"The computer programmer is a creator of universes for which he alone is the lawgiver … Universes of virtually unlimited complexity can be created in the form of computer programs … They compliantly obey their laws and vividly exhibit their obedient behavior. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or a field of battle and to command such unswervingly dutiful actors of troops."

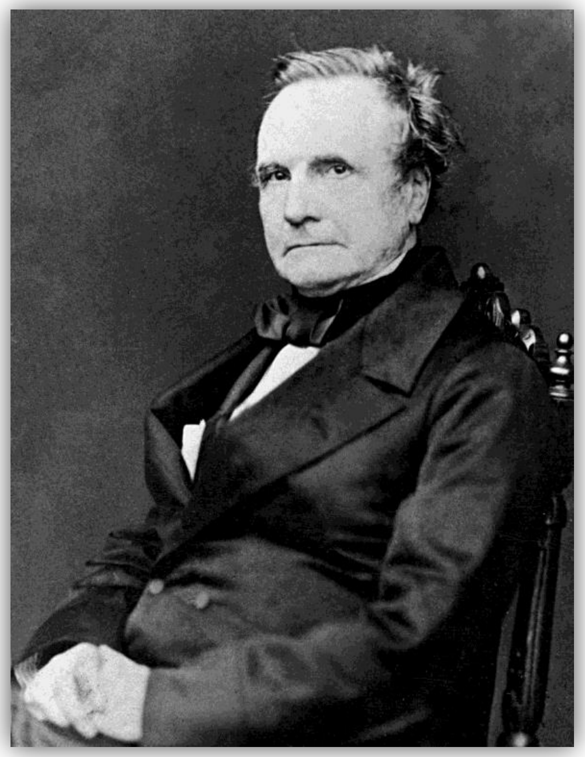
-- J. Weizenbaum, Computer Power and Human Reason: From Judgment to Calculation

Concepts and Strategies in Parallel Programming

A Brief Introduction

“As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise — by what course of calculation can these results be arrived at by the machine in the shortest time?”

~CHARLES BABBAGE (1864)



Part I – Basic Concepts

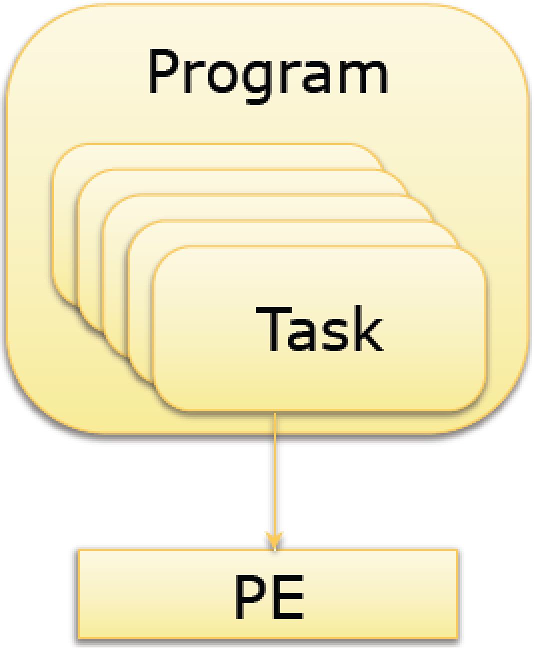
So what does it mean to do Parallel Computing/Programming?

