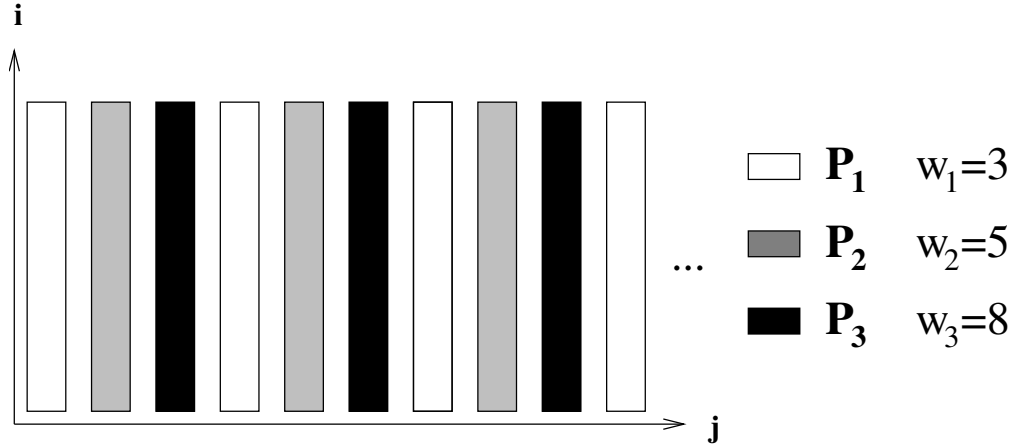
- With column-wise allocation,

$$T_{opt} \approx \frac{N_1 \times N_2}{\sum_{i=1}^{p} \frac{1}{w_i}}.$$

- Greedy (demand-driven) allocation $\Rightarrow$ **slowdown ?!**
- Execution progresses at the pace of the slowest processor ☹

# Allocation strategy (3/3)

With 3 processors, $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$:

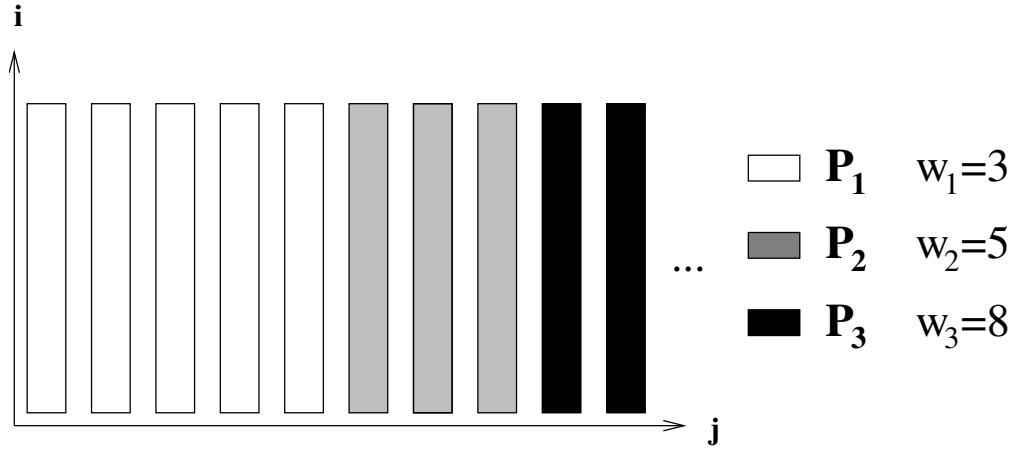



$$T_{exe} \approx \frac{8}{3} N_1 N_2 \approx 2.67\ N_1 N_2$$

$$T_{opt} \approx \frac{120}{79} N_1 N_2 \approx 1.52\ N_1 N_2$$

# Periodic static allocation (1/2)

With 3 processors, $w_1 = 3$, $w_2 = 5$, and $w_3 = 8$:




Assigning blocks of $B = 10$ columns, $T_{exe} \approx 1.6\ N_1 N_2$

# Periodic static allocation (2/2)

- $L = \text{lcm}(w_1, w_2, \ldots, w_p)$
  **Example:** $L = \text{lcm}(3, 5, 8) = 120$
- $P_1$ receives first $n_1 = L/w_1$ columns, $P_2$ next $n_2 = L/w_2$ columns, and so on
- Period: block of $B = n_1 + n_2 + \ldots + n_p$ contiguous columns
  **Example:** $B = n_1 + n_2 + n_3 = 40 + 24 + 15 = 79$
- **Change schedule:**
  - Sort processors so that $n_1 w_1 \leq n_2 w_2 \leq \ldots \leq n_p w_p$
  - Process horizontally within blocks
- **Optimal** ☺

# Lesson learnt?

With different-speed processors . . .
. . . we need to think (design static schedules)

. . . but implementation may remain dynamic ☺

**Example:** demand-driven allocation of blocks of adequate size

. . . well, in some cases it gets truly complicated ☹

# Lesson learnt?

With different-speed processors . . .
. . . we need to think (design static schedules)

. . . but implementation may remain dynamic ☺

**Example:** demand-driven allocation of blocks of adequate size

. . . well, in some cases it gets truly complicated ☹

# Lesson learnt?

With different-speed processors . . .
. . . we need to think (design static schedules)

. . . but implementation may remain dynamic ☺

**Example:** demand-driven allocation of blocks of adequate size

. . . well, in some cases it gets truly complicated ☹

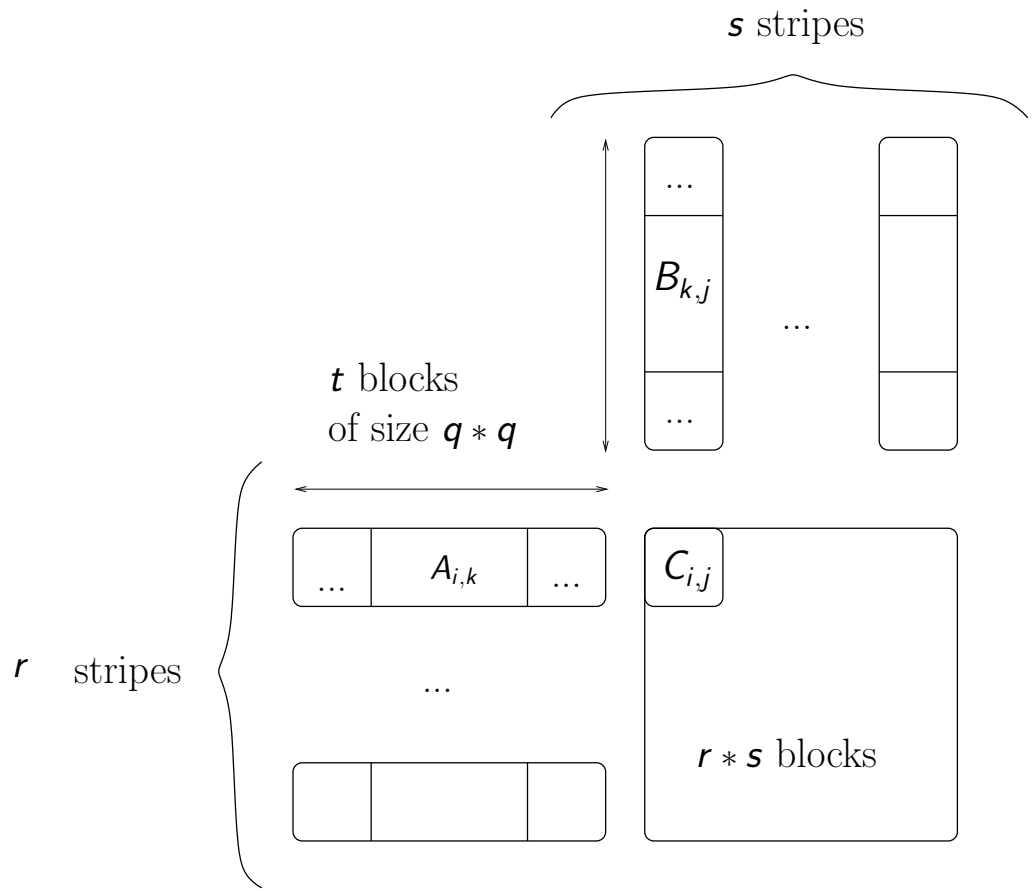
# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Why revisit matrix-product?

- A fundamental computational kernel (the mother of parallel algorithms)
- Archetype of a tightly-coupled application
- Well-understood for homogeneous 2D-arrays of processors
  - Cannon algorithm
  - ScaLAPACK outer product algorithm

# Application model

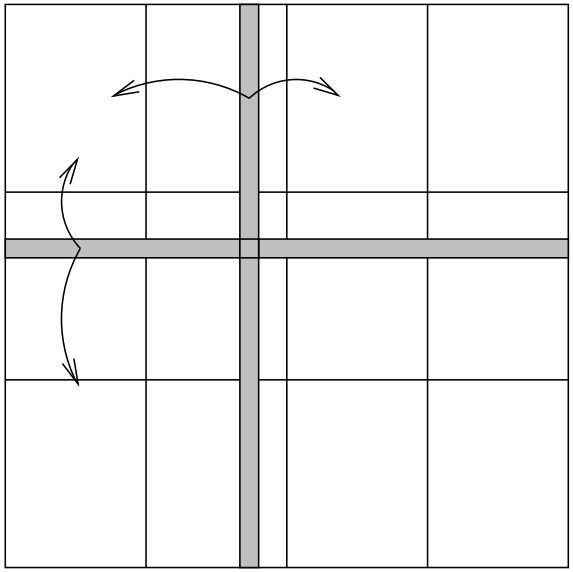



Use $q \times q$ blocks to harness efficiency of Level 3 BLAS

# ScaLAPACK algorithm on (homogeneous) 2D grids (1/2)

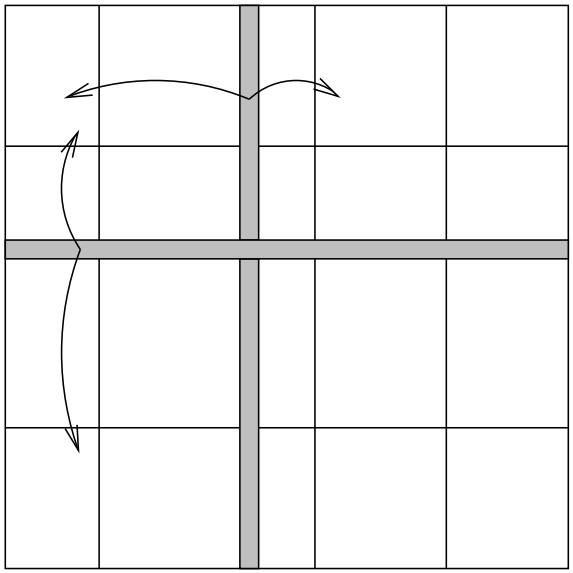
- $C = AB$ on a $p_1 \times p_2$ processor grid
- Granularity: one element = one square $q \times q$ block
- Each matrix is partitioned into $p_1 \times p_2$ rectangles
- Each processor is responsible for updating its rectangle
- Outer product version: at each step,
  - a column of blocks is communicated (broadcast) horizontally
  - a row of blocks is communicated (broadcast) vertically

# ScaLAPACK algorithm on (homogeneous) 2D grids (2/2)



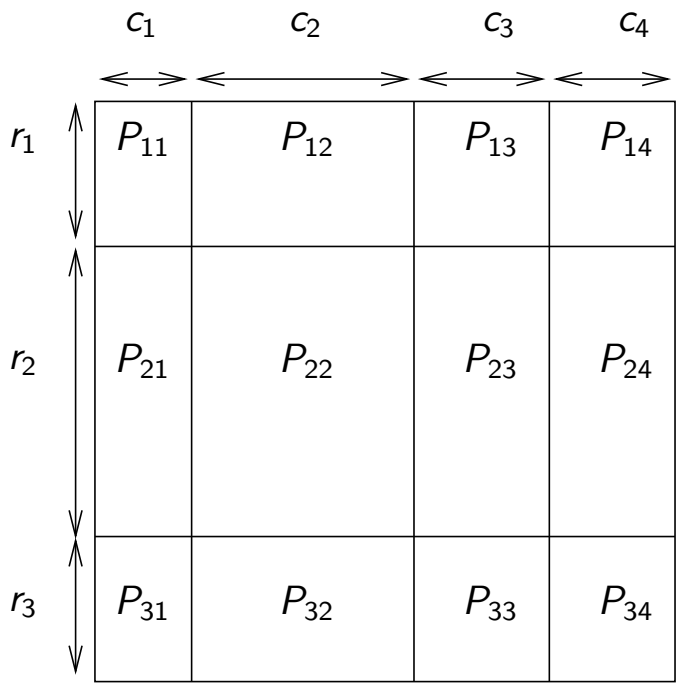
**Matrix product on a $3 \times 4$ homogeneous 2D-grid**

# Algorithm on (heterogeneous) 2D grids (2/2)



**Matrix product on a $3 \times 4$ heterogeneous 2D-grid**

# 2D load balancing (1/2)




**Objective:** $\max_{r_i \times w_{ij} \times c_j \leq 1} \left\{ \left( \sum_{i=1}^{p_1} r_i \right) \times \left( \sum_{j=1}^{p_2} c_j \right) \right\}$

Maximize total number of elements processed within one time unit

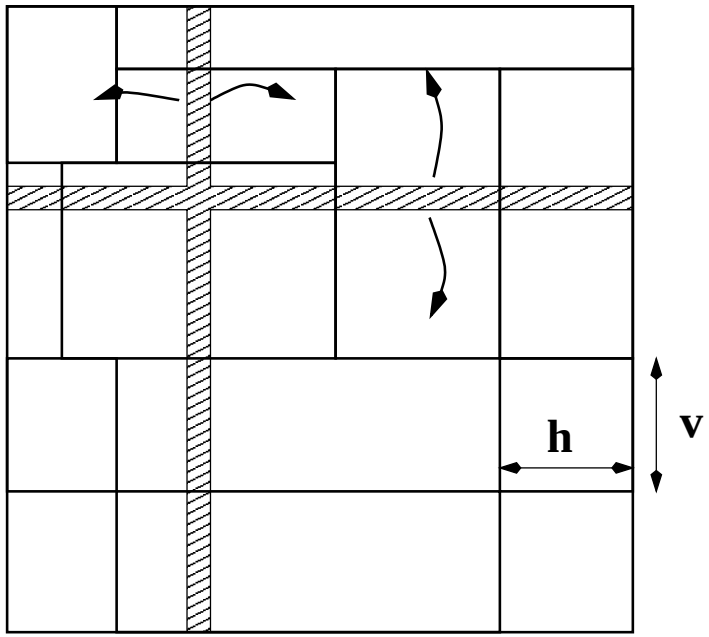
# 2D load balancing (2/2)

Given $p$ processors, how to arrange them along a 2D grid of size $p_1 \times p_2 \leq p$ ...

... so as to optimally load-balance the work of the processors

- Search among all possible arrangements of $p_1 \times p_2$ processors as a $p_1 \times p_2$ grid
- For each arrangement, solve optimization problem
- **NP-hard** ☹

# Matrix product on heterogeneous clusters




**Matrix product with 13 heterogeneous processors**

# Optimization

How to compute the *area* and *shape* of the $p$ rectangles?

- **Load-balancing computations** assign *areas* proportional to speeds
- **Minimizing communication overhead** choose *shapes*:
  - total communication volume

$$\hat{C} = \sum_{i=1}^{p} (h_i + v_i)$$

  *sum* of the half perimeters of the $p$ rectangles
  - for parallel communications:

$$\hat{M} = \max_{i=1}^{p} (h_i + v_i)$$

- Both problems NP-hard ☹

# Optimization

How to compute the *area* and *shape* of the $p$ rectangles?

- **Load-balancing computations** assign *areas* proportional to speeds
- **Minimizing communication overhead** choose *shapes*:
  - total communication volume

$$\hat{C} = \sum_{i=1}^{p} (h_i + v_i)$$

  *sum* of the half perimeters of the $p$ rectangles
  - for parallel communications:

$$\hat{M} = \max_{i=1}^{p} (h_i + v_i)$$

- **Both problems NP-hard** ☹

# Lesson learnt?

Can provide efficient algorithms for tightly coupled applications but requires lots of efforts

. . . implementation cannot be demand-driven
unless ready to pay huge performance degradation

**Example:** resource selection plus static ordering mandatory for heterogeneous platforms

# Lesson learnt?

Can provide efficient algorithms for tightly coupled applications but requires lots of efforts

. . . implementation cannot be demand-driven
unless ready to pay huge performance degradation

**Example:** resource selection plus static ordering mandatory for heterogeneous platforms

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

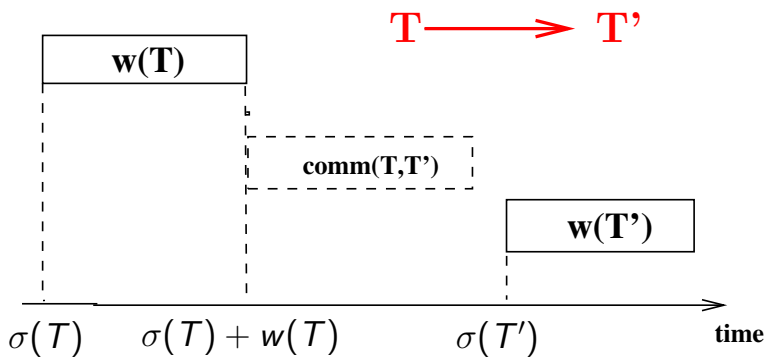
# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Traditional scheduling – Framework

- Application = DAG $G = (\mathcal{T}, E, w)$
  - $\mathcal{T}$ = set of tasks
  - $E$ = dependence constraints
  - $w(T)$ = computational cost of task $T$ (execution time)
  - $c(T, T')$ = communication cost (data sent from $T$ to $T'$)
- Platform
  - Set of $p$ identical processors
- Schedule
  - $\sigma(T)$ = date to begin execution of task $T$
  - alloc$(T)$ = processor assigned to it

# Traditional scheduling – Constraints




- Data dependences If $(T, T') \in E$ then
  - if $\text{alloc}(T) = \text{alloc}(T')$ then $\sigma(T) + w(T) \leq \sigma(T')$
  - if $\text{alloc}(T) \neq \text{alloc}(T')$ then $\sigma(T) + w(T) + c(T, T') \leq \sigma(T')$
- Resource constraints

$$\text{alloc}(T) = \text{alloc}(T') \Rightarrow$$
$$(\sigma(T) + w(T) \leq \sigma(T')) \text{ or } (\sigma(T') + w(T') \leq \sigma(T))$$

# Traditional scheduling – Objective functions

- Makespan or total execution time

$$MS(\sigma) = \max_{T \in \mathcal{T}} (\sigma(T) + w(T))$$

- Other classical objectives:
  - Sum of completion times
  - With release dates: max flow (response time), or sum flow
  - Fairness oriented: max stretch, or sum stretch

# Traditional scheduling – About the model

- Simple but OK for computational resources
  - No CPU sharing, even in models with preemption
  - At most one task running per processor at any time-step
- Very crude for network resources
  - Unlimited number of simultaneous sends/receives per processor
  - No contention $\rightarrow$ unbounded bandwidth on any link
  - Fully connected interconnection graph (clique)
- In fact, model assumes infinite network capacity

# Makespan minimization

- NP-hardness
  - $Pb(p)$ NP-complete for independent tasks and no communications ($E = \emptyset$, $p = 2$ and $c = \overline{0}$)
  - $Pb(p)$ NP-complete for UET-UCT graphs ($w = c = \overline{1}$)

- Approximation algorithms
  - Without communications, list scheduling is a $(2 - \frac{1}{p})$-approximation
  - With communications, result extends to coarse-grain graphs
  - With communications, no $\lambda$-approximation in general

# List scheduling – Without communications

Initialization:

- Compute priority level of all tasks
- Priority queue = list of free tasks (tasks without predecessors) sorted by priority

While there remain tasks to execute:

- Add new free tasks, if any, to the queue.
- If there are $q$ available processors and $r$ tasks in the queue, remove first $\min(q, r)$ tasks from the queue and execute them

Priority level

- Use critical path: longest path from the task to an exit node
- Computed recursively by a bottom-up traversal of the graph

# List scheduling – Without communications

Initialization:

- Compute priority level of all tasks
- Priority queue = list of free tasks (tasks without predecessors) sorted by priority

While there remain tasks to execute:

- Add new free tasks, if any, to the queue.
- If there are $q$ available processors and $r$ tasks in the queue, remove first $\min(q, r)$ tasks from the queue and execute them

Priority level

- Use critical path: longest path from the task to an exit node
- Computed recursively by a bottom-up traversal of the graph

# List scheduling – With communications (1/2)

- Priority level
  - Use pessimistic critical path: include all edge costs in the weight
  - Computed recursively by a bottom-up traversal of the graph

- MCP Modified Critical Path
  - Assign free task with highest priority to best processor
  - Best processor = finishes execution first, given already taken scheduling decisions
  - Free tasks may not be ready for execution (communication delays)
  - May explore inserting the task in empty slots of schedule
  - Complexity $O(|V| \log |V| + (|E| + |V|)p)$

# List scheduling – With communications (2/2)

- EFT Earliest Finish Time
  - Dynamically recompute priorities of free tasks
  - Select free task that finishes execution first (on best processor), given already taken scheduling decisions
  - Higher complexity $O(|V|^3 p)$
  - May miss "urgent" tasks on critical path
- Other approaches
  - Two-step: clustering + load balancing
    - DSC Dominant Sequence Clustering $O((|V| + |E|) \log |V|)$
    - LLB List-based Load Balancing $O(C \log C + |V|)$ ($C$ number of clusters generated by DSC)
  - Low-cost: FCP Fast Critical Path
    - Maintain constant-size sorted list of free tasks:
    - Best processor = first idle or the one sending last message
    - Low complexity $O(|V| \log p + |E|)$

# Extending the model to heterogeneous clusters

- Task graph with $n$ tasks $T_1, \ldots, T_n$.
- Platform with $p$ heterogeneous processors $P_1, \ldots, P_p$.
- Computation costs:
  - $w_{iq}$ = execution time of $T_i$ on $P_q$
  - $\overline{w_i} = \frac{\sum_{q=1}^{p} w_{iq}}{p}$ average execution time of $T_i$
  - particular case: consistent tasks $w_{iq} = w_i \times \gamma_q$
- Communication costs:
  - data$(i,j)$: data volume for edge $e_{ij} : T_i \rightarrow T_j$
  - $v_{qr}$: communication time for unit-size message from $P_q$ to $P_r$ (zero if $q = r$)
  - com$(i,j,q,r)$ = data$(i,j) \times v_{qr}$ communication time from $T_i$ executed on $P_q$ to $P_j$ executed on $P_r$
  - $\overline{\text{com}_{ij}} = \text{data}(i,j) \times \frac{\sum_{1 \leq q,r \leq p, q \neq r} v_{qr}}{p(p-1)}$ average communication cost for edge $e_{ij} : T_i \rightarrow T_j$

# Rewriting constraints

**Dependences** For $e_{ij} : T_i \to T_j$, $q = \text{alloc}(T_i)$ and $r = \text{alloc}(T_j)$:

$$\sigma(T_i) + w_{iq} + \text{com}(i, j, q, r) \leq \sigma(T_j)$$

**Resources** If $q = \text{alloc}(T_i) = \text{alloc}(T_j)$, then

$$(\sigma(T_i) + w_{iq} \leq \sigma(T_j)) \text{ or } (\sigma(T_j) + w_{jq} \leq \sigma(T_i))$$

**Makespan**

$$\max_{1 \leq i \leq n} \left(\sigma(T_i) + w_{i,\text{alloc}(T_i)}\right)$$

# HEFT: Heterogeneous Earliest Finish Time

- Priority level:
  - $\text{rank}(T_i) = \overline{w_i} + \max_{T_j \in \text{Succ}(T_i)} (\overline{\text{com}_{ij}} + \text{rank}(T_j))$,
    where $\text{Succ}(T)$ is the set of successors of $T$
  - Recursive computation by bottom-up traversal of the graph
- Allocation
  - For current task $T_i$, determine best processor $P_q$:
    minimize $\sigma(T_i) + w_{iq}$
  - Enforce constraints related to communication costs
  - Insertion scheduling: look for $t = \sigma(T_i)$ s.t. $P_q$ is available during interval $[t, t + w_{iq}[$
- Complexity: same as MCP without/with insertion

# What's wrong?

- ☺ Nothing (still may need to map a DAG onto a platform!)
- ☹ Absurd communication model:
  complicated: many parameters to instantiate
  while not realistic (clique + no contention)
- ☹ Wrong metric: need to relax makespan minimization objective

# What's wrong?

- ☺ Nothing (still may need to map a DAG onto a platform!)
- ☹ Absurd communication model:
  complicated: many parameters to instantiate
  while not realistic (clique + no contention)
- ☹ Wrong metric: need to relax makespan minimization objective

# What's wrong?

- ☺ Nothing (still may need to map a DAG onto a platform!)
- ☹ Absurd communication model:
  complicated: many parameters to instantiate
  while not realistic (clique + no contention)
- ☹ Wrong metric: need to relax makespan minimization objective

# Favorite communication models

**Bounded multi-port**

- several concurrent incoming and outgoing communications
- share total available bandwidth
- bottleneck = network card
- multi-threaded systems

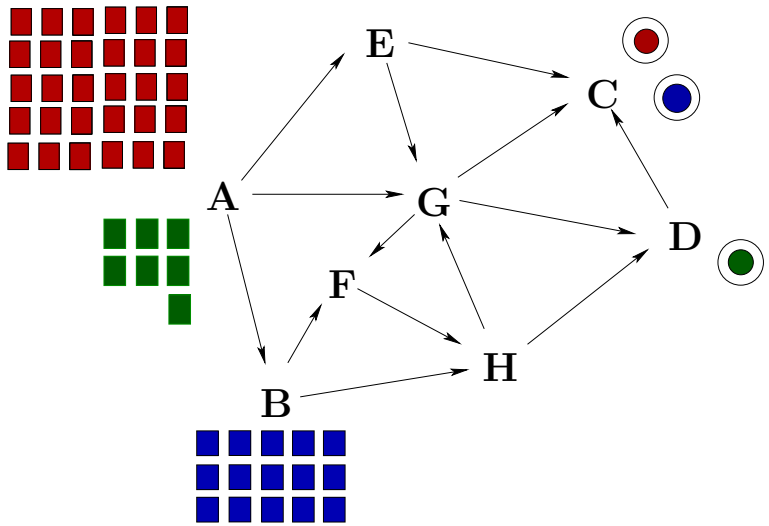
**One-port**

- serialize incoming (and/or) outgoing communications
- pessimistic but realistic
- standard MPI

# Outline

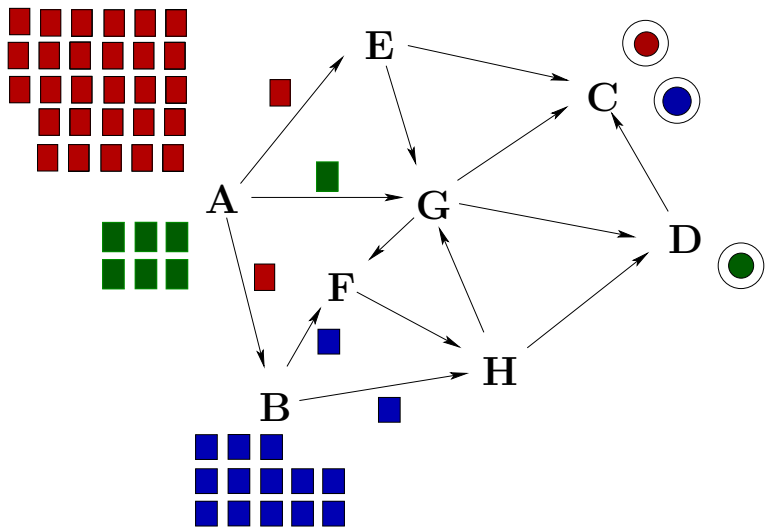
1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Problem



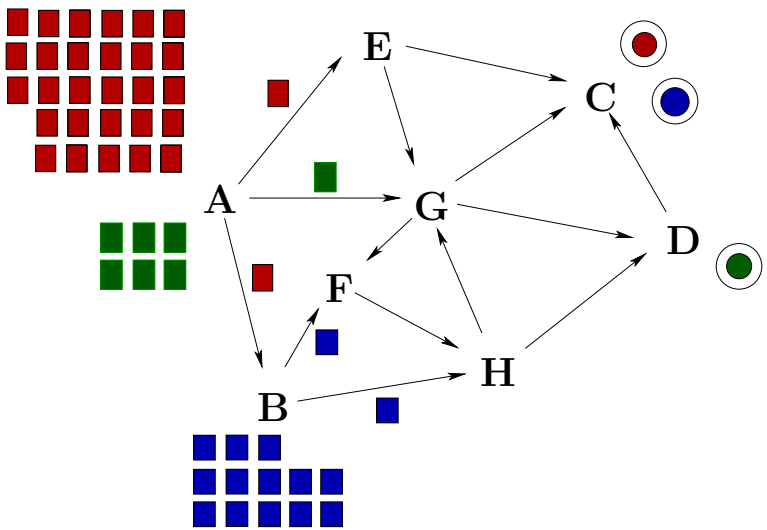

- Routing sets of messages from sources to destinations
- Paths not fixed a priori
- Packets of same message may follow different paths

# Hypotheses




- A packet crosses an edge within one time-step
- At any time-step, at most one packet crosses an edge

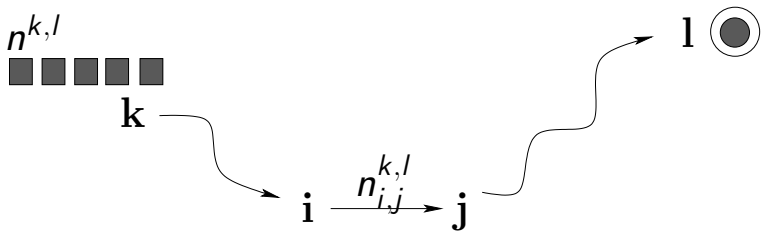
# Hypotheses




- A packet crosses an edge within one time-step
- At any time-step, at most one packet crosses an edge

Scheduling: for each time-step, decide which packet crosses any given edge

# Notation




- $n^{k,l}$: total number of packets to be routed from $k$ to $l$
- $n_{i,j}^{k,l}$: total number of packets routed from $k$ to $l$ and crossing edge $(i,j)$

# Lower bound

**Congestion** $C_{i,j}$ of edge $(i,j)$
= total number of packets that cross $(i,j)$

$$C_{i,j} = \sum_{(k,l)|n^{k,l}>0} n_{i,j}^{k,l} \qquad C_{\max} = \max_{i,j} C_{i,j}$$

$C_{\max}$ lower bound on schedule makespan

$$C^* \geq C_{\max}$$

$\Rightarrow$ "Fluidified" solution in $C_{\max}$?

# Lower bound

**Congestion** $C_{i,j}$ of edge $(i,j)$
= total number of packets that cross $(i,j)$

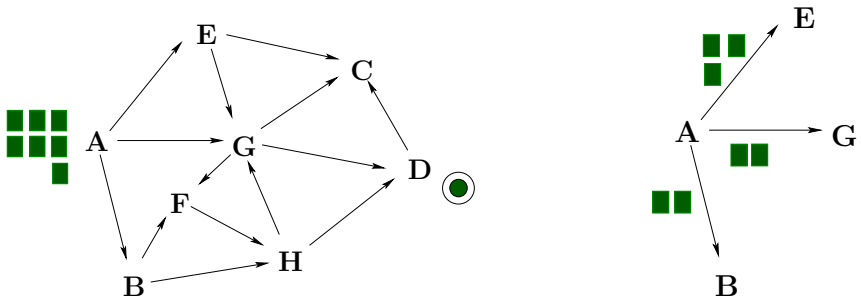
$$C_{i,j} = \sum_{(k,l)|n^{k,l}>0} n_{i,j}^{k,l} \qquad C_{\max} = \max_{i,j} C_{i,j}$$

$C_{\max}$ lower bound on schedule makespan

$$C^* \geq C_{\max}$$

$\Rightarrow$ "Fluidified" solution in $C_{\max}$?

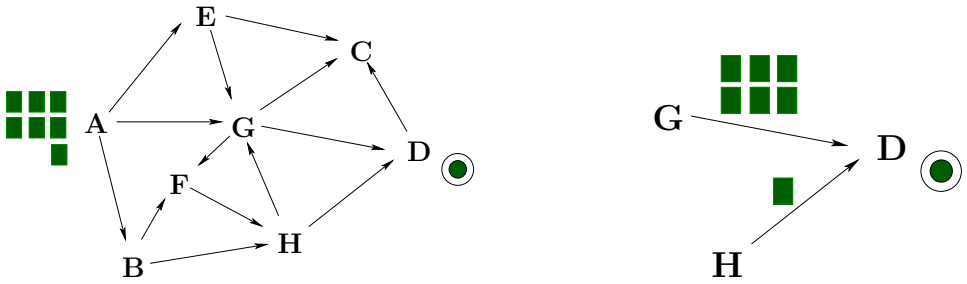
# Equations (1/2)




- Initialization (packets leave node $k$): $\sum_{j|(k,j)\in A} n_{k,j}^{k,l} = n^{k,l}$
- Reception (packets reach node $l$): $\sum_{i|(i,l)\in A} n_{i,l}^{k,l} = n^{k,l}$
- Conservation law (crossing intermediate node $i$):

$$\sum_{i|(i,j)\in A} n_{i,j}^{k,l} = \sum_{i|(j,i)\in A} n_{j,i}^{k,l} \quad \forall (k,l), j \neq k, j \neq l$$

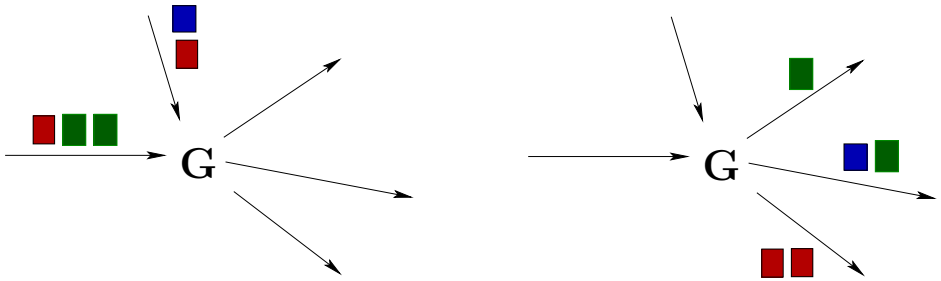
# Equations (1/2)




- Initialization (packets leave node $k$): $\sum_{j|(k,j)\in A} n_{k,j}^{k,l} = n^{k,l}$
- Reception (packets reach node $l$): $\sum_{i|(i,l)\in A} n_{i,l}^{k,l} = n^{k,l}$
- Conservation law (crossing intermediate node $i$):

$$\sum_{i|(i,j)\in A} n_{i,j}^{k,l} = \sum_{i|(j,i)\in A} n_{j,i}^{k,l} \quad \forall (k,l), j \neq k, j \neq l$$

# Equations (1/2)




- Initialization (packets leave node $k$): $\sum_{j|(k,j)\in A} n_{k,j}^{k,l} = n^{k,l}$
- Reception (packets reach node $l$): $\sum_{i|(i,l)\in A} n_{i,l}^{k,l} = n^{k,l}$
- Conservation law (crossing intermediate node $i$):

$$\sum_{i|(i,j)\in A} n_{i,j}^{k,l} = \sum_{i|(j,i)\in A} n_{j,i}^{k,l} \quad \forall (k,l), j \neq k, j \neq l$$

# Equations (2/2)

- Congestion

$$C_{i,j} = \sum_{(k,l)|n^{k,l}>0} n_{i,j}^{k,l}$$

- Objective function

$$C_{\max} \geq C_{i,j}, \qquad \forall i, j$$

$$\text{Minimize } C_{\max}$$

Linear program in rational numbers: polynomial-time solution. In practice use GLPK, Maple, Mupad ...

# Equations (2/2)

- Congestion

$$C_{i,j} = \sum_{(k,l)|n^{k,l}>0} n_{i,j}^{k,l}$$

- Objective function

$$C_{\max} \geq C_{i,j}, \qquad \forall i, j$$

$$\text{Minimize } C_{\max}$$

Linear program in rational numbers: polynomial-time solution. In practice use GLPK, Maple, Mupad ...

# Routing algorithm

- Compute optimal solution $C_{\max}$, $n_{i,j}^{k,l}$ of previous linear program
- Periodic schedule:
  - Define $\Omega = \sqrt{C_{\max}}$
  - Use $\left\lceil \frac{C_{\max}}{\Omega} \right\rceil$ periods of length $\Omega$
  - During each period, edge $(i, j)$ forwards (at most)

$$m_{i,j}^{k,l} = \left\lfloor \frac{n_{i,j}^{k,l} \Omega}{C_{\max}} \right\rfloor$$

    packets that go from $k$ to $l$
- Clean-up: sequentially process residual packets inside network

# Performance

- Schedule is feasible
- Schedule is asymptotically optimal:

$$C_{max} \leq C^* \leq C_{max} + O(\sqrt{C_{max}})$$

# Why does it work?

- Relaxation of objective function
- Rational number of packets in LP formulation
- Periods long enough so that rounding down to integer numbers has negligible impact
- Periods numerous enough so that loss in first and last periods has negligible impact
- Periodic schedule, described in compact form

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Master-worker tasking: framework

## Heterogeneous resources

- Processors of different speeds
- Communication links with various bandwidths

## Large number of independent tasks to process

- Tasks are atomic
- Tasks have same size

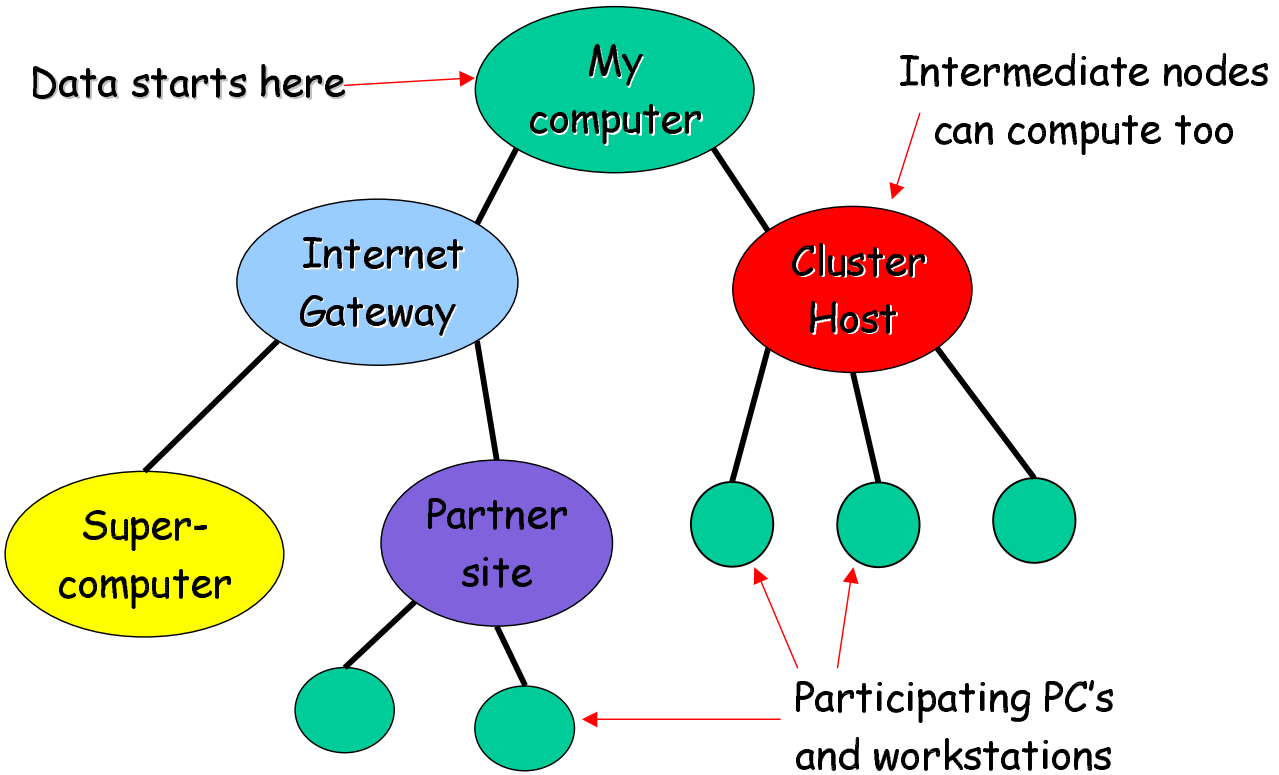
## Single data repository

- One master initially holds data for all tasks
- Several workers arranged along a star, a tree or a general graph