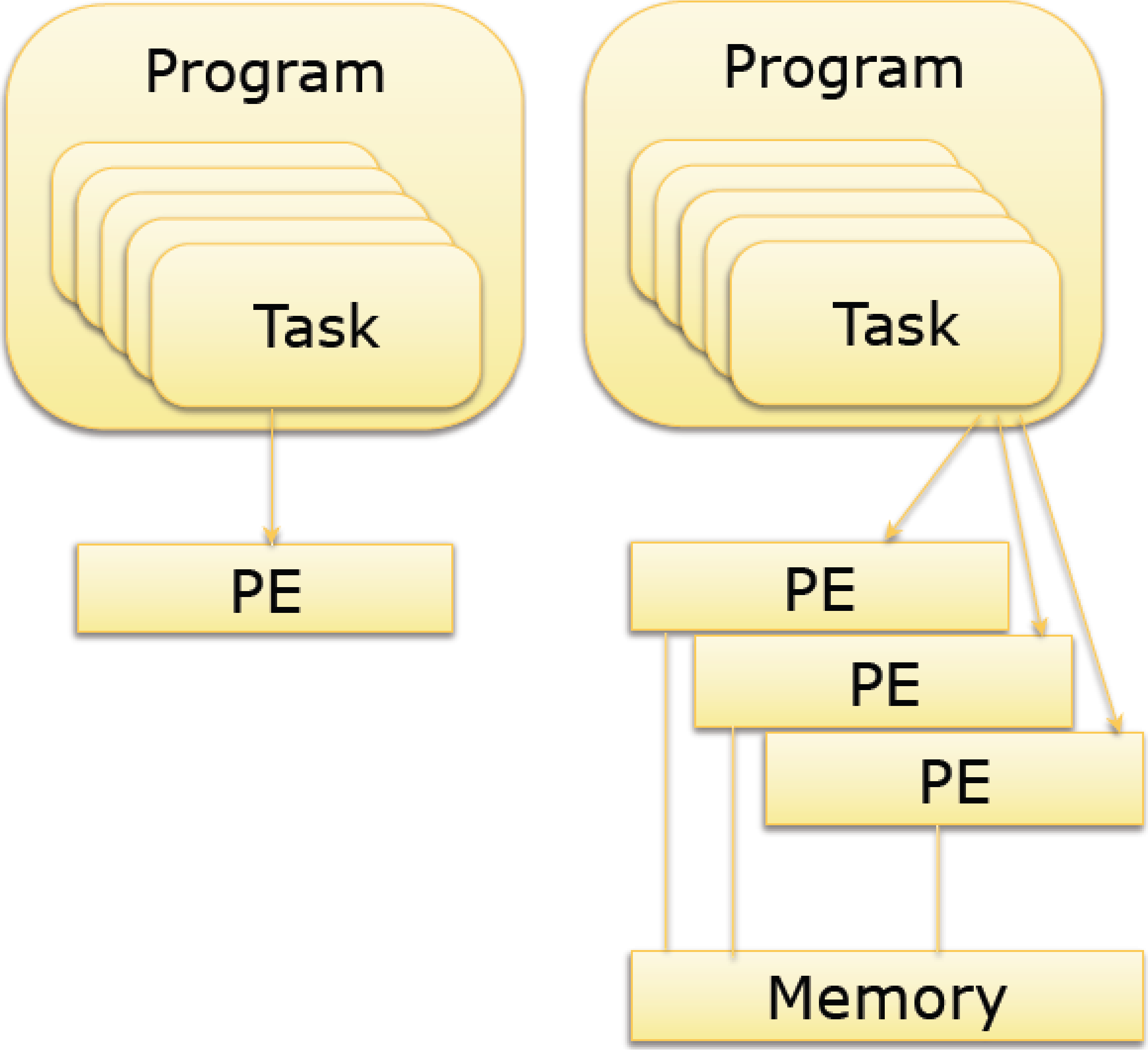
"I decided long ago to stick to what I know best. Other people understand parallel machines much better than I do; programmers should listen to them, not me, for guidance on how to deal with simultaneity."

--**Donald Knuth** (Stanford)

"Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end."

--**Herb Sutter**
(Chair of the ISO C++ standards committee, Microsoft)