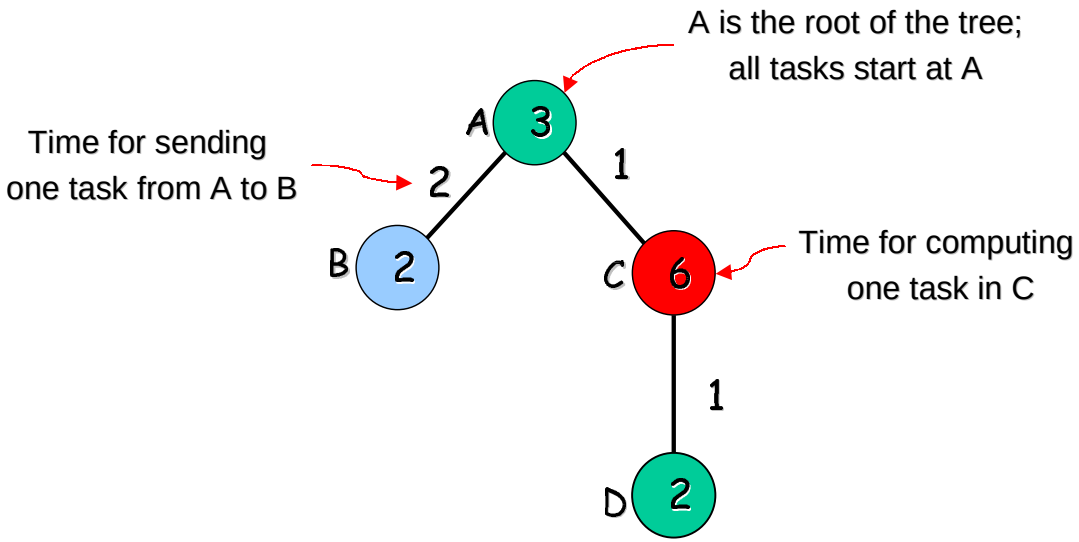
# Application examples

- Monte Carlo methods
- SETI@home
- Factoring large numbers
- Searching for Mersenne primes
- Particle detection at CERN (LHC@home)
- ... and many others: see BOINC at `http://boinc.berkeley.edu`

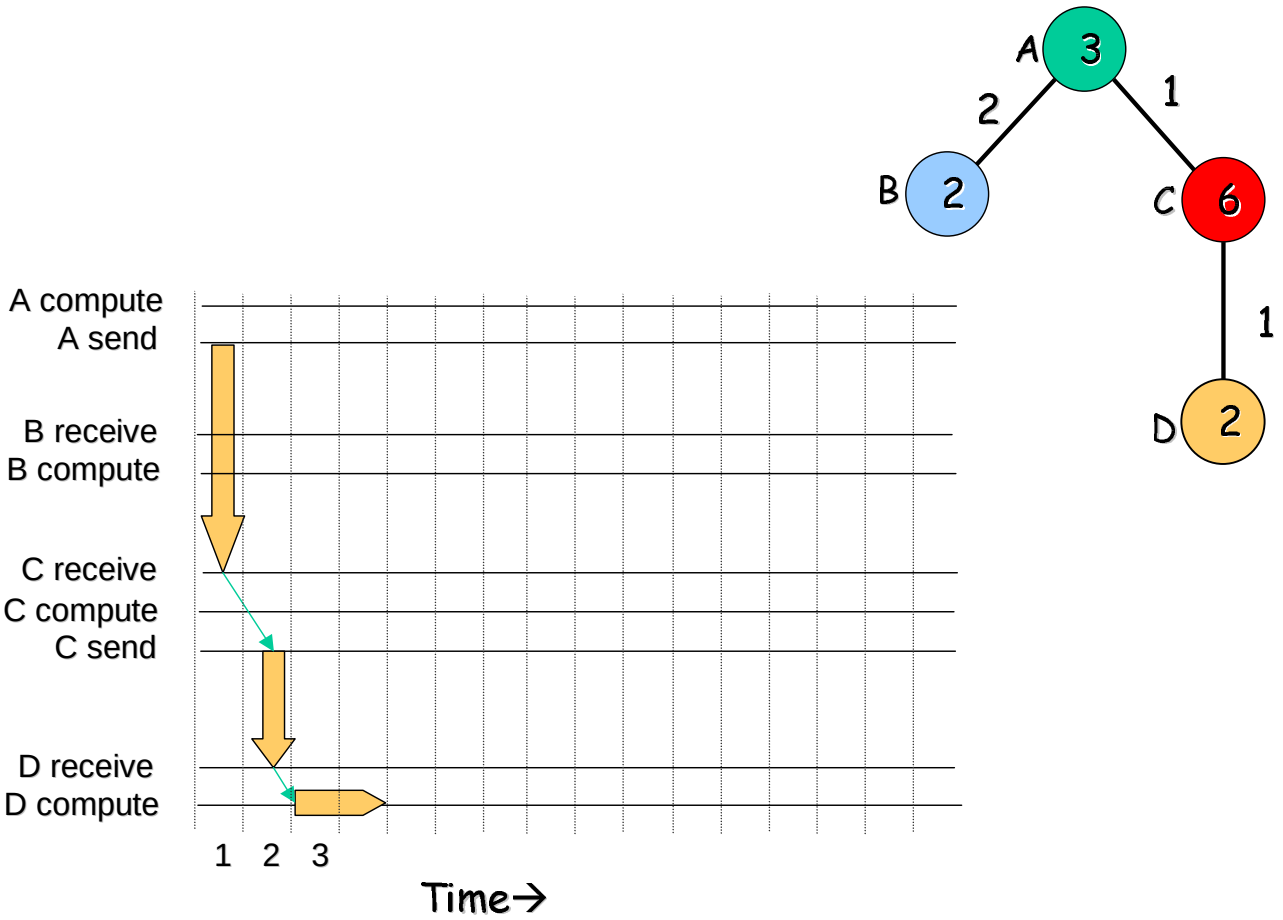
# Example


Data starts here
My computer
Intermediate nodes can compute too
Internet Gateway
Cluster Host
Super-computer
Partner site
Participating PC's and workstations

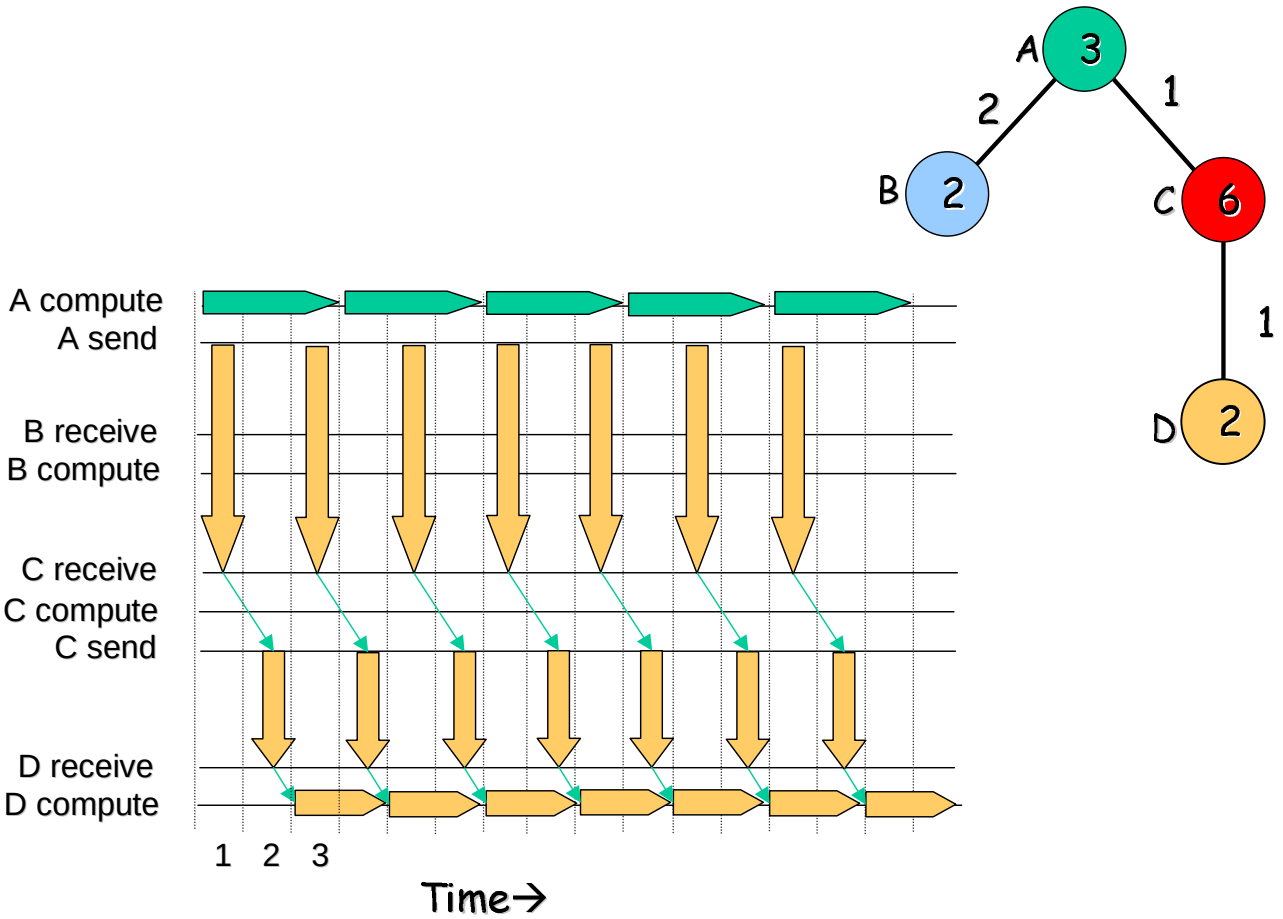
# Example


A is the root of the tree;
all tasks start at A
Time for sending
one task from A to B
A
3
2
1
B
2
C
6
Time for computing
one task in C
1
D
2

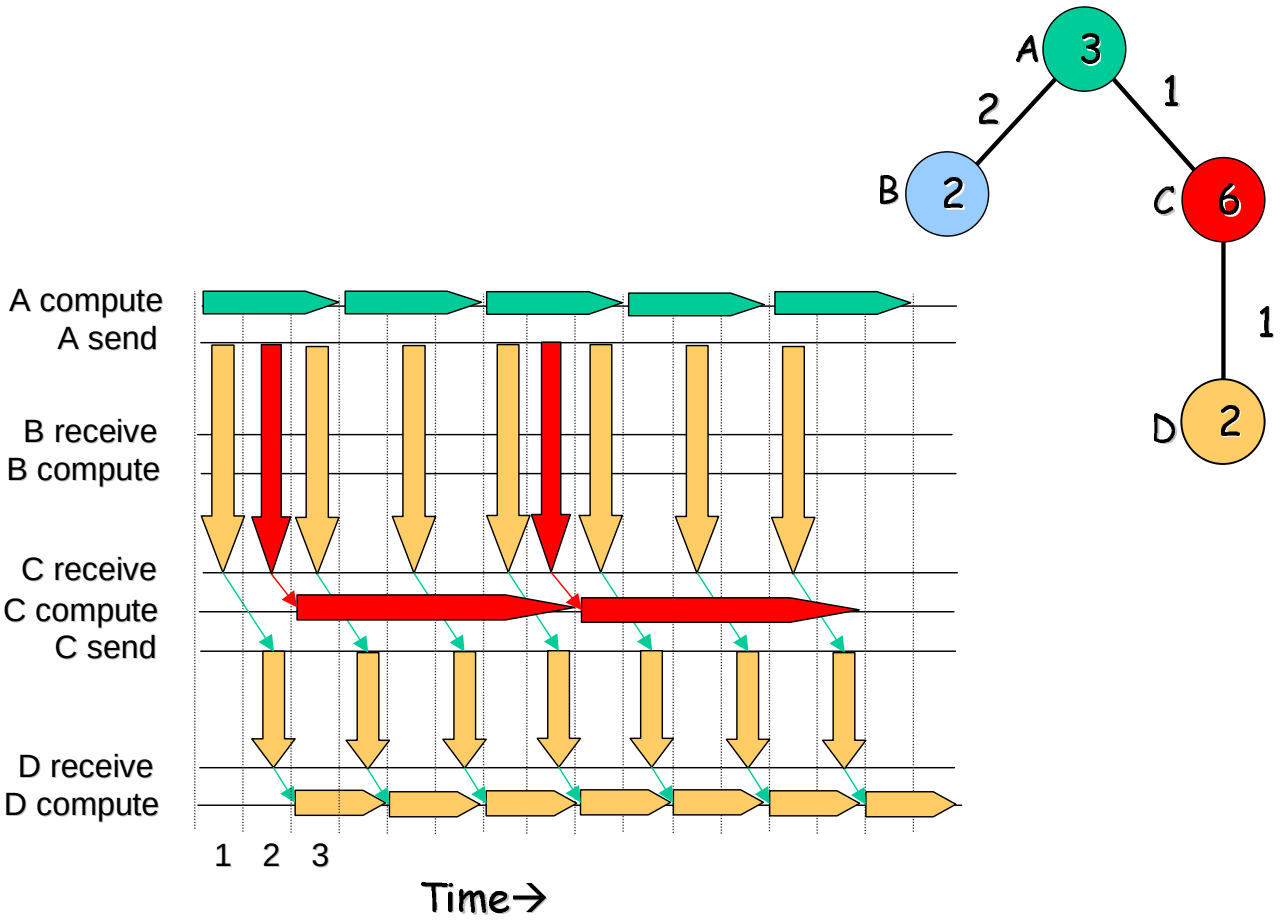
# Example


A
3
2
1
B
2
C
6
1
D
2
A compute
A send
B receive
B compute
C receive
C compute
C send
D receive
D compute
1 2 3
Time→

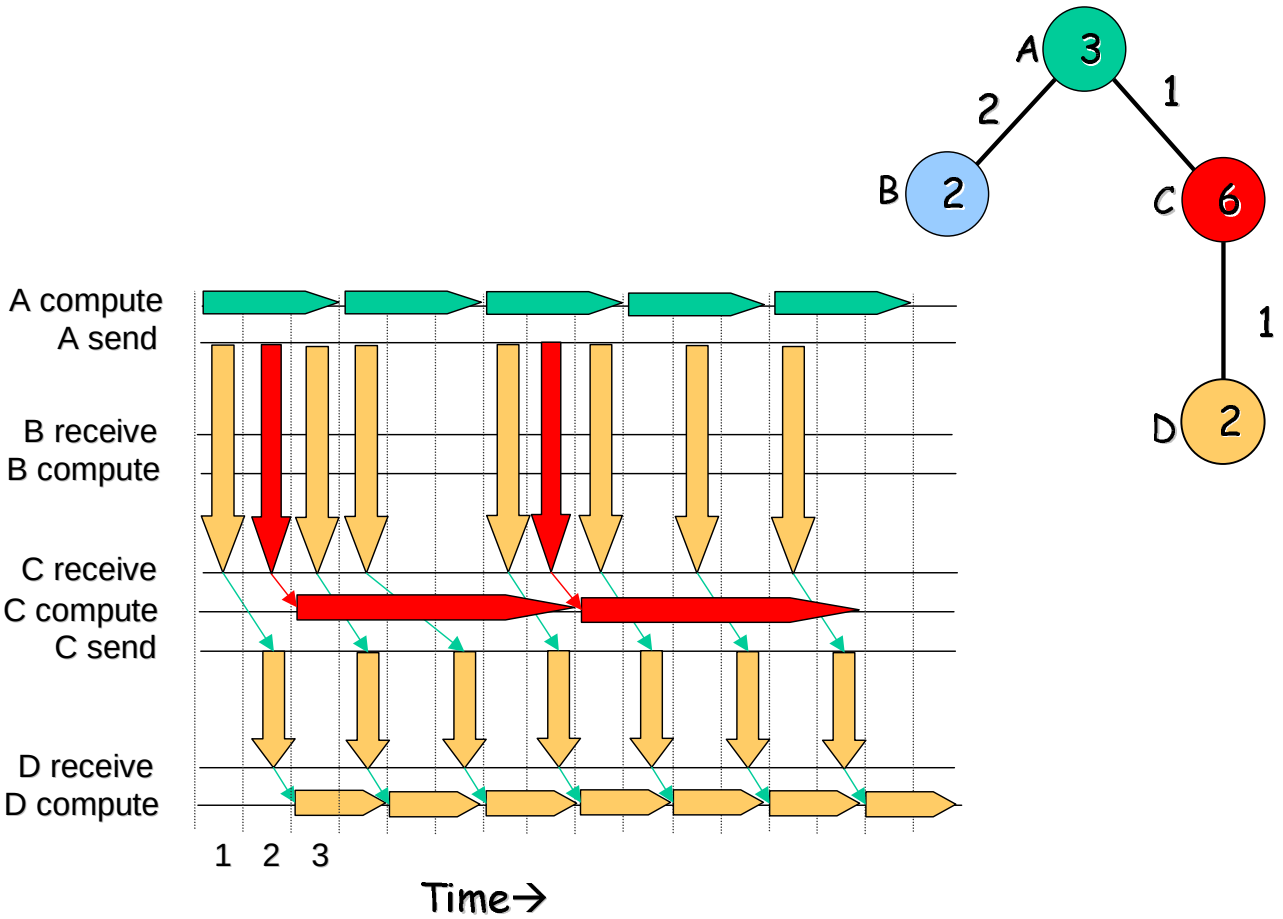
# Example


A
3
2
1
B
2
C
6
1
D
2
A compute
A send
B receive
B compute
C receive
C compute
C send
D receive
D compute
1 2 3
Time→

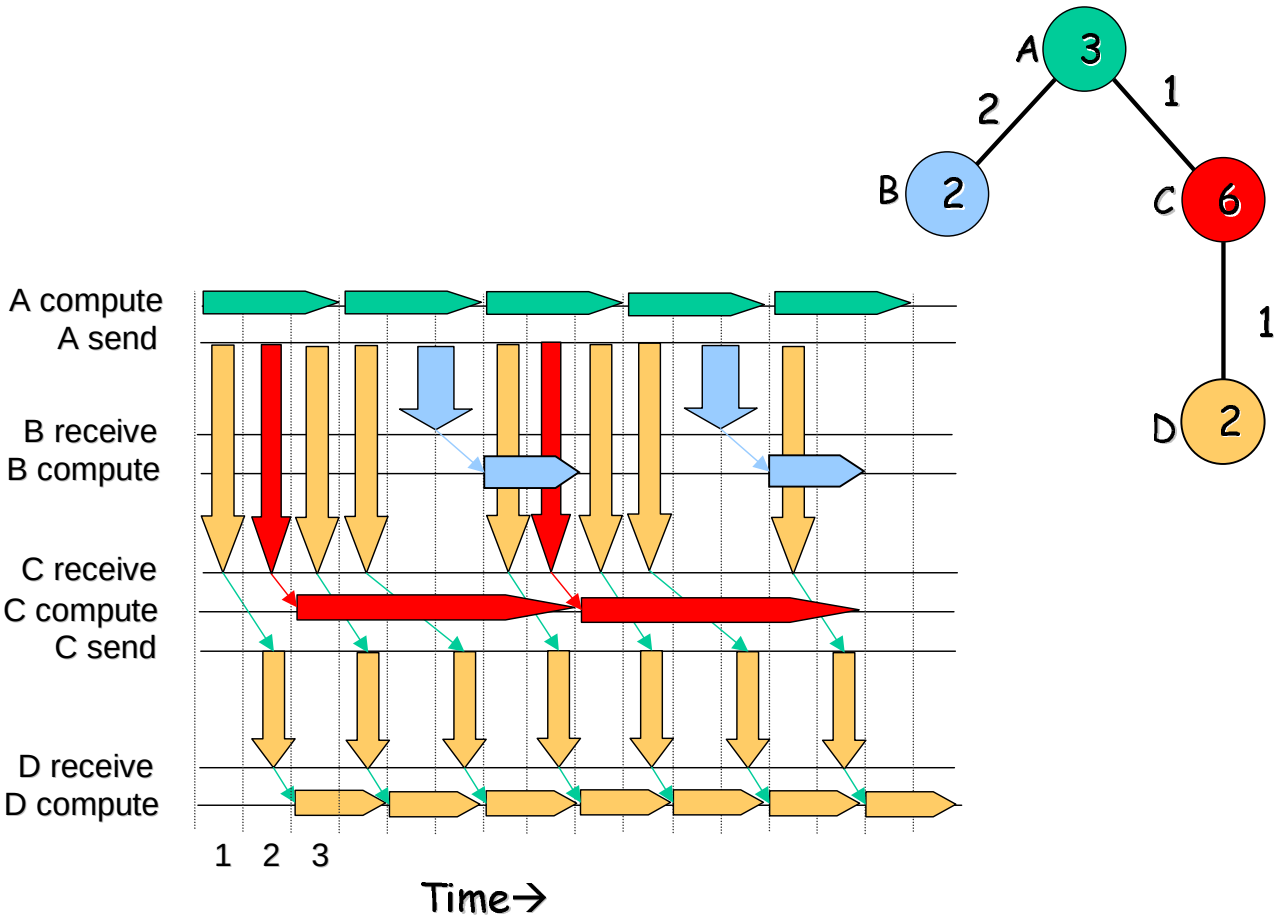
# Example


A
3
2
1
B
2
C
6
1
D
2
A compute
A send
B receive
B compute
C receive
C compute
C send
D receive
D compute
1 2 3
Time→

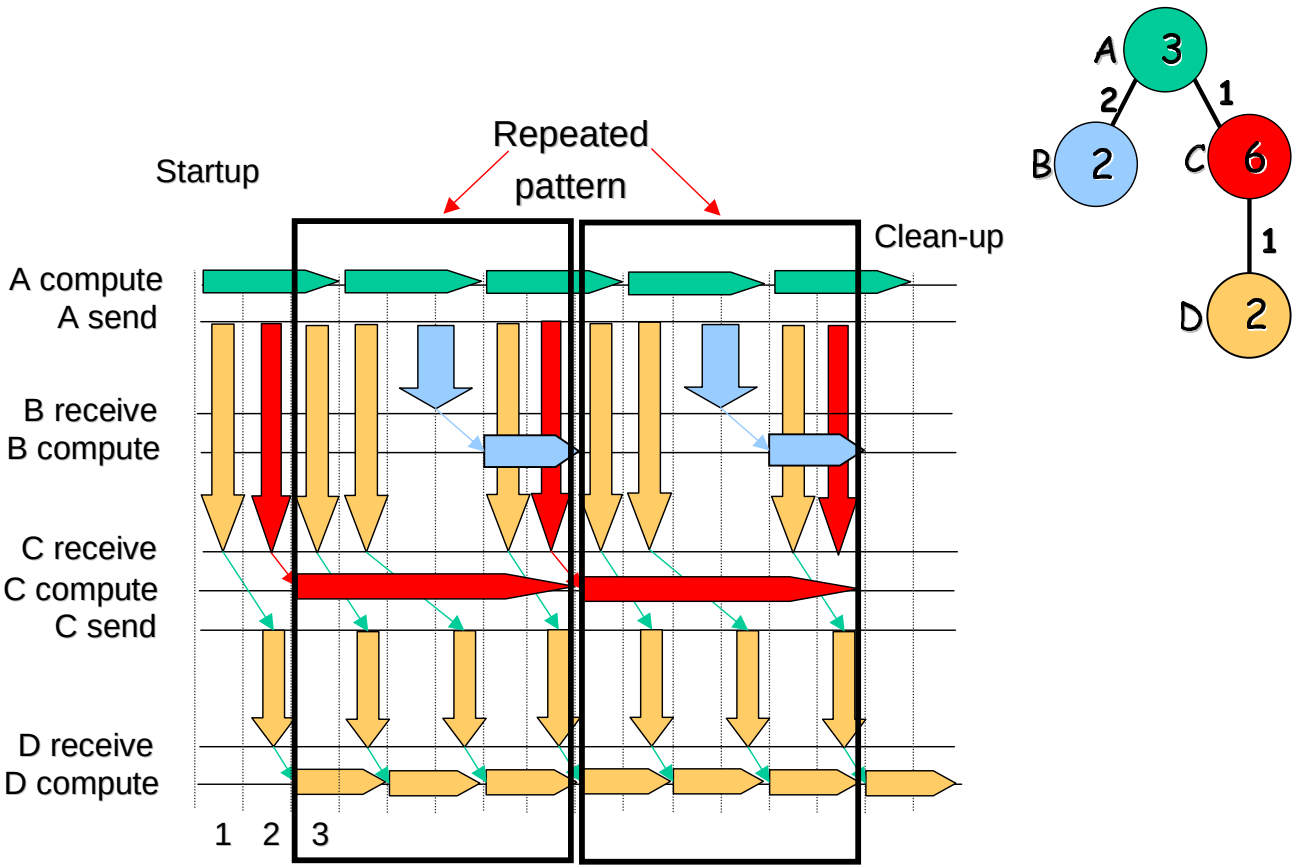
# Example


A
3
2
1
B
2
C
6
1
D
2
A compute
A send
B receive
B compute
C receive
C compute
C send
D receive
D compute
1 2 3
Time→

# Example

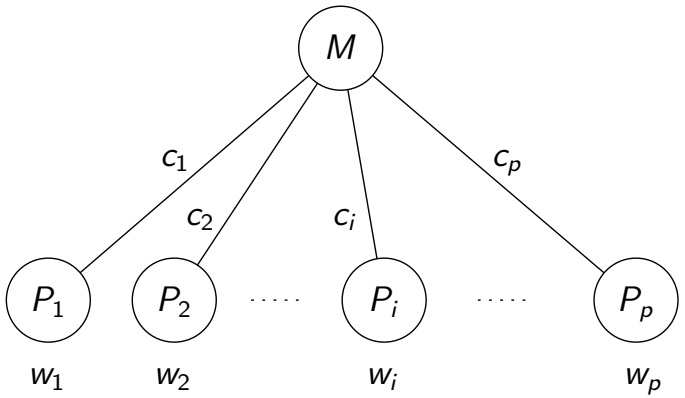

A
3
2
1
B
2
C
6
1
D
2
A compute
A send
B receive
B compute
C receive
C compute
C send
D receive
D compute
1 2 3
Time→

# Example



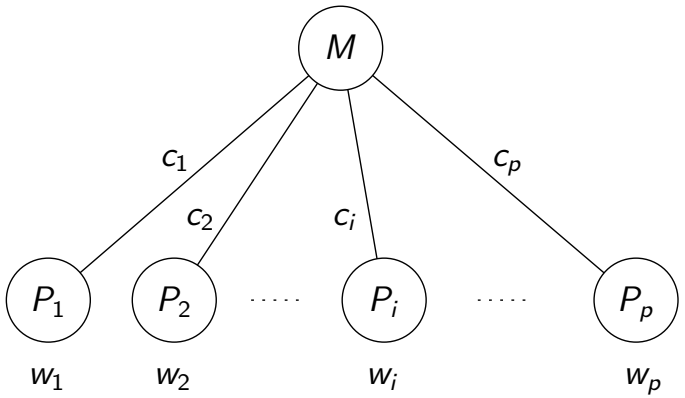

Steady-state: 7 tasks every 6 time units

# Rule of the game




- Master sends tasks to workers sequentially, and without preemption
- Full computation/communication overlap for each worker
- Worker $P_i$ receives a task in $c_i$ time-units
- Worker $P_i$ processes a task in $w_i$ time-units

# Equations




- Worker $P_i$ executes $\alpha_i$ tasks per time-unit
- Computations: $\alpha_i w_i \leq 1$
- Communications: $\sum_i \alpha_i c_i \leq 1$
- Objective: maximize throughput

$$\rho = \sum_i \alpha_i$$

# Solution

- Faster-communicating workers first: $c_1 \leq c_2 \leq \ldots$
- Make full use of first $q$ workers, where $q$ largest index s.t.

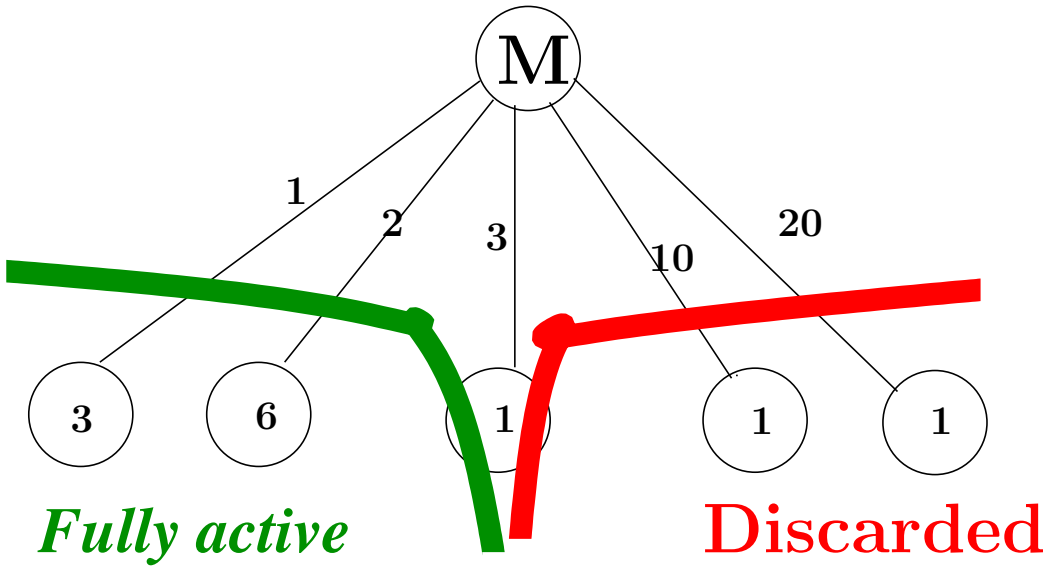
$$\sum_{i=1}^{q} \frac{c_i}{w_i} \leq 1$$

- Make partial use of next worker $P_{q+1}$
- **Discard** other workers

**Bandwidth-centric strategy**

- Delegate work to the fastest communicating workers
- It doesn't matter if these workers are computing slowly
- Slow workers will not contribute much to overall throughput

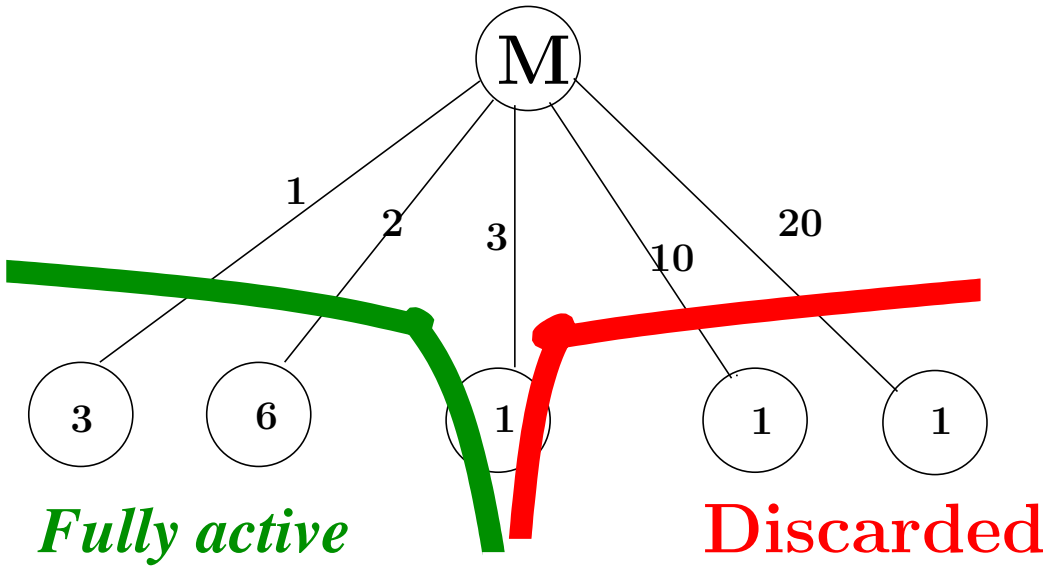
# Example




| Tasks | Communication | Computation |
|---|---|---|
| 6 tasks to $P_1$ | $6c_1 = 6$ | $6w_1 = 18$ |
| 3 tasks to $P_2$ | $3c_2 = 6$ | $3w_2 = 18$ |
| 2 tasks to $P_3$ | $2c_3 = 6$ | $2w_3 = \mathbf{2}$ |

11 tasks every 18 time-units ($\rho = 11/18 \approx 0.6$)

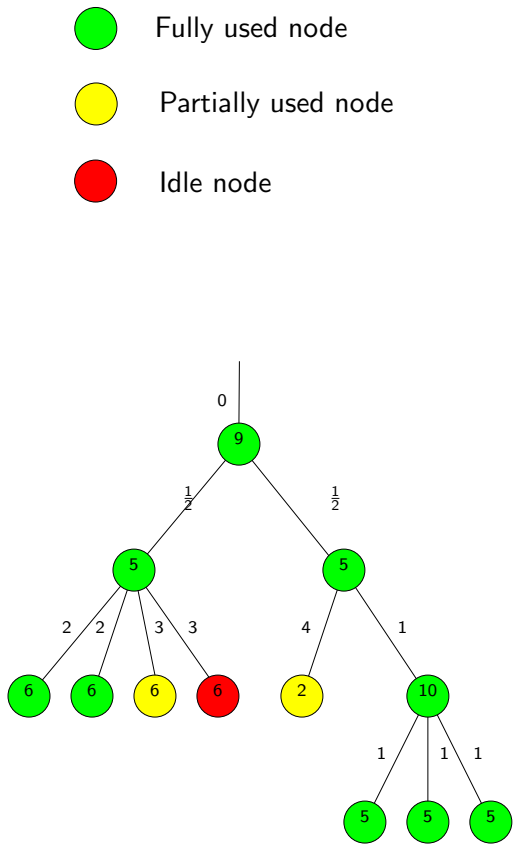
# Example




☺ Compare to purely greedy (demand-driven) strategy!
5 tasks every 36 time-units ($\rho = 5/36 \approx 0.14$)

Even if resources are cheap and abundant,
resource selection is key to performance

# Extension to trees




Resource selection based on **local** information (children)

# Does this really work?

- Can we deal with arbitrary platforms (including cycles)?
- Can we deal with return messages?
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)?

# Does this really work?

- Can we deal with arbitrary platforms (including cycles)? Yes
- Can we deal with return messages?
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)?

# Does this really work?

- Can we deal with arbitrary platforms (including cycles)? Yes
- Can we deal with return messages?
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)?

# Does this really work?

- Can we deal with arbitrary platforms (including cycles)? Yes
- Can we deal with return messages? Yes
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)?

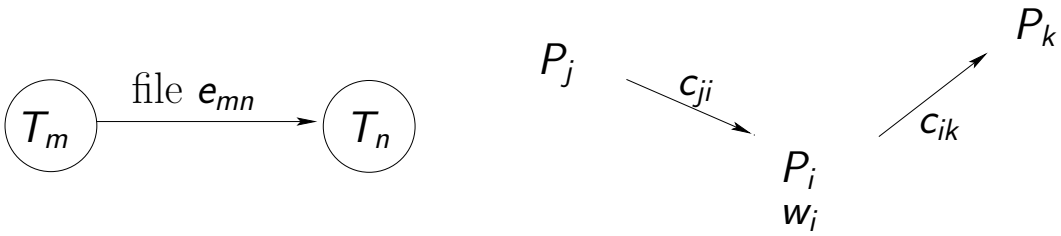
# Does this really work?

- Can we deal with arbitrary platforms (including cycles)? Yes
- Can we deal with return messages? Yes
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)?

# Does this really work?

- Can we deal with arbitrary platforms (including cycles)? Yes
- Can we deal with return messages? Yes
- In fact, can we deal with more complex applications (arbitrary collections of DAGs)? Yes, I mean, almost!

# LP formulation still works well ...




## Conservation law

$$\forall m, n \quad \sum_j \text{sent}(P_j \to P_i, e_{mn}) + \text{executed}(P_i, T_m) = \text{executed}(P_i, T_n) + \sum_k \text{sent}(P_i \to P_k, e_{mn})$$

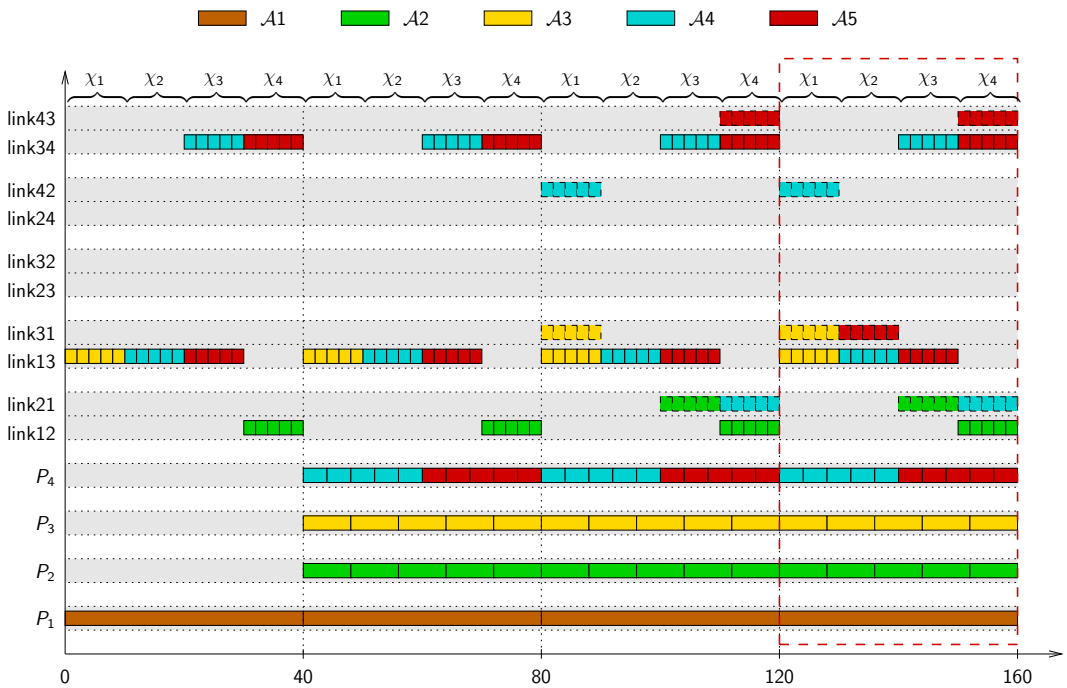
## Computations

$$\sum_m \text{executed}(P_i, T_m) \times \text{flops}(T_m) \times w_i \leq 1$$

## Outgoing communications

$$\sum_{m,n} \sum_j \text{sent}(P_j \to P_i, e_{mn}) \times \text{bytes}(e_{mn}) \times c_{ij} \leq 1$$

# . . . but schedule reconstruction is harder




- ☺ Actual (cyclic) schedule obtained in polynomial time
- ☺ Asymptotic optimality
- ☹ A couple of practical problems (large period, # buffers)
- ☹ No **local** scheduling policy

# The beauty of steady-state scheduling

**Rationale** Maximize throughput (total load executed per period)

**Simplicity** Relaxation of makespan minimization problem

- Ignore initialization and clean-up phases
- Precise ordering of tasks/messages not needed
- Characterize resource activity per time-unit:
  - which (rational) fraction of time is spent computing for which application?
  - which (rational) fraction of time is spent receiving from or sending to which neighbor?

**Efficiency** Optimal throughput $\Rightarrow$ optimal schedule (up to a constant number of tasks)

Periodic schedule, described in compact form
$\Rightarrow$ compiling a loop instead of a DAG!