Program
Task
PE



Program

Task

PE

Program

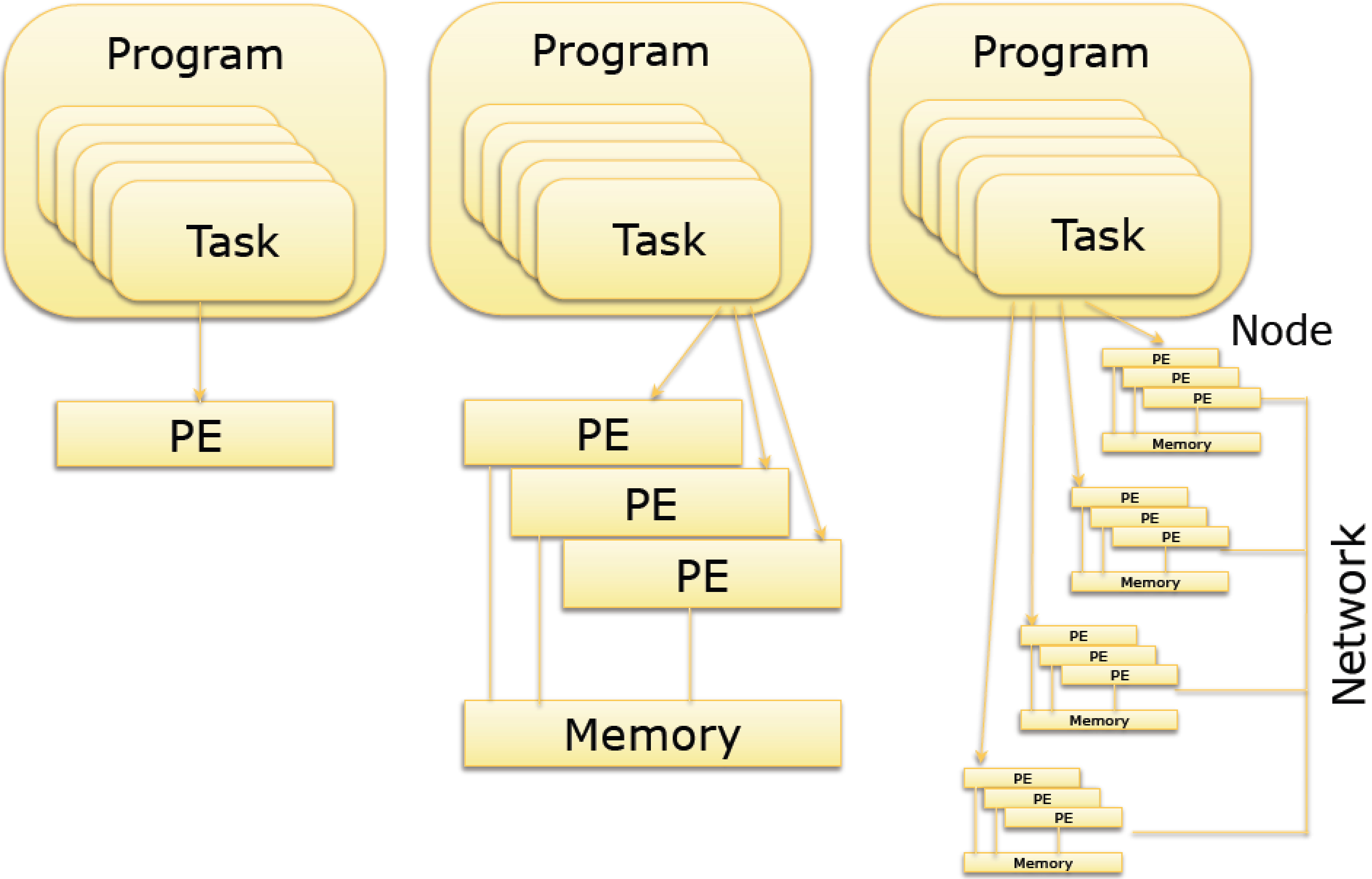
Task

PE

PE

PE

Memory



Program

Task

PE

Program

Task

PE

PE

PE

Memory

Program

Task

Node

PE

PE

PE

Memory

PE

PE

PE

Memory

PE

PE

PE

Memory

PE

PE

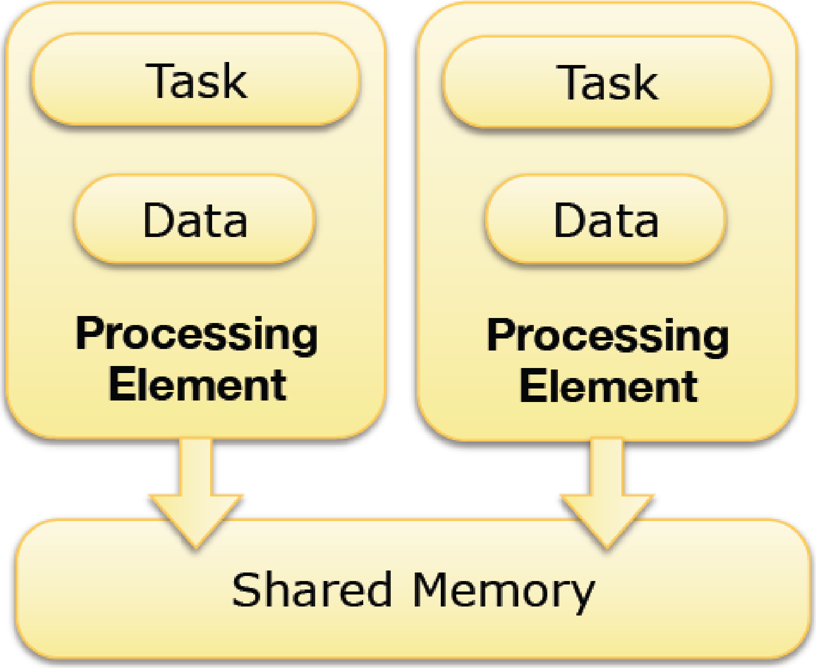
PE

Memory

Network

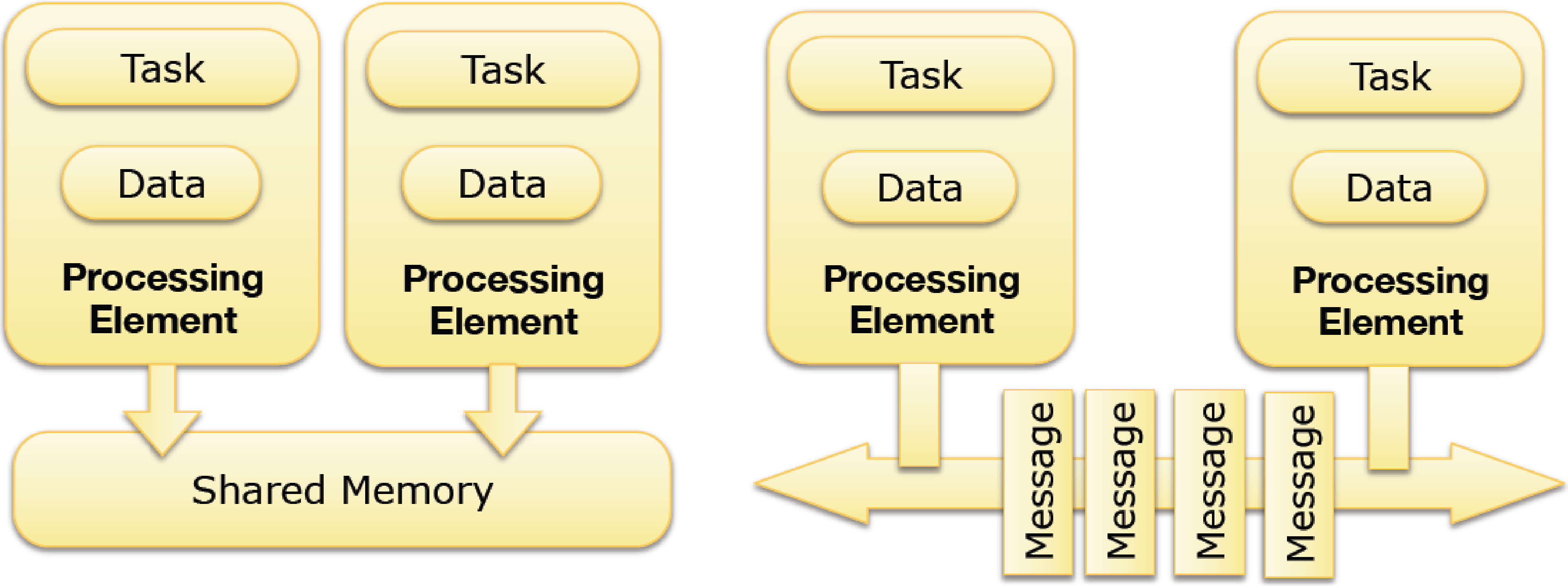
Shared-Memory vs. Shared-Nothing

- **Pfister:** *Shared-Memory* vs. *Distributed-Memory*
- **Foster:** *Multi-Processor* vs. *Multi-Computer*
- **Tennenbaum:** *Shared-Memory* vs. *Private-Memory*



Shared-Memory vs. Shared-Nothing

- **Pfister:** *Shared-Memory* vs. *Distributed-Memory*
- **Foster:** *Multi-Processor* vs. *Multi-Computer*
- **Tennenbaum:** *Shared-Memory* vs. *Private-Memory*



The Essence of Parallel Programming

- *So essentially, what we try to do is,*
  - – Look at a serial program and identify the parts that are the most compute intensive → Computational *Hotspots*
    - The *80/20* rule … mehh!

The Essence of Parallel Programming

- *So essentially, what we try to do is,*
  - Look at a serial program and identify the parts that are the most compute intensive → Computational *Hotspots*
    - The *80/20* rule … mehh!
  - Extract *parallelism* from Hotspots

The Essence of Parallel Programming

- *So essentially, what we try to do is,*
  - Look at a serial program and identify the parts that are the most compute intensive → Computational *Hotspots*
    - The *80/20* rule … mehh!
  - Extract *parallelism* from Hotspots
  - Design parallelization strategy

The Essence of Parallel Programming

- *So essentially, what we try to do is,*
  - Look at a serial program and identify the parts that are the most compute intensive → Computational *Hotspots*
    - The *80/20* rule … mehh!
  - Extract *parallelism* from Hotspots
  - Design parallelization strategy
  - Make it possible for the code to exploit the available parallel processing resources

Concurrency vs. Parallelism

- Concurrent
  - “A system is said to be *concurrent* if it can support two or more actions *in progress* at the same time.”