# Lesson learnt?

**Resource selection is mandatory**

. . . implementation may still be dynamic,
provided that static allocation is enforced by scheduler

**Example:** demand-driven assignment of enrolled workers

# Lesson learnt?

Resource selection is mandatory

. . . implementation may still be dynamic,
provided that static allocation is enforced by scheduler

**Example:** demand-driven assignment of enrolled workers

# Outline

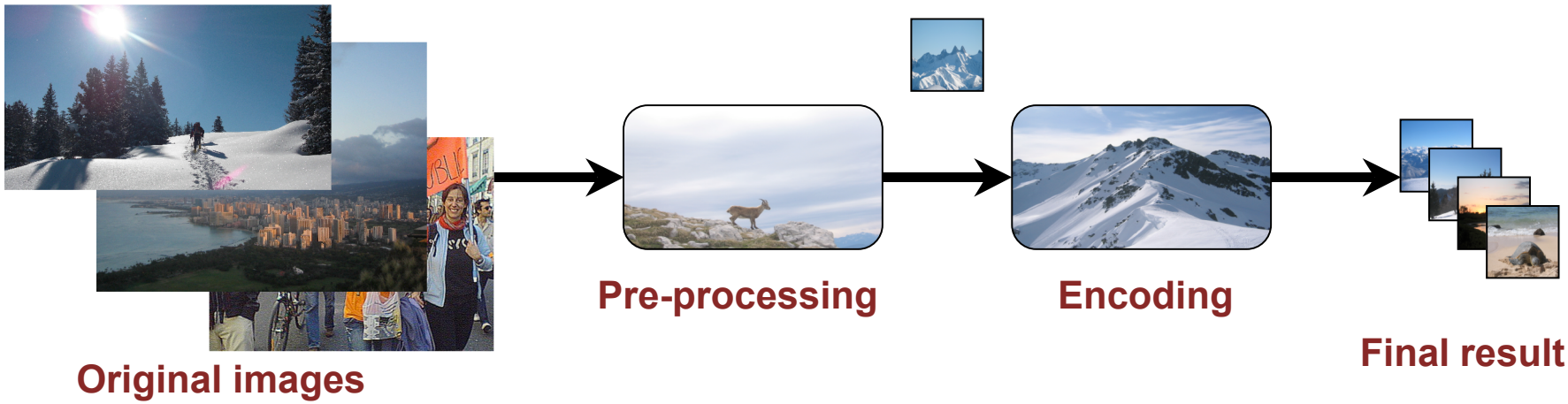
1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Scheduling pipelined applications

- Data streams: images, frames, matrices, etc.
- Structured applications: several steps to process each data set



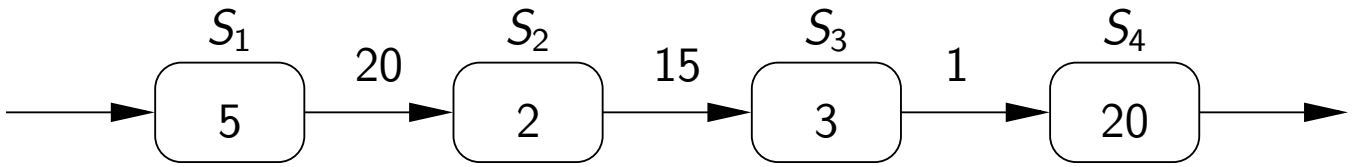

Goal: "efficiently" use computing resources

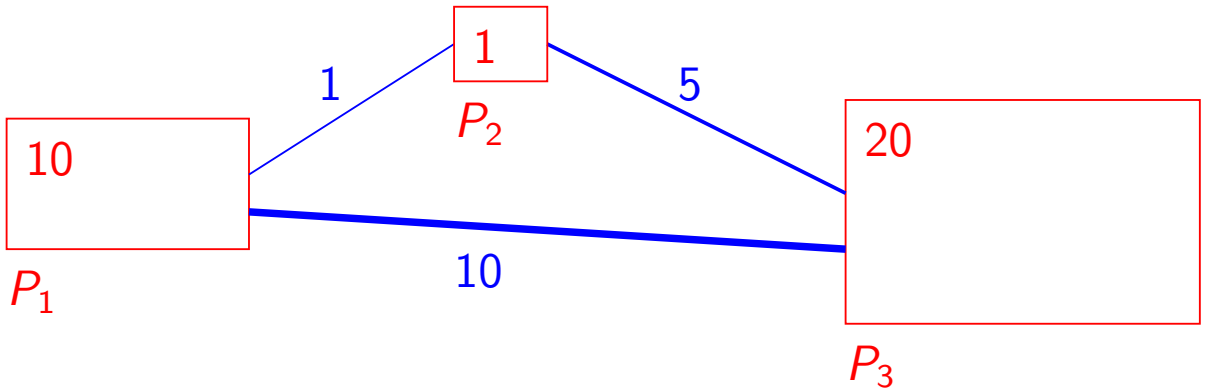
# Example

- 4 processing stages, 3 processors at our disposal
- Where/how can we execute the application?

# Example

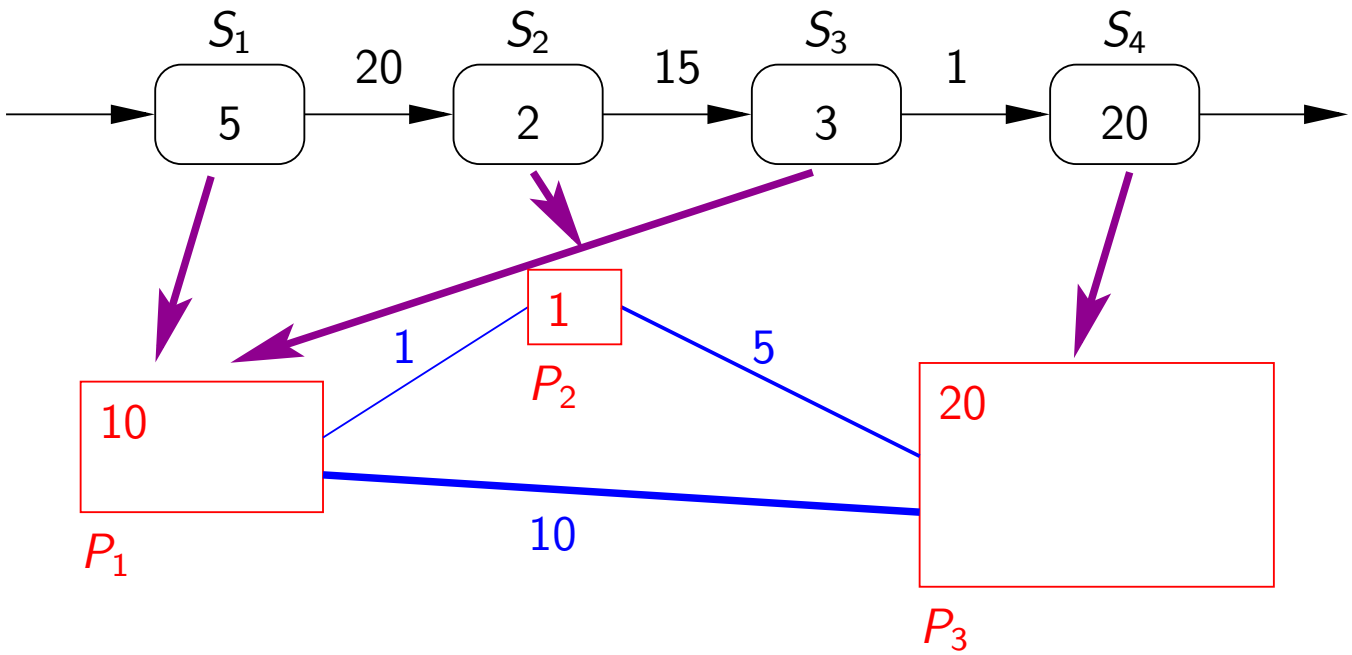
- 4 processing stages, 3 processors at our disposal
- Where/how can we execute the application?



# Example

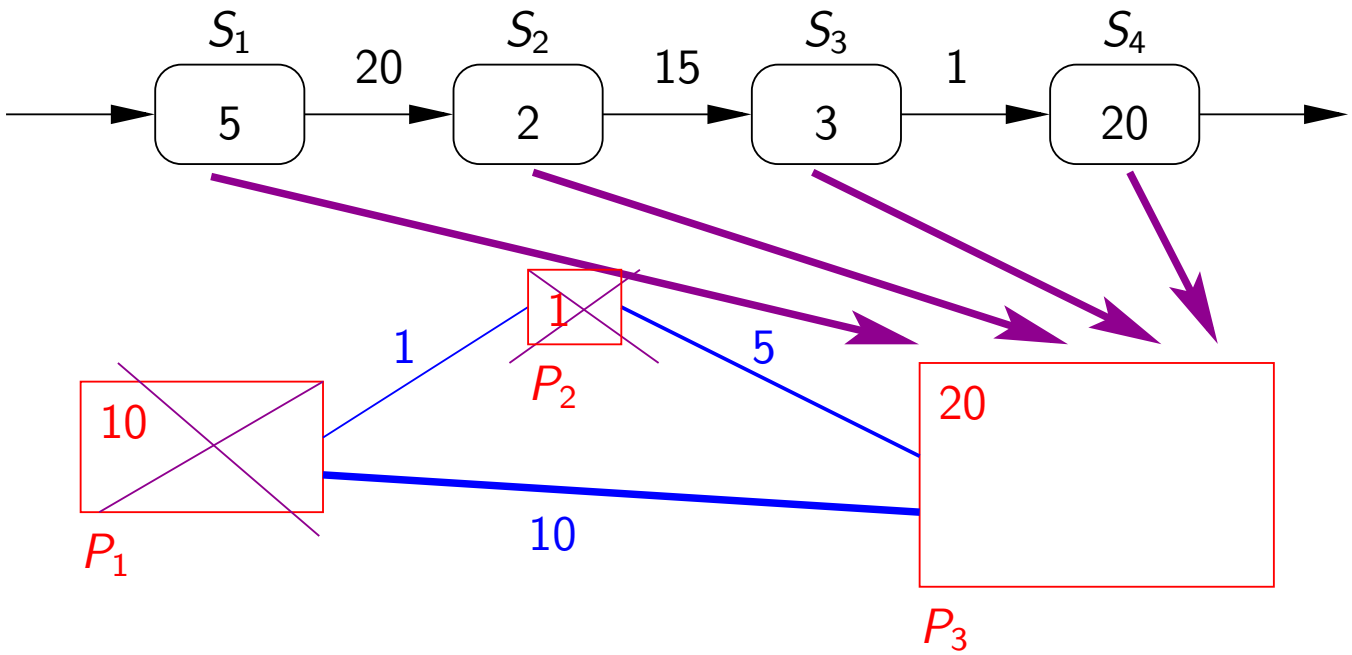
- 4 processing stages, 3 processors at our disposal
- Where/how can we execute the application?




- Use all resources greedily
- Many communications to pay, not efficient at all!

# Example

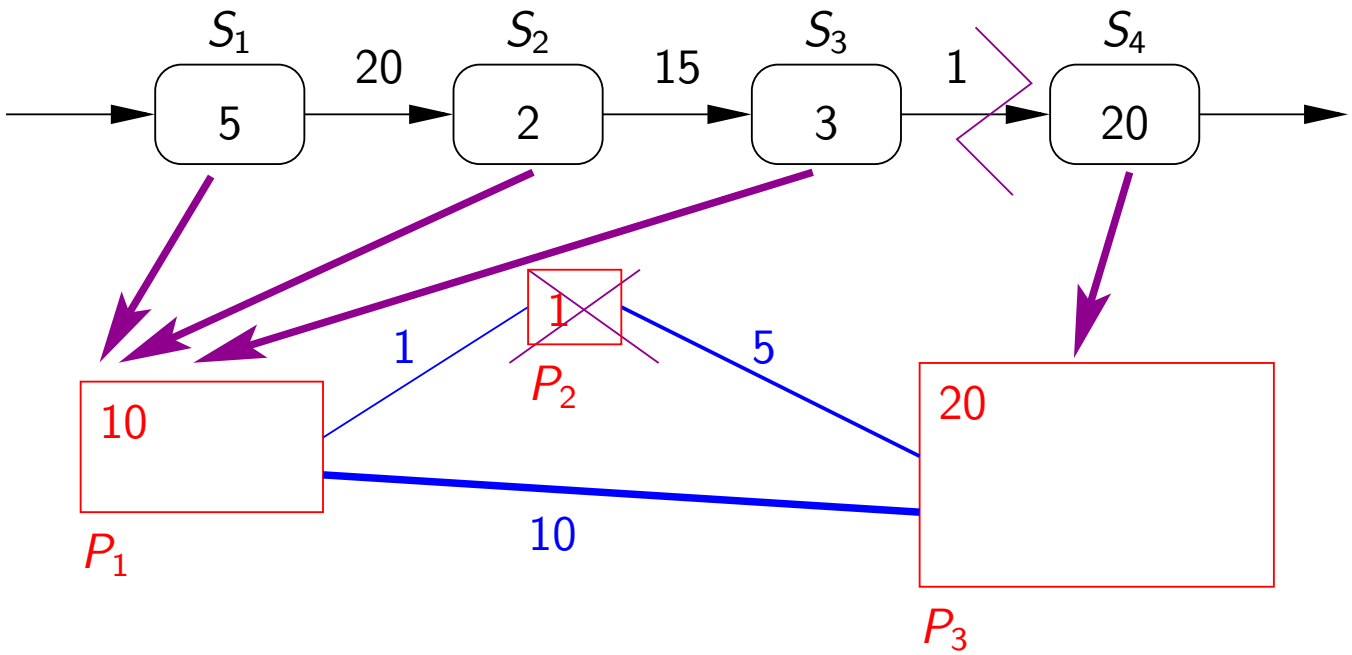
- 4 processing stages, 3 processors at our disposal
- Where/how can we execute the application?




- Everything on fastest processor: no communication
- Optimal execution time to process one single data

# Example

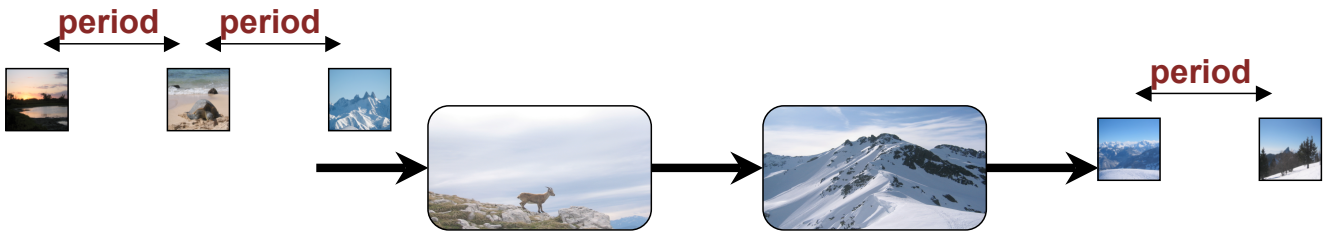
- 4 processing stages, 3 processors at our disposal
- Where/how can we execute the application?




- Optimal throughput: processing different data in parallel
- Resource selection: do not use slowest processor

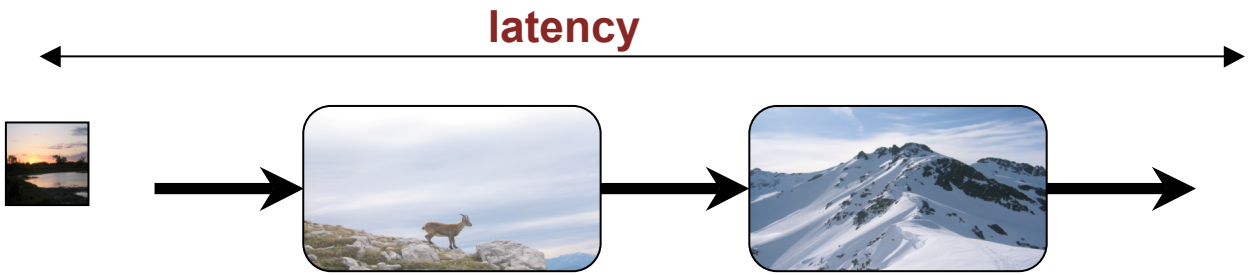
# Optimization criteria

- Period $\mathcal{P}$: time interval between the beginning of execution of two consecutive data sets (inverse of throughput)
- Latency $\mathcal{L}$: maximal time elapsed between beginning and end of execution of a data set
- Reliability: inverse of $\mathcal{F}$, probability of failure of the application (i.e., some data sets will not be processed)
- Energy $\mathcal{E}$: total power dissipated by platform



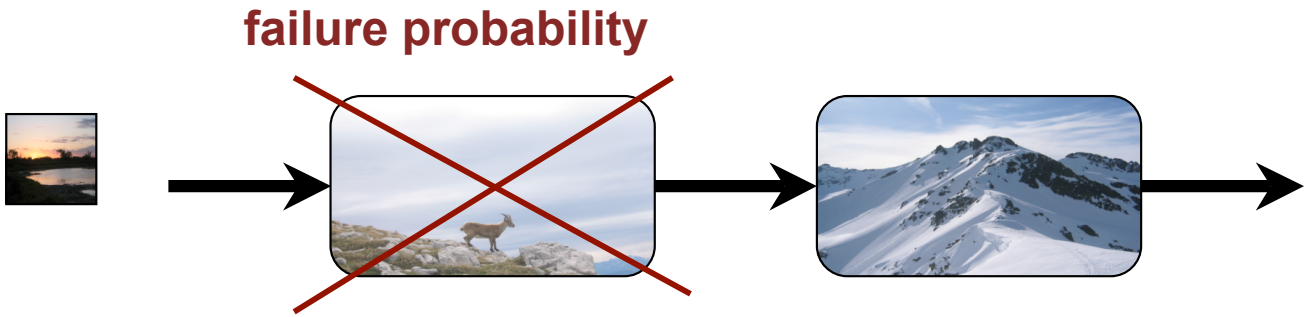
# Optimization criteria

- Period $\mathcal{P}$: time interval between the beginning of execution of two consecutive data sets (inverse of throughput)
- Latency $\mathcal{L}$: maximal time elapsed between beginning and end of execution of a data set
- Reliability: inverse of $\mathcal{F}$, probability of failure of the application (i.e., some data sets will not be processed)
- Energy $\mathcal{E}$: total power dissipated by platform



# Optimization criteria

- Period $\mathcal{P}$: time interval between the beginning of execution of two consecutive data sets (inverse of throughput)
- Latency $\mathcal{L}$: maximal time elapsed between beginning and end of execution of a data set
- Reliability: inverse of $\mathcal{F}$, probability of failure of the application (i.e., some data sets will not be processed)
- Energy $\mathcal{E}$: total power dissipated by platform



# Optimization criteria

- Period $\mathcal{P}$: time interval between the beginning of execution of two consecutive data sets (inverse of throughput)
- Latency $\mathcal{L}$: maximal time elapsed between beginning and end of execution of a data set
- Reliability: inverse of $\mathcal{F}$, probability of failure of the application (i.e., some data sets will not be processed)
- Energy $\mathcal{E}$: total power dissipated by platform

**"The internet begins with coal"**

# A primer on energy consumption

Algorithmic techniques:

- Shut down idle processors
- Dynamic speed scaling: processors can run at variable speed, e.g., Intel XScale, Intel Speed Step, AMD PowerNow
- VDD Hopping: from discrete modes to continuous speed

Dissipated power

- The higher the speed, the higher the power consumption
- $Power = f \times V^2$, and $V$ (voltage) increases with $f$ (frequency)
- Speed $s$: $P(s) = s^{\alpha} + P_{static}$, with $2 \leq \alpha \leq 3$

# A primer on energy consumption

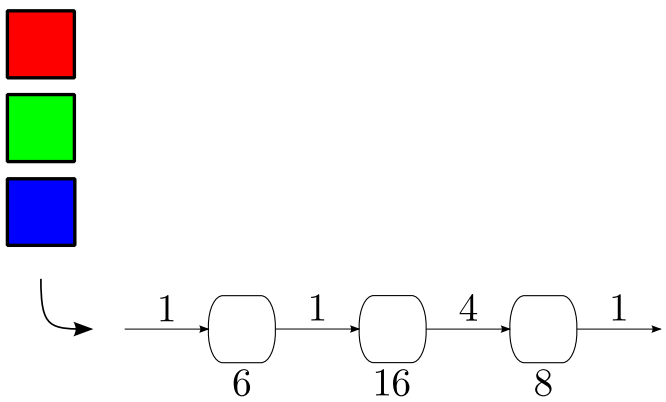
## Algorithmic techniques:

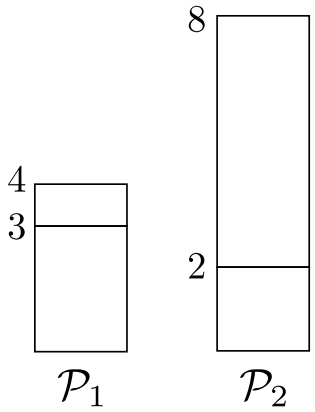
- Shut down idle processors
- Dynamic speed scaling: processors can run at variable speed, e.g., Intel XScale, Intel Speed Step, AMD PowerNow
- VDD Hopping: from discrete modes to continuous speed

## Dissipated power

- The higher the speed, the higher the power consumption
- $Power = f \times V^2$, and $V$ (voltage) increases with $f$ (frequency)
- Speed $s$: $P(s) = s^{\alpha} + P_{static}$, with $2 \leq \alpha \leq 3$

# Example





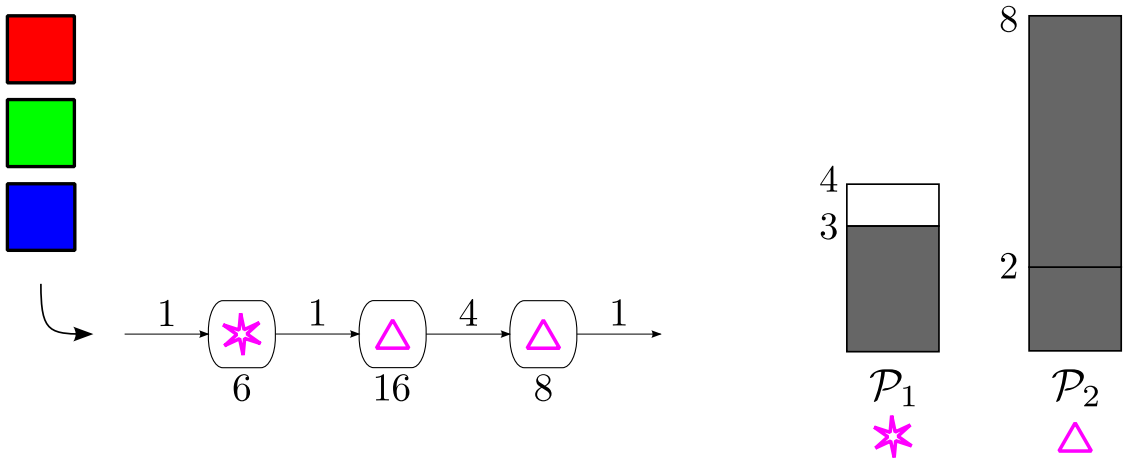


$\mathcal{P}_1$

$\mathcal{P}_2$

- Period: $T = 3$
- Latency: $L = 8$

# Example





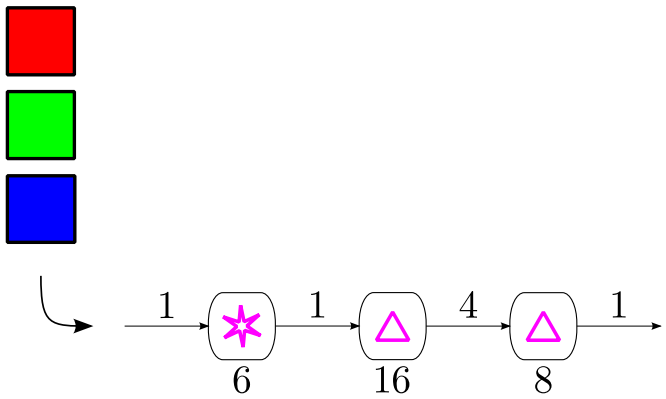


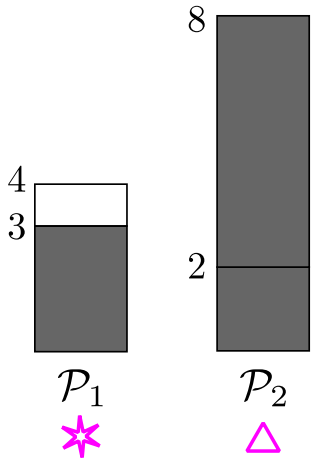
$\mathcal{P}_1$

$\mathcal{P}_2$

- Period: $T = 3$
- Latency: $L = 8$

# Example




$$P = 3^3 + 8^3$$
$$= 539$$

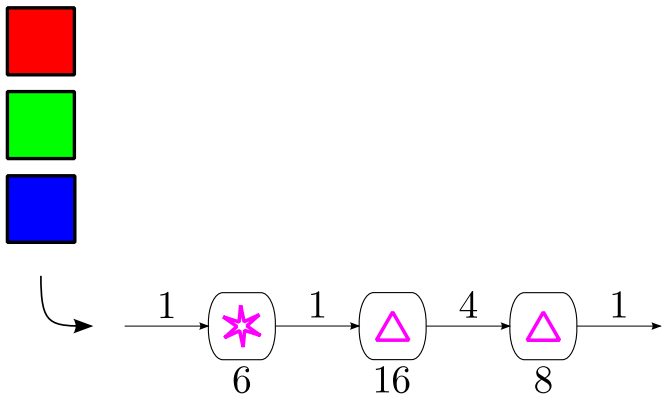


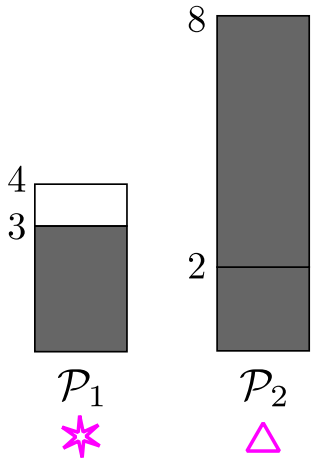

$\mathcal{P}_1$

$\mathcal{P}_2$

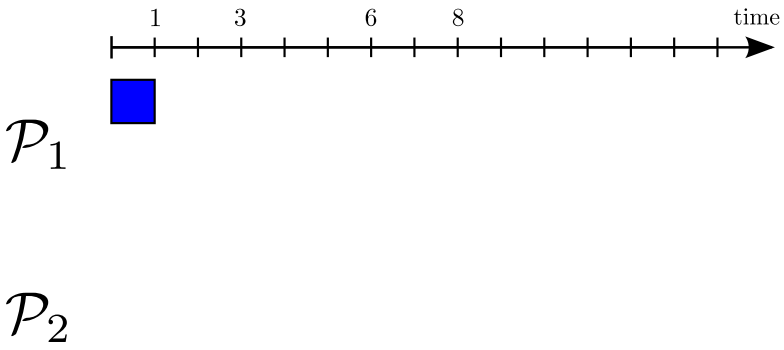
- Period: $T = 3$
- Latency: $L = 8$

# Example



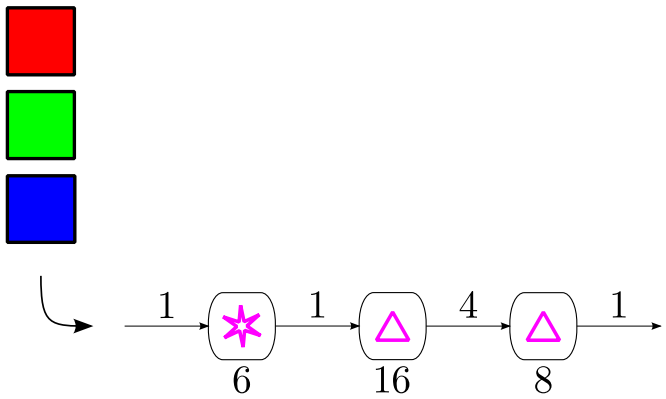



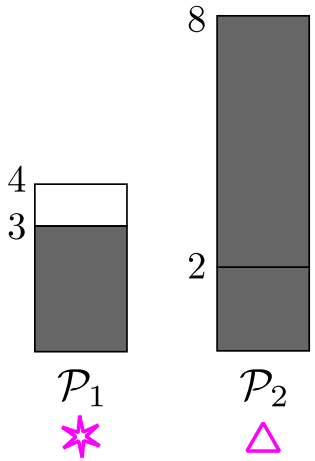

$$P = 3^3 + 8^3$$
$$= 539$$



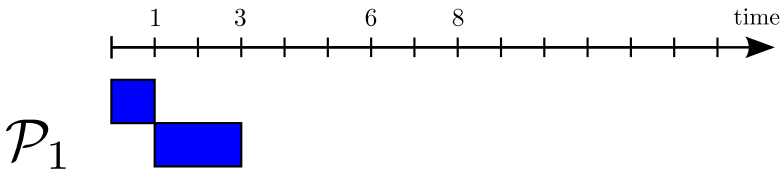

- Period: $T = 3$
- Latency: $L = 8$

# Example



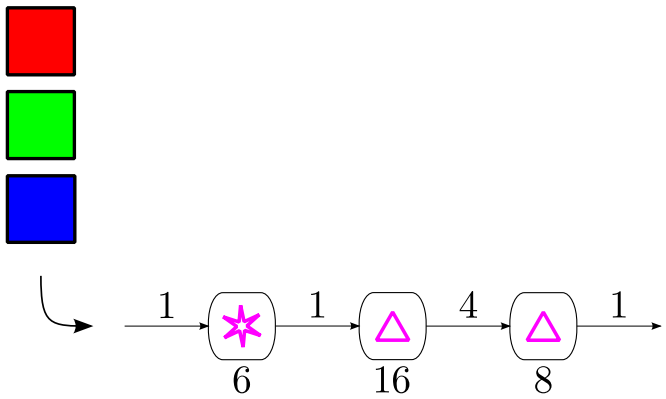

$$P = 3^3 + 8^3$$
$$= 539$$

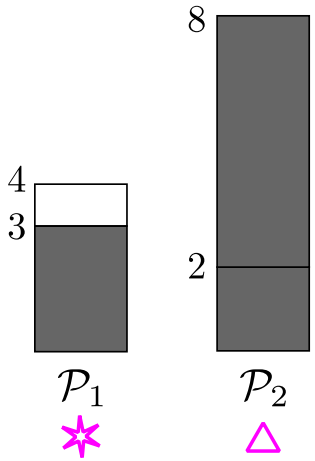



$\mathcal{P}_2$

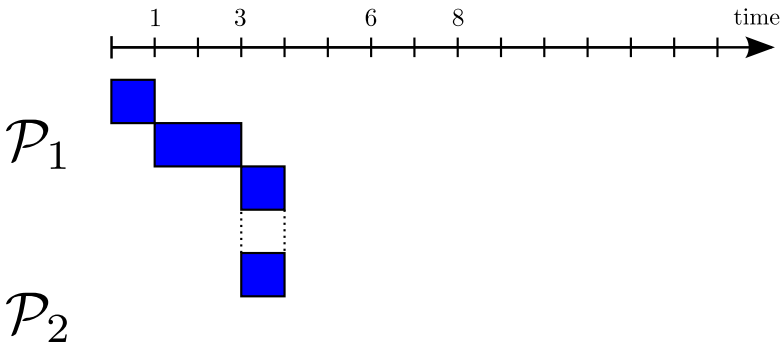
- Period: $T = 3$
- Latency: $L = 8$

# Example



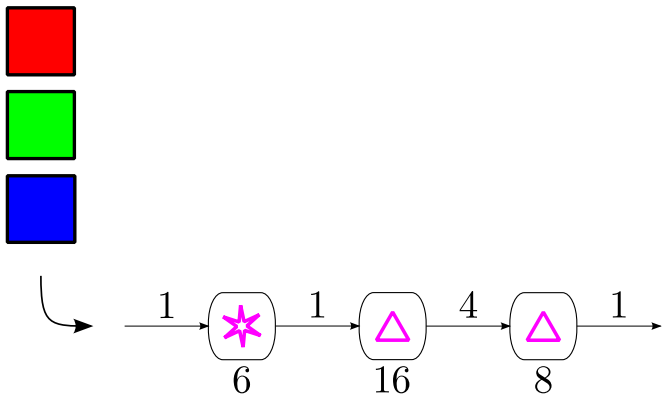



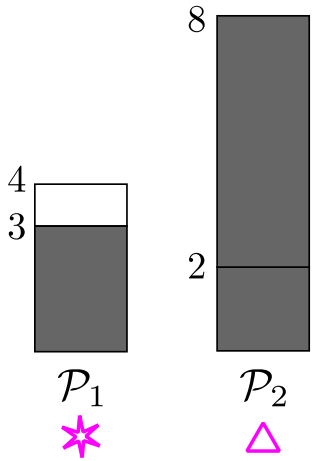

$$P = 3^3 + 8^3$$
$$= 539$$



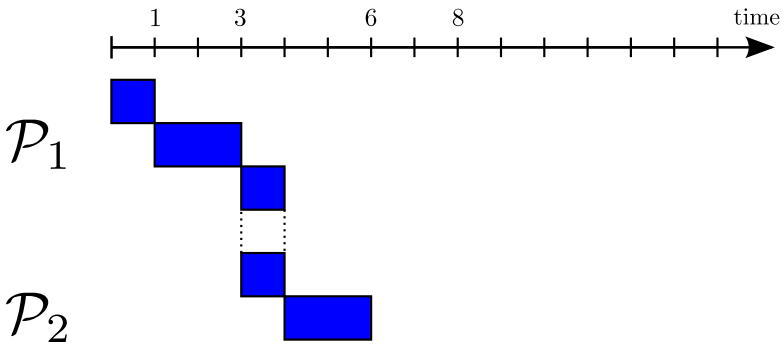

- Period: $T = 3$
- Latency: $L = 8$

# Example



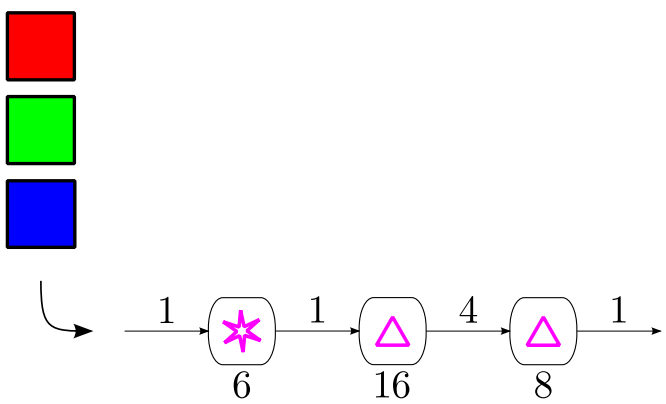



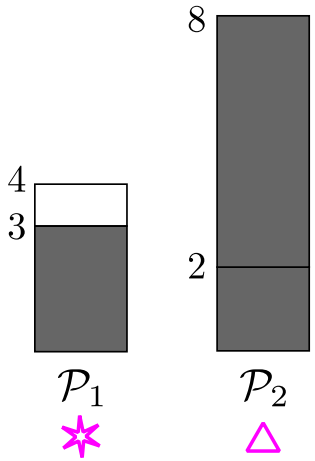

$$P = 3^3 + 8^3$$
$$= 539$$



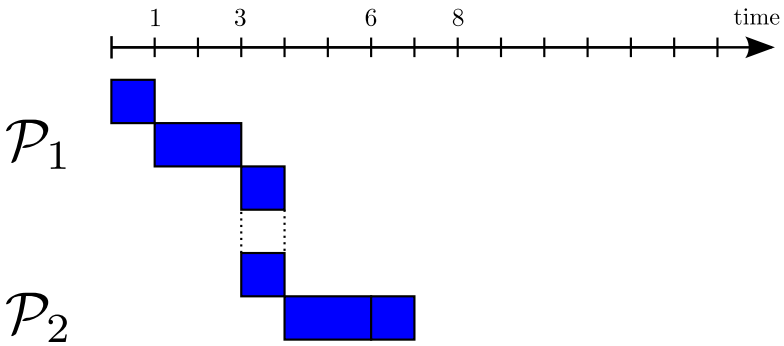

- Period: $T = 3$
- Latency: $L = 8$

# Example



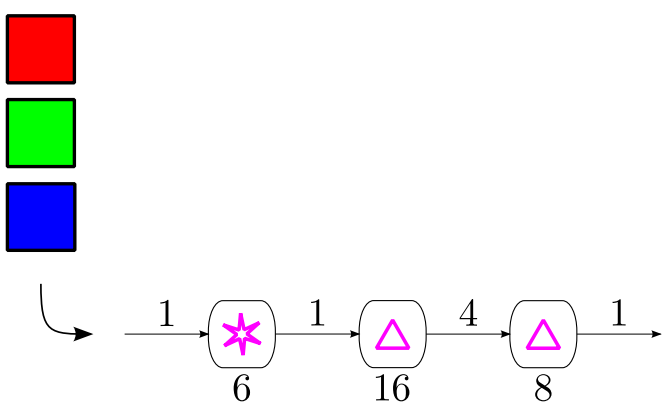



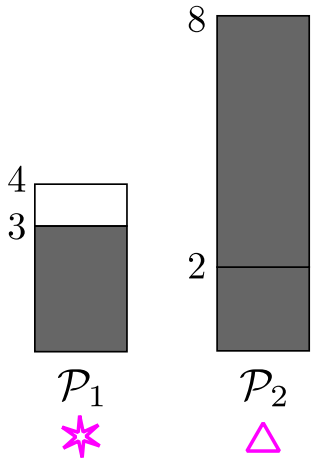

$$P = 3^3 + 8^3$$
$$= 539$$



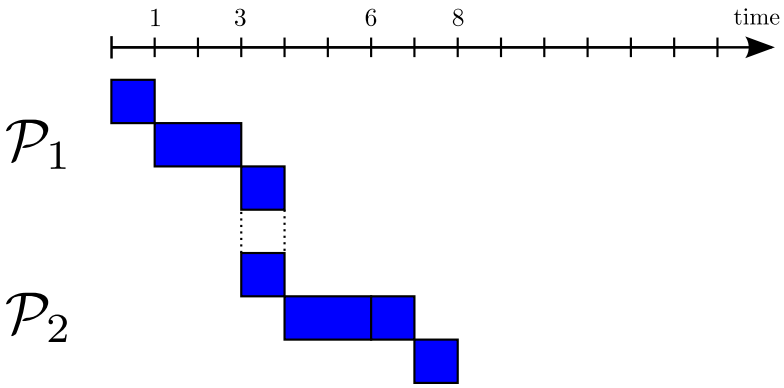

- Period: $T = 3$
- Latency: $L = 8$

# Example



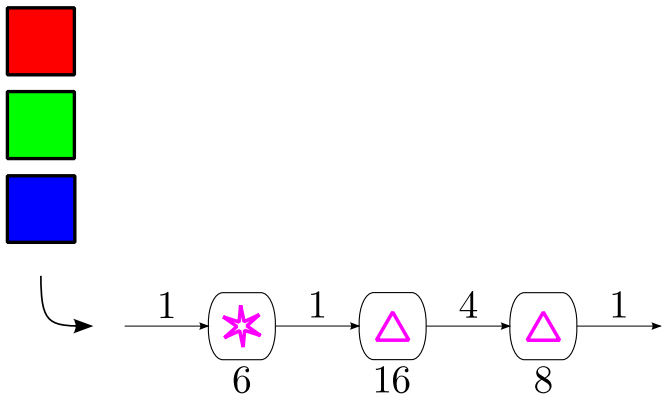



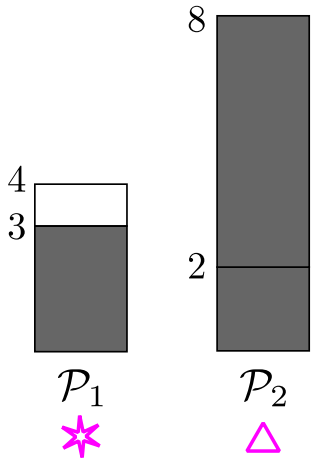

$$P = 3^3 + 8^3$$
$$= 539$$



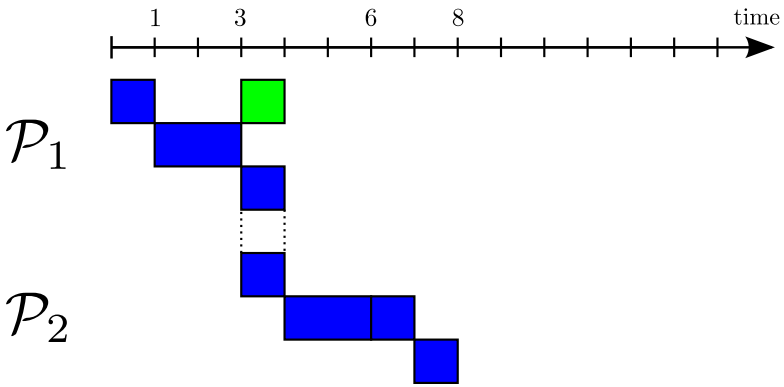

- Period: $T = 3$
- Latency: $L = 8$

# Example



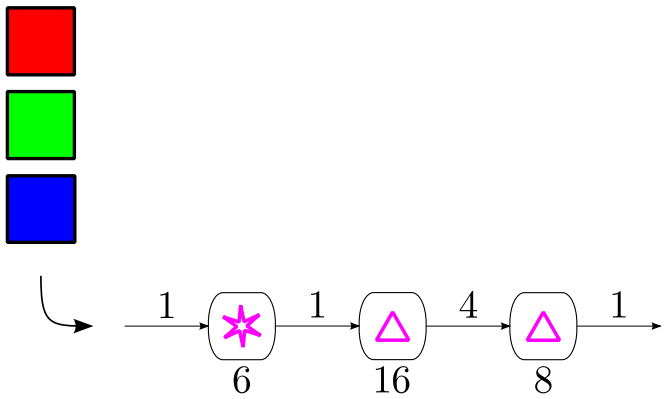



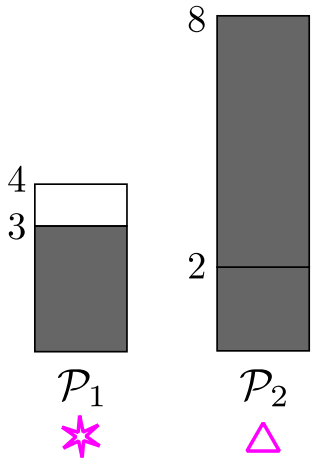

$$P = 3^3 + 8^3$$
$$= 539$$



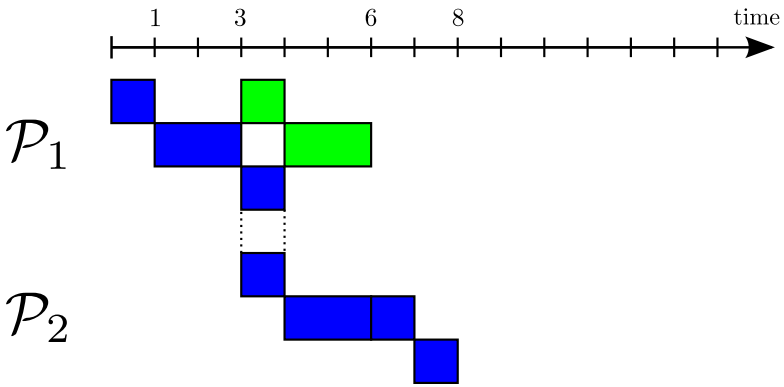

- Period: $T = 3$
- Latency: $L = 8$

# Example



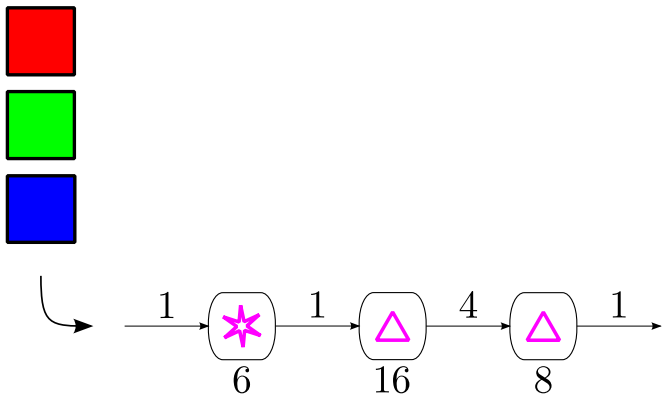



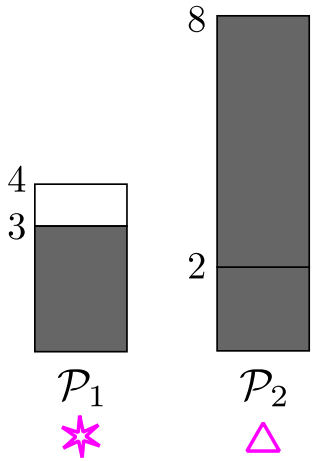

$$P = 3^3 + 8^3$$
$$= 539$$



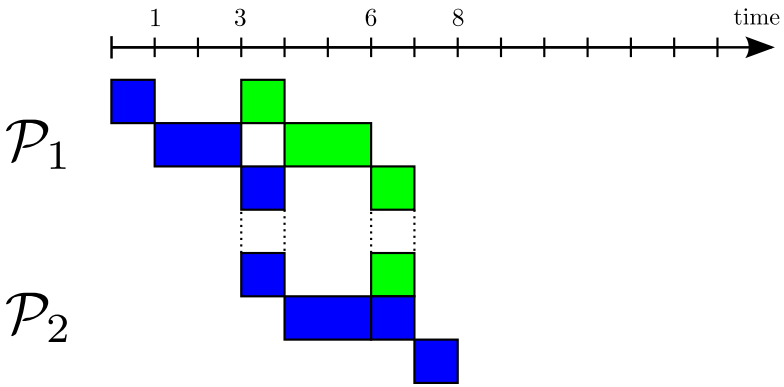

- Period: $T = 3$
- Latency: $L = 8$

# Example



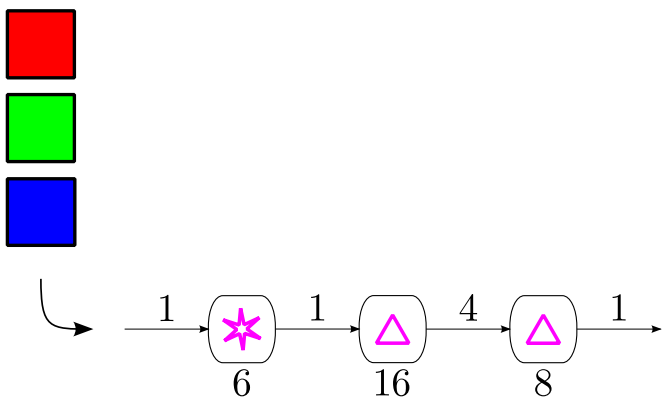



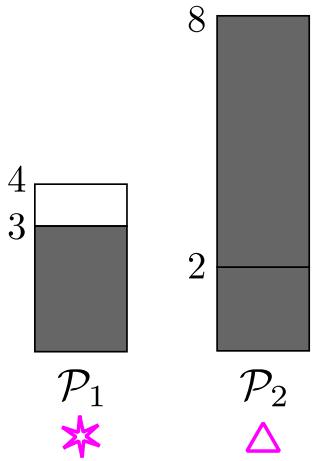

$$P = 3^3 + 8^3$$
$$= 539$$



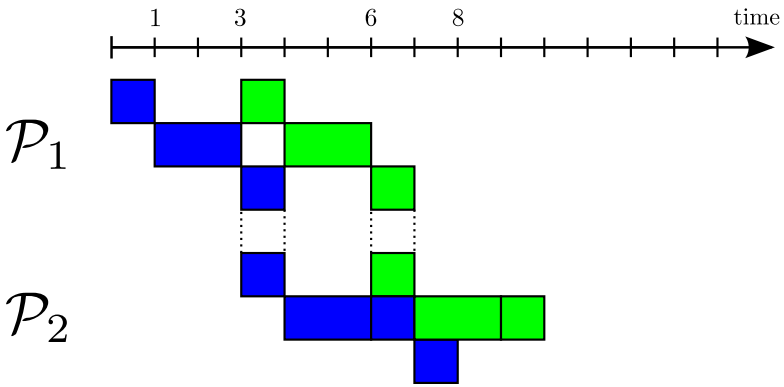

- Period: $T = 3$
- Latency: $L = 8$

# Example



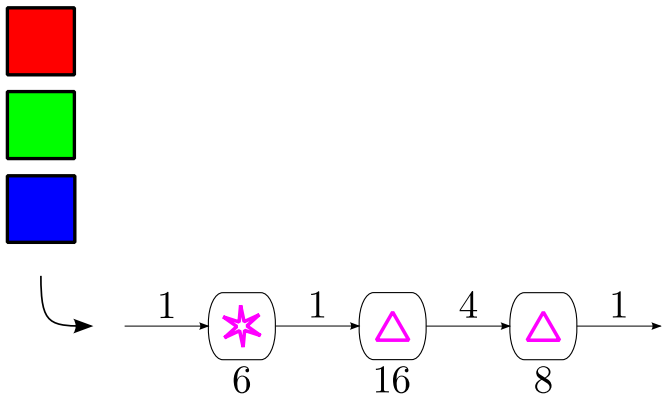



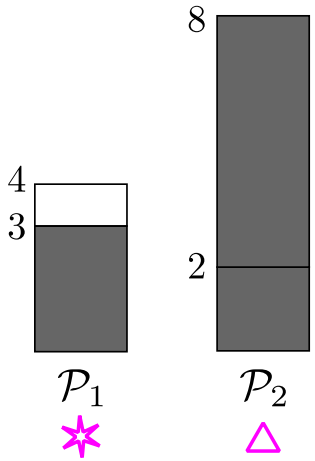

$$P = 3^3 + 8^3$$
$$= 539$$



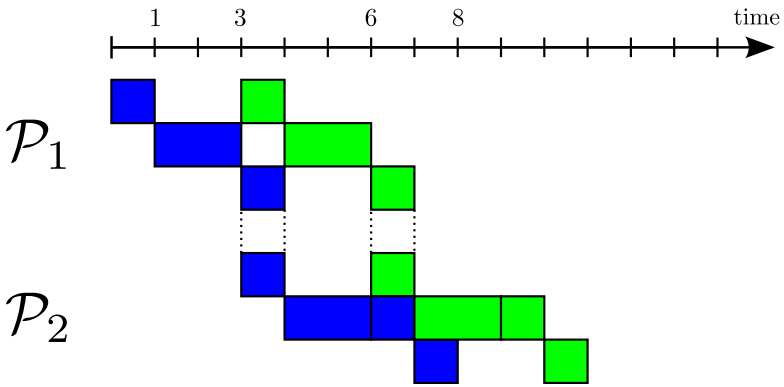

- Period: $T = 3$
- Latency: $L = 8$

# Example



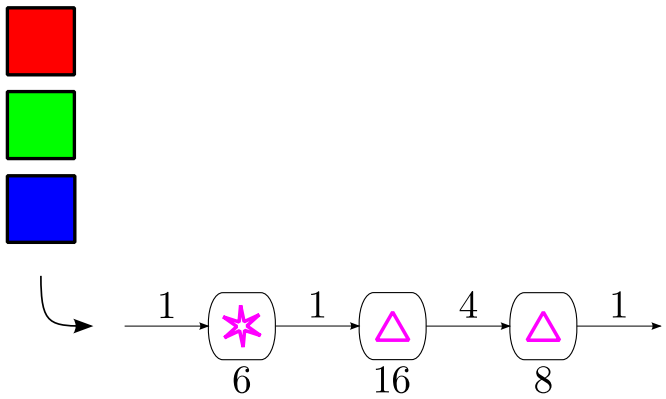



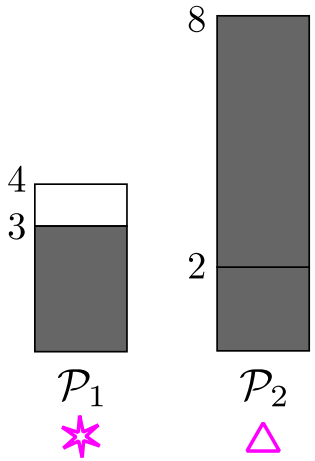

$$P = 3^3 + 8^3$$
$$= 539$$



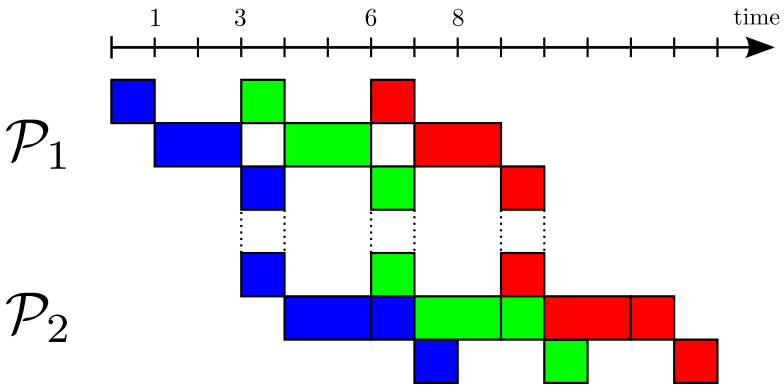

- Period: $T = 3$
- Latency: $L = 8$

# Example



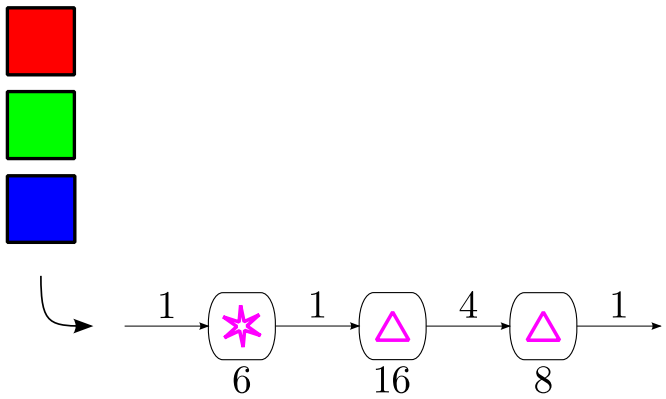



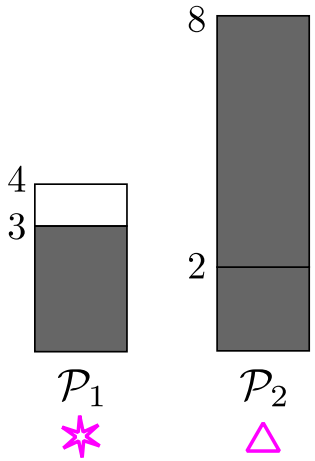

$$P = 3^3 + 8^3$$
$$= 539$$



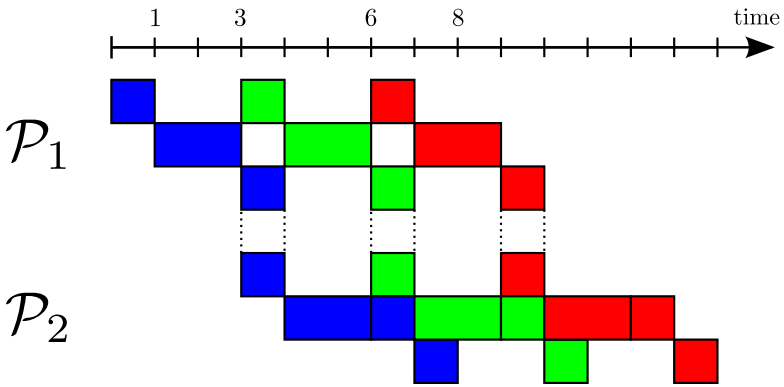

- Period: $T = 3$
- Latency: $L = 8$

# Example



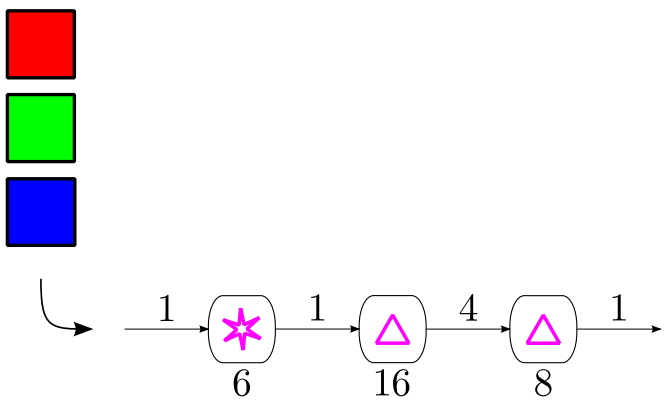



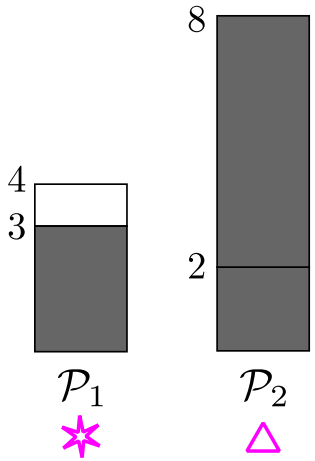

$$P = 3^3 + 8^3$$
$$= 539$$



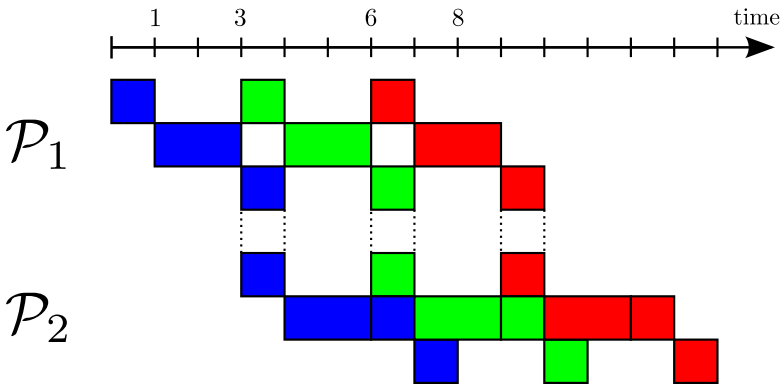

- Period: $T = 3$
- Latency: $L = 8$

# Example



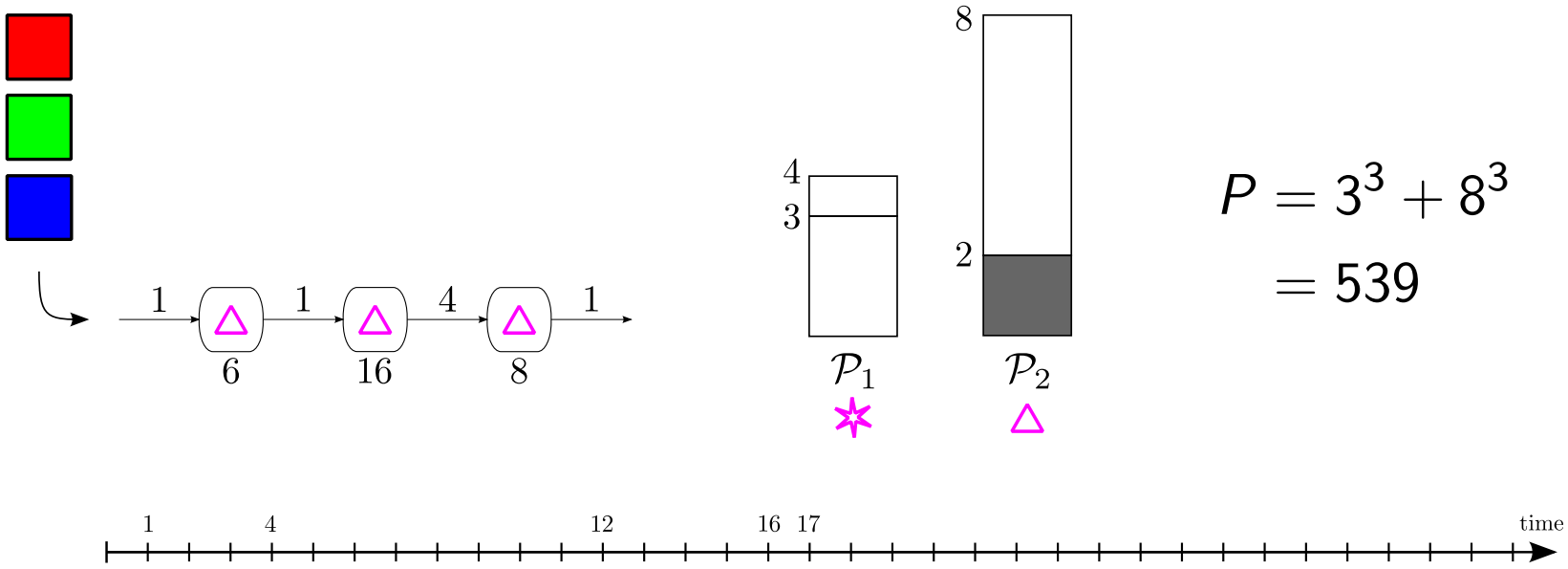




$$P = 3^3 + 8^3$$
$$= 539$$



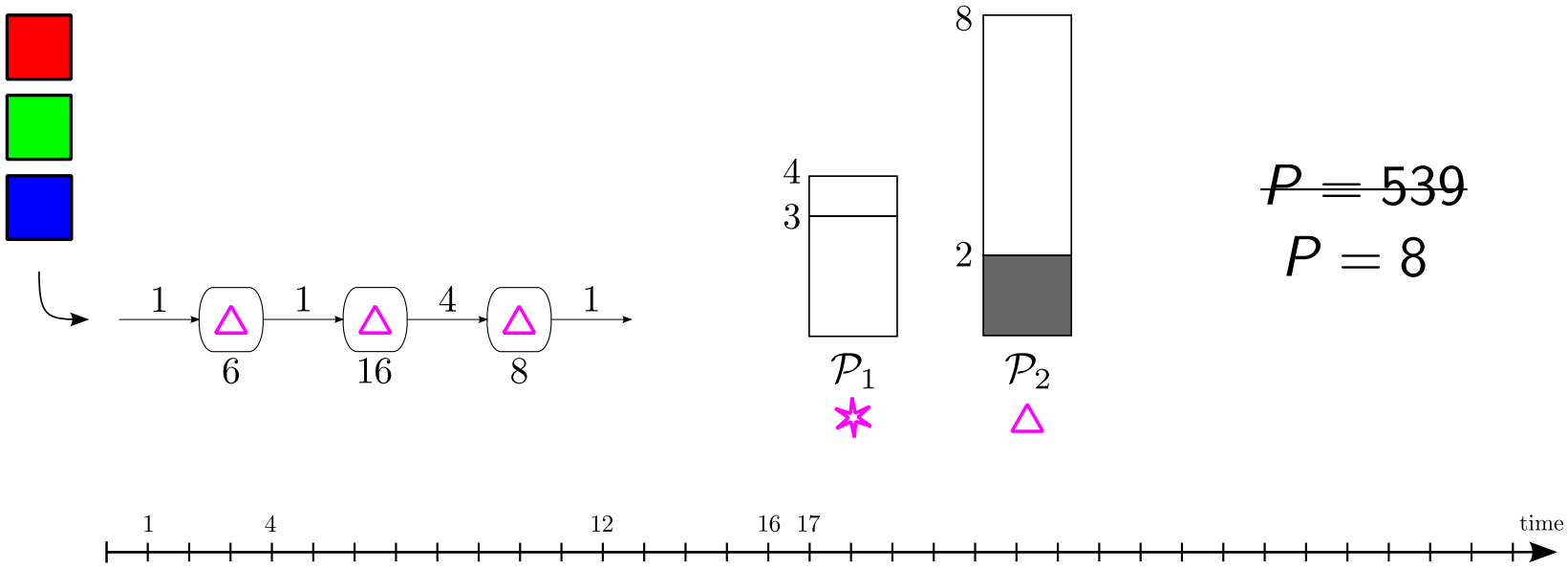

- Period: $T = 3$
- Latency: $L = 8$

# Example



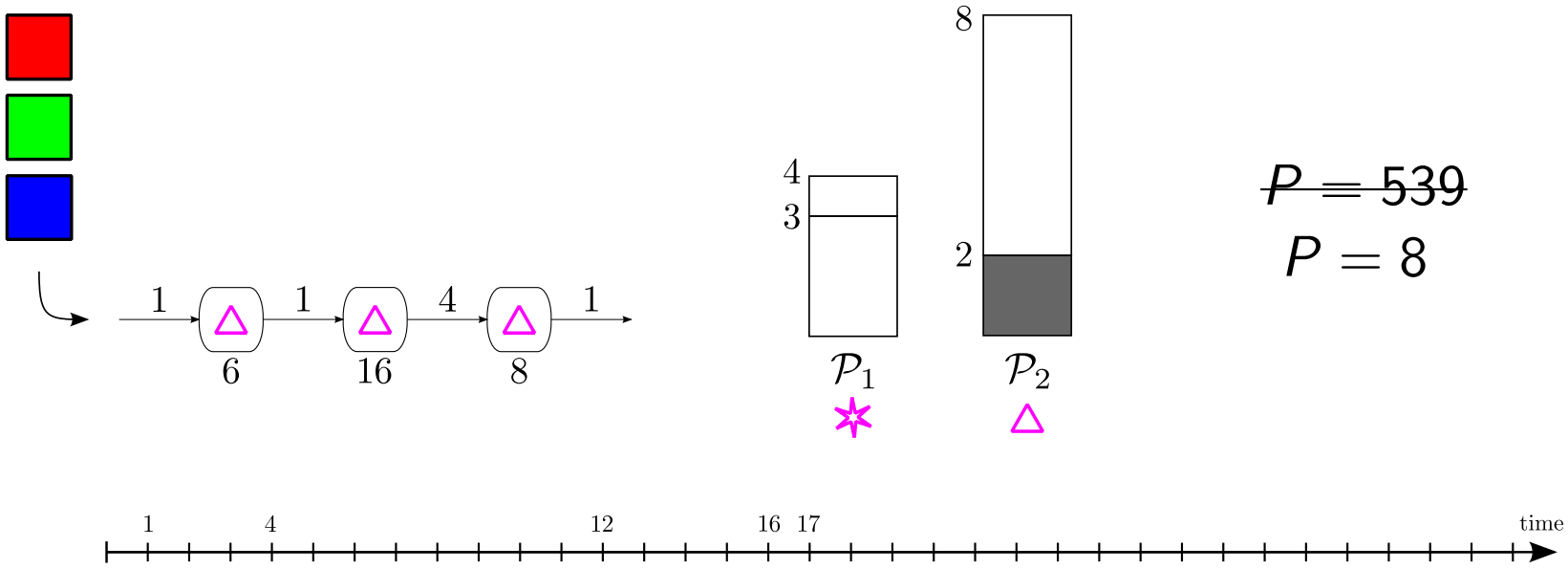

- Period: $T = 3$
- Latency: $L = 8$

# Example



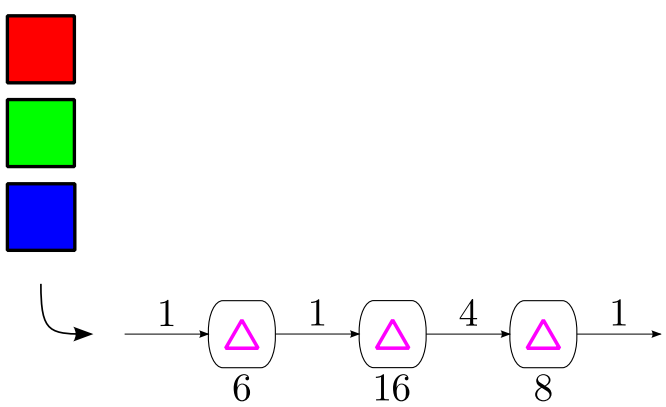

- Period: $T = 3$
- Latency: $L = 8$

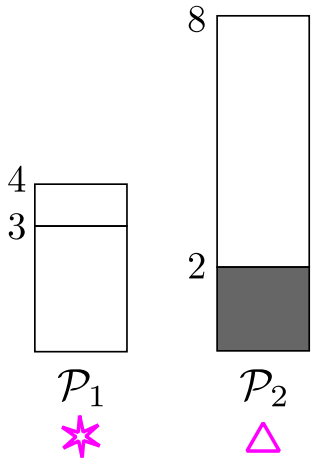
# Example




- Period: ~~$T = 3$~~ $T = 15$
- Latency: $L = 8$

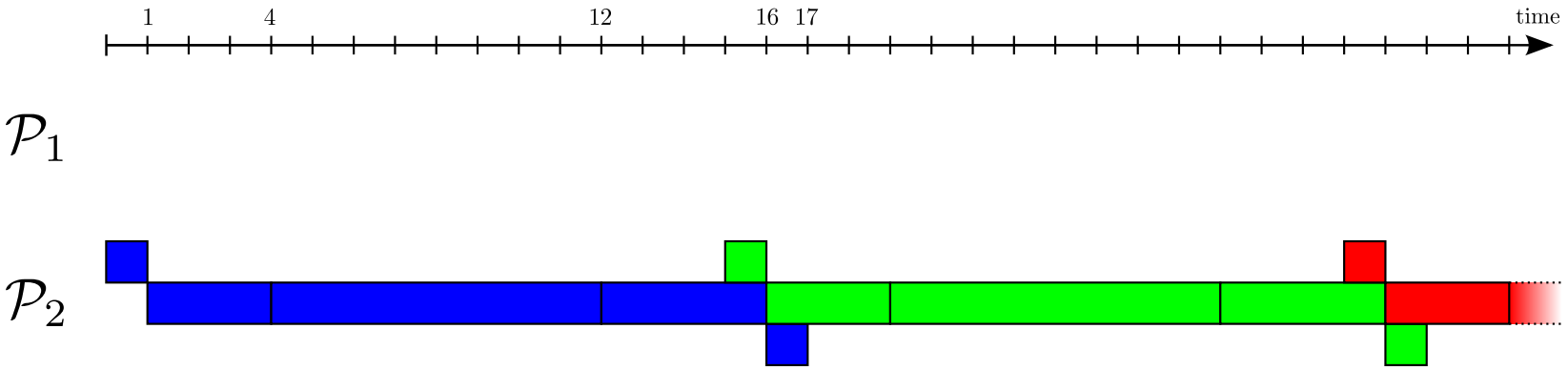
# Example




~~$P = 539$~~

$P = 8$




- Period: ~~$T = 3$~~ $T = 15$
- Latency: ~~$L = 8$~~ $L = 17$

# Multi-criteria optimization

## Performance-oriented objectives

- Minimize period (inverse of throughput)
- Minimize latency (time to process a data set)
- Minimize application failure probability

## Environmental objectives

- Minimize energy consumption
- Minimize platform cost

## Objective function

- Minimize $\alpha.\mathcal{P} + \beta.\mathcal{L} + \gamma.\mathcal{E}$?
- Minimize period, given a response time bound and an energy budget!

# Multi-criteria optimization

## Performance-oriented objectives

- Minimize period (inverse of throughput) $\mathcal{P}$
- Minimize latency (time to process a data set) $\mathcal{L}$
- Minimize application failure probability

## Environmental objectives

- Minimize energy consumption $\mathcal{E}$
- Minimize platform cost

## Objective function

- Minimize $\alpha.\mathcal{P} + \beta.\mathcal{L} + \gamma.\mathcal{E}$?
- Minimize period, given a response time bound and an energy budget!

# Multi-criteria optimization

## Performance-oriented objectives

- Minimize period (inverse of throughput) $\mathcal{P}$