Concurrency vs. Parallelism

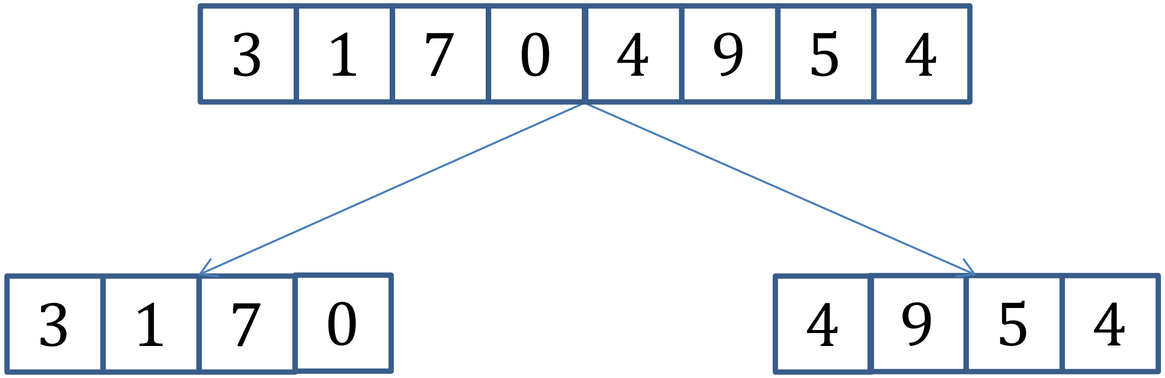
- Concurrent
  - "A system is said to be *concurrent* if it can support two or more actions *in progress* at the same time."
- Parallel
  - "A system is said to be *parallel* if it can support two or more actions *executing simultaneously*."

Reference: Breshears, Clay. *The art of concurrency: A thread monkey's guide to writing parallel applications*. O'Reilly Media, Inc., 2009.