- Minimize latency (time to process a data set) $\mathcal{L}$
- Minimize application failure probability

## Environmental objectives

- Minimize energy consumption $\mathcal{E}$
- Minimize platform cost

## Objective function

- Minimize $\alpha.\mathcal{P} + \beta.\mathcal{L} + \gamma.\mathcal{E}$?
- Minimize period, given a response time bound and an energy budget!

# Research problems

## Single workflow

- Performance (period/latency) vs. environment (energy, cost)
- Robust mappings (replication for performance vs. for reliability

## Several (concurrent) workflows

- Competition for CPU and network resources
- Fairness between applications:
  - max-min throughput
  - min-max stretch (slowdown factor)
- Sensitivity to application/platform parameter changes

# Research problems

## Single workflow

- Performance (period/latency) vs. environment (energy, cost)
- Robust mappings (replication for performance vs. for reliability

## Several (concurrent) workflows

- Competition for CPU and network resources
- Fairness between applications:
  - max-min throughput
  - min-max stretch (slowdown factor)
- Sensitivity to application/platform parameter changes

# Lesson learnt?

Period, latency, stretch, robustness, fairness, cost, energy and combinations
lead to difficult optimization problems

Lot of work for young and talented algorithmicians ☺

**Example:** almost everything yet to be done!

# Lesson learnt?

Period, latency, stretch, robustness, fairness, cost, energy and combinations
lead to difficult optimization problems

Lot of work for young and talented algorithmicians ☺

**Example:** almost everything yet to be done!

# Outline

1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Problem

- Large divisible computational workload
- Assemblage of $p$ identical computers
- Unrecoverable interruptions
- A-priori knowledge of risk (failure probability)

Goal: maximize expected amount of work done

# Related work

- Landmark paper by Bhatt, Chung, Leighton & Rosenberg on cycle stealing
- Hardware failures

☺ **Fault tolerant computing (hence scheduling) becomes unavoidable**

☹ **Well, same story told since very long!**

# Related work

- Landmark paper by Bhatt, Chung, Leighton & Rosenberg on cycle stealing
- Hardware failures

☺ **Fault tolerant computing (hence scheduling) becomes unavoidable**

☹ **Well, same story told since very long!**

# Related work

- Landmark paper by Bhatt, Chung, Leighton & Rosenberg on cycle stealing
- Hardware failures

☺ **Fault tolerant computing (hence scheduling) becomes unavoidable**

☹ **Well, same story told since very long!**

# Cycle-stealing scenario (1/2)

- Execute 4 jobs $\boxed{A}$ $\boxed{B}$ $\boxed{C}$ $\boxed{D}$ during week-end
- Replicate them on 3 machines $P_1$, $P_2$ and $P_3$
- Risk increases with time
- Machines reclaimed at 8am on Monday with probability 1

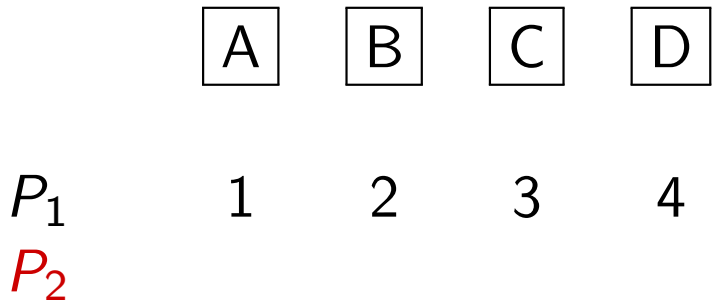
# Cycle-stealing scenario (1/2)

- Execute 4 jobs $\boxed{A}$ $\boxed{B}$ $\boxed{C}$ $\boxed{D}$ during week-end
- Replicate them on 3 machines $P_1$, $P_2$ and $P_3$
- Risk increases **linearly** with time
- Machines reclaimed at 8am on Monday with probability 1

# Cycle-stealing scenario (2/2)

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |

# Cycle-stealing scenario (2/2)



# Cycle-stealing scenario (2/2)

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 4 | 3 | 2 | 1 |

# Cycle-stealing scenario (2/2)

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 4 | 3 | 2 | 1 |
| $P_3$ | | | | |

# Cycle-stealing scenario (2/2)

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 4 | 3 | 2 | 1 |
| $P_3$ | 4 | 3 | 2 | 1 |

# Cycle-stealing scenario (2/2)

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 4 | 3 | 2 | 1 |
| $P_3$ | 4 | 3 | 2 | 1 |

| | A | B | C | D |
|---|---|---|---|---|
| $P_1$ | 1 | 2 | 3 | 4 |
| $P_2$ | 4 | 3 | 1 | 2 |
| $P_3$ | 3 | 2 | 4 | 1 |

# Orchestrating

| Chunk | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 1 | 6 | 9 | 12 | 2 | 5 | 8 | 11 | 3 | 4 | 7 | 10 |
| $P_2$ | 12 | 1 | 6 | 9 | 11 | 2 | 5 | 8 | 10 | 3 | 4 | 7 |
| $P_3$ | 9 | 12 | 1 | 6 | 8 | 11 | 2 | 5 | 7 | 10 | 3 | 4 |
| $P_4$ | 6 | 9 | 12 | 1 | 5 | 8 | 11 | 2 | 4 | 7 | 10 | 3 |

Time-steps for execution of n = 12 chunks with g = 4 processors

# Group schedules

| Chunk | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 1 | 6 | 9 | 12 | 2 | 5 | 8 | 11 | 3 | 4 | 7 | 10 |
| $P_2$ | 12 | 1 | 6 | 9 | 11 | 2 | 5 | 8 | 10 | 3 | 4 | 7 |
| $P_3$ | 9 | 12 | 1 | 6 | 8 | 11 | 2 | 5 | 7 | 10 | 3 | 4 |
| $P_4$ | 6 | 9 | 12 | 1 | 5 | 8 | 11 | 2 | 4 | 7 | 10 | 3 |

| Group 1 chunks 1-4 | Group 2 chunks 5-8 | Group 3 chunks 9-12 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

Time-steps for group execution

# Group schedules

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

# Group schedules

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

All four executions fail with probability proportional to $1 \times 6 \times 9 \times 12$

# Group schedules

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

All four executions fail with probability proportional to $2 \times 5 \times 8 \times 11$

# Group schedules

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

All four executions fail with probability proportional to $3 \times 4 \times 7 \times 10$

# Group schedules

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

All four executions fail with probability proportional to $3 \times 4 \times 7 \times 10$

$$K = \sum_{j=1}^{\frac{n}{g}} \prod_{i=1}^{g} G_{i,j} = 1.6.9.12 + 2.5.8.11 + 3.4.7.10$$

**Better performance for small K**

# Scheduling objective

$$E(\mathsf{sl}, \mathsf{n}) = \mathsf{sl}\left(1 - \frac{\mathsf{g}}{\mathsf{n}}\left(\frac{\mathsf{sl}\kappa}{\mathsf{n}}\right)^{\mathsf{g}} \sum_{j=1}^{\frac{\mathsf{n}}{\mathsf{g}}} \prod_{i=1}^{\mathsf{g}} G_{i,j}\right)$$

**Problem**

Minimize

$$\mathsf{K} = \sum_{j=1}^{\frac{\mathsf{n}}{\mathsf{g}}} \prod_{i=1}^{\mathsf{g}} G_{i,j}$$

where entries of $G$ are a permutation of $[1..n]$

**Bound**

$$\mathsf{K}_{\mathsf{min}} = \left\lceil \frac{\mathsf{n}}{\mathsf{g}} (\mathsf{n}!)^{\frac{\mathsf{g}}{\mathsf{n}}} \right\rceil$$

# Heuristics (1/3)

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 10 | 11 | 12 |

**(a) Cyclic:** K = 3104

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

**(b) Reverse:** K = 2368

# Heuristics (2/3)

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 9 | 8 | 7 |
| 12 | 11 | 10 |

**(c) Mirror:** K = 2572

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 6 | 5 | 4 |
| 7 | 8 | 9 |
| 12 | 11 | 10 |

**(d) Snake:** K = 2464

# Heuristics (3/3)

| Group 1 | Group 2 | Group 3 |
|---|---|---|
| 1 | 2 | 3 |
| 8 | 6 | 4 |
| 9 | 7 | 5 |
| 10 | 11 | 12 |

**(e) Worm:** $K = 2364$

| Step 1 | 1 | 2 | 3 |
|---|---|---|---|
| CCP | 1 | 2 | 3 |
| Step 2 | 6 | 5 | 4 |
| CCP | 6 | 10 | 12 |
| Step 3 | 9 | 8 | 7 |
| CCP | 54 | 80 | 84 |
| Step 4 | 12 | 11 | 10 |

**(f) Greedy:** $K = 2368 \geq K_{min} = 2348$

# Comparing group schedules for n = 9 and g = 3

| | | |
|---|---|---|
| 1 2 3 | 1 2 3 | 1 2 3 |
| 4 5 6 | 6 5 4 | 6 5 4 |
| 7 8 9 | 7 8 9 | 9 8 7 |
| $K_{cyclic} = 270$ | $K_{snake} = 230$ | $K_{reverse} = K_{greedy} = 218$ |

| | |
|---|---|
| 1 2 3 | 1 2 3 |
| 8 6 4 | 8 5 4 |
| 9 7 5 | 9 7 6 |
| $K_{worm} = 216$ | $K_{optimal} = K_{min} = 214$ |

# Comparing group schedules for n = 20 and g = 4

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |

$K_{cyclic} = 34104$

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 15 | 14 | 13 | 12 | 11 |
| 20 | 19 | 18 | 17 | 16 |

$K_{mirror} = 27284$

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 10 | 9 | 8 | 7 | 6 |
| 15 | 14 | 13 | 12 | 11 |
| 20 | 19 | 18 | 17 | 16 |

$K_{reverse} = 24396$

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 10 | 9 | 8 | 7 | 6 |
| 11 | 12 | 13 | 14 | 15 |
| 20 | 19 | 18 | 17 | 16 |

$K_{snake} = 25784$

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 14 | 12 | 10 | 8 | 6 |
| 15 | 13 | 11 | 9 | 7 |
| 16 | 17 | 18 | 19 | 20 |

$K_{worm} = 24276$

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 10 | 9 | 8 | 7 | 6 |
| 15 | 14 | 13 | 12 | 11 |
| 20 | 19 | 18 | 16 | 17 |

$K_{greedy} = 24390$

$K_{min} = 23780$

# A nice little algorithmic challenge

| | Group 1 | Group 2 | Group 3 | ... | Group n |
|---|---|---|---|---|---|
| $P_1$ | x | x | x | ... | x |
| $P_2$ | x | x | x | ... | x |
| $P_3$ | x | x | x | ... | x |
| $P_4$ | x | x | x | ... | x |
| ... | ... | ... | ... | ... | ... |
| $P_p$ | x | x | x | ... | x |

Fill up matrix with a **permutation of** $[1..n \times p]$ minimizing the **sum of column products**

# Outline


1. Parallel algorithms
   - Independent tasks
   - A simple tiling problem
   - Matrix product (ScaLAPACK)
2. Scheduling
   - Background: scheduling DAGs
   - Packet routing
   - Master-worker tasking
3. Advanced examples
   - Pipeline workflows
   - Coping with failures
4. Conclusion

# Tools for the road

- Forget absolute makespan minimization
- Resource selection mandatory
- Divisible load (fractional tasks)
- Single application: period / latency / energy / robustness
- Several applications: max-min fairness, MAX stretch
- Linear programming: absolute bound to assess heuristics

# Scheduling for large-scale platforms

- If platform is well identified and relatively stable, try to:
  (i) accurately model hierarchical structure
  (ii) design efficient/robust/energy-aware scheduling algorithms

- If platform is not stable enough, or if it evolves too fast, dynamic schedulers are the only option

- Otherwise, grab any opportunity to

  **inject static knowledge into dynamic schedulers**

  ☹ Is this opportunity a niche?
  ☺ Does it encompass a wide range of applications?