Geometric Decomposition

3	1	7	0	4	9	5	4

Geometric Decomposition

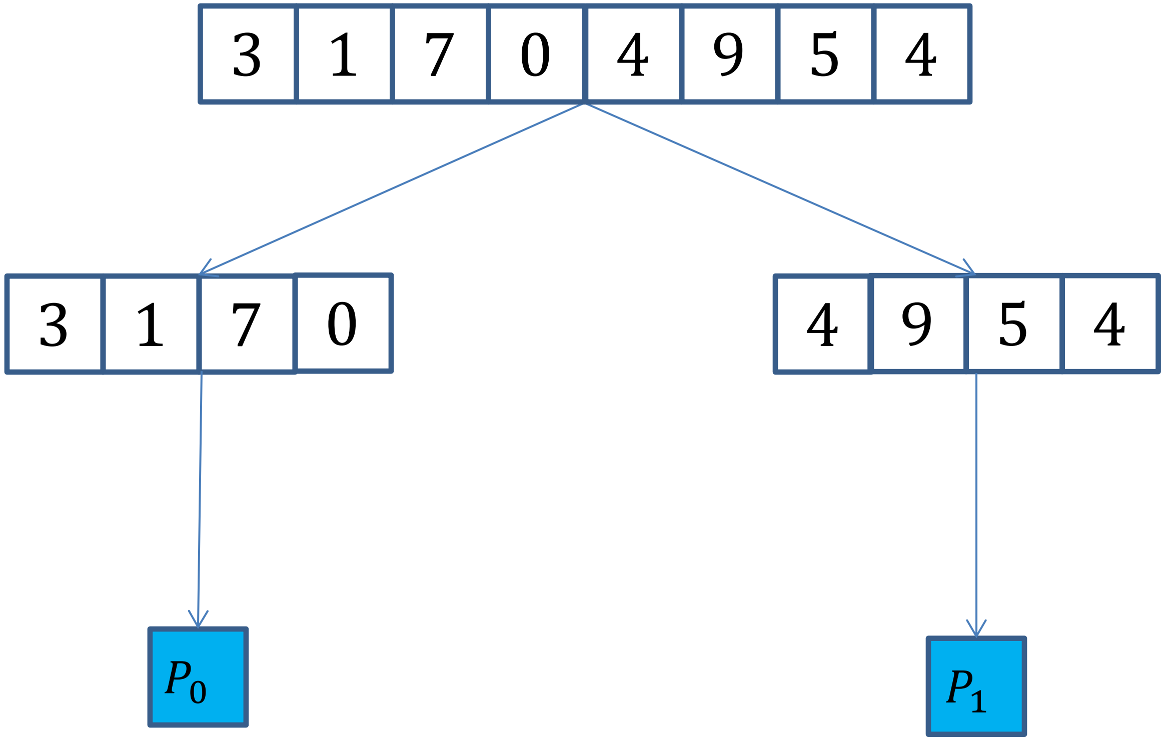


3 1 7 0 4 9 5 4

3 1 7 0

4 9 5 4

Geometric Decomposition



3	1	7	0	4	9	5	4

3	1	7	0

4	9	5	4

P_0

P_1

Map Pattern – Independent Loop Iterations

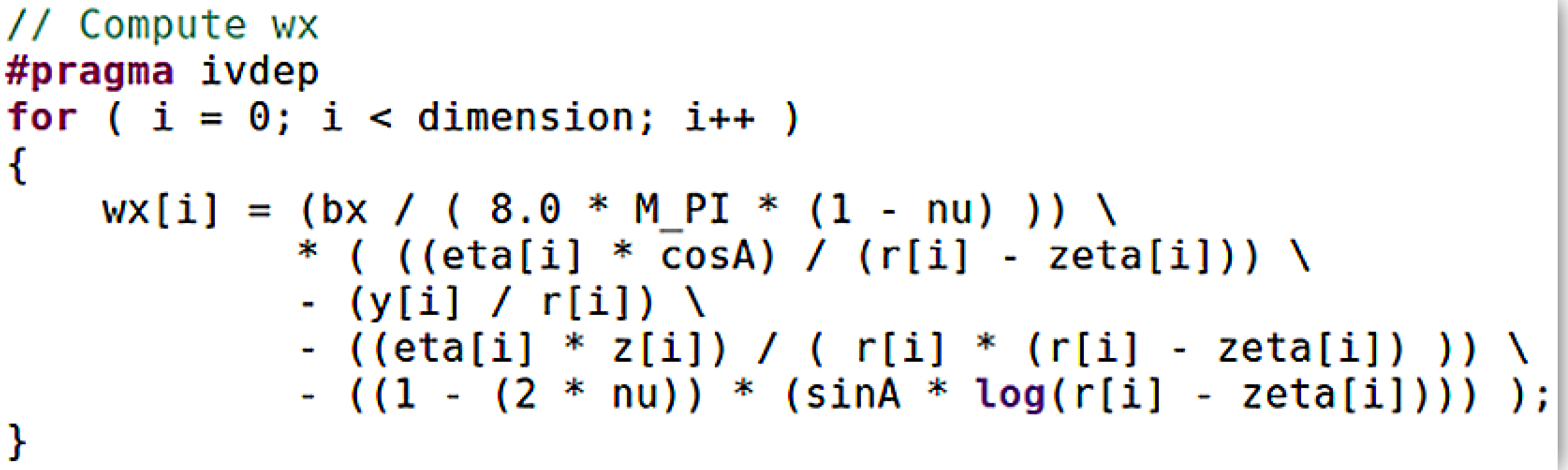
$$w = \frac{b}{8\pi(1-\sigma)} \\ \times \left(\frac{\eta \cos\alpha}{(r-\zeta)} - \frac{y}{r} - \frac{\eta z}{r(r-\zeta)} - (1-2\sigma) \sin\alpha \log(r-\zeta) \right)$$

Elizabeth H. Yoffe (1960): The angular dislocation, *Philosophical Magazine*, 5:50, 161-175

Map Pattern – Independent Loop Iterations

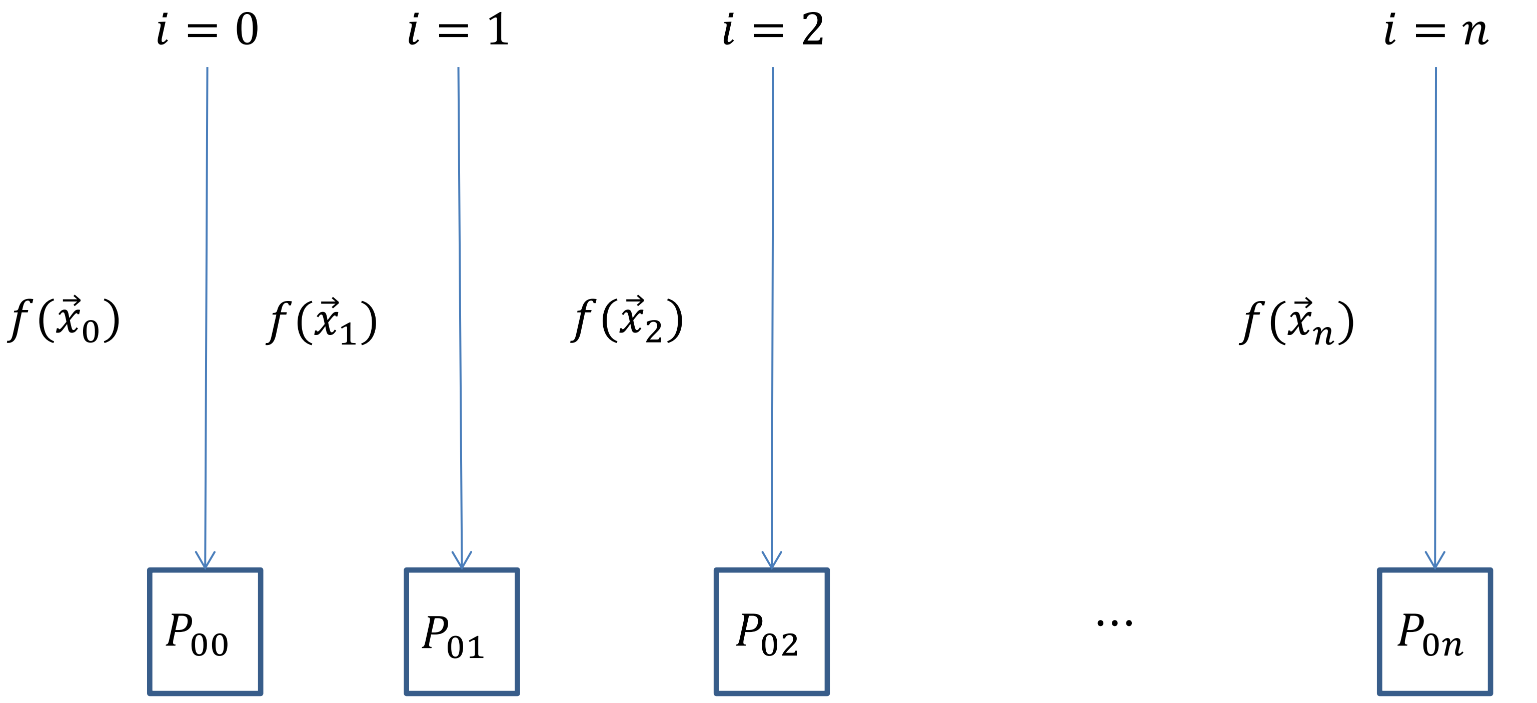
$$w = \frac{b}{8\pi(1-\sigma)} \times \left(\frac{\eta \cos\alpha}{(r-\zeta)} - \frac{y}{r} - \frac{\eta z}{r(r-\zeta)} - (1-2\sigma)\sin\alpha \log(r-\zeta) \right)$$

Elizabeth H. Yoffe (1960): The angular dislocation, *Philosophical Magazine*, 5:50, 161-175



```
// Compute wx
#pragma ivdep
for ( i = 0; i < dimension; i++ )
{
    wx[i] = (bx / ( 8.0 * M_PI * (1 - nu) )) \
            * ( ((eta[i] * cosA) / (r[i] - zeta[i])) \
            - (y[i] / r[i]) \
            - ((eta[i] * z[i]) / ( r[i] * (r[i] - zeta[i]) )) \
            - ((1 - (2 * nu)) * (sinA * log(r[i] - zeta[i]))) );
}
```

Mapping Independent Loop Iterations to Processing Units



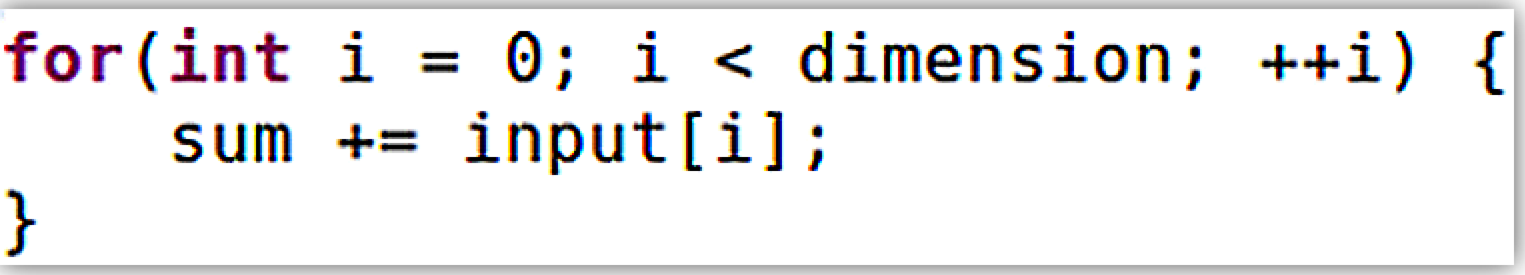
$i = 0$ $i = 1$ $i = 2$ $i = n$

$f(\vec{x}_0)$ $f(\vec{x}_1)$ $f(\vec{x}_2)$ $f(\vec{x}_n)$

P_{00} P_{01} P_{02} $\cdots$ P_{0n}

Reduce Pattern – Data Dependency in a Loop

- *Reduction*
  - Fuse all elements in an array into one element, using an associative (and commutative) operation, e.g., sum
- Serial +Reduction

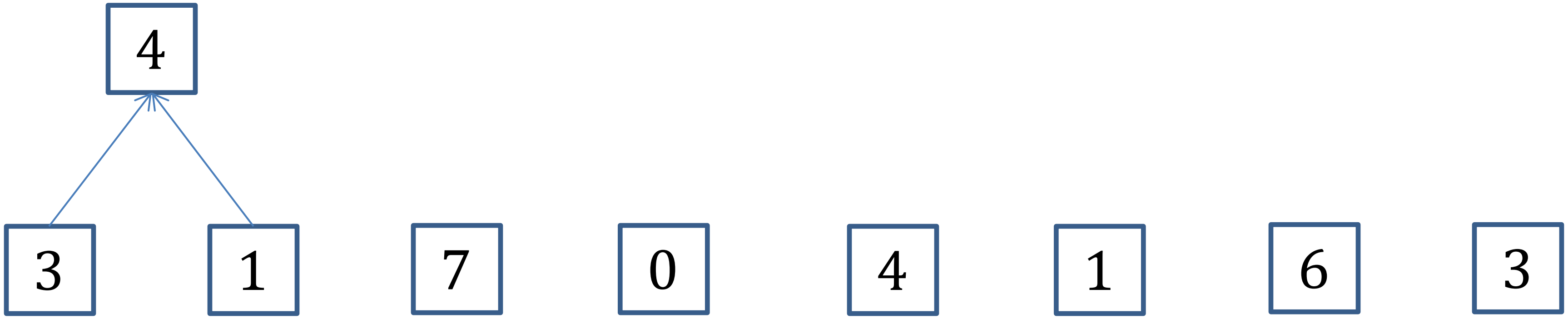


```
for(int i = 0; i < dimension; ++i) {
    sum += input[i];
}
```

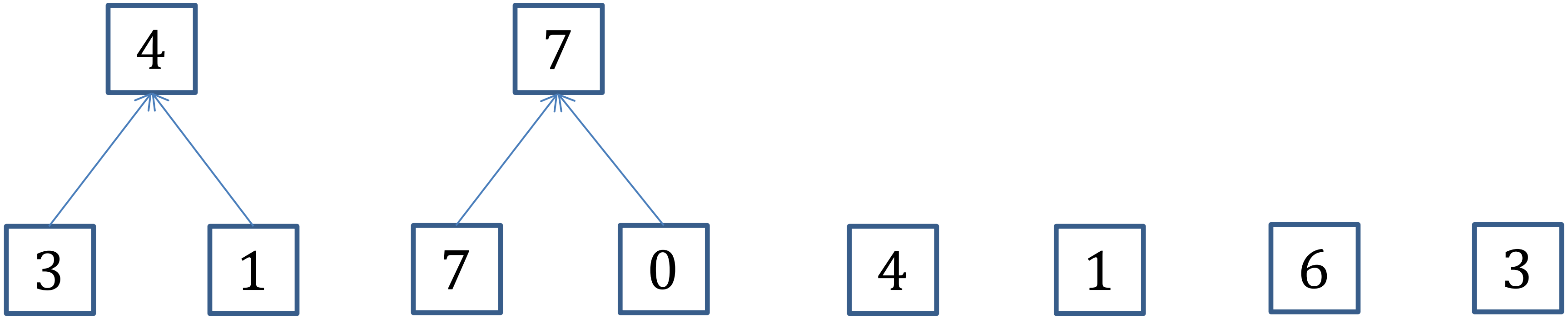
Parallelization despite Dependence – Parallel Reduction

3 1 7 0 4 1 6 3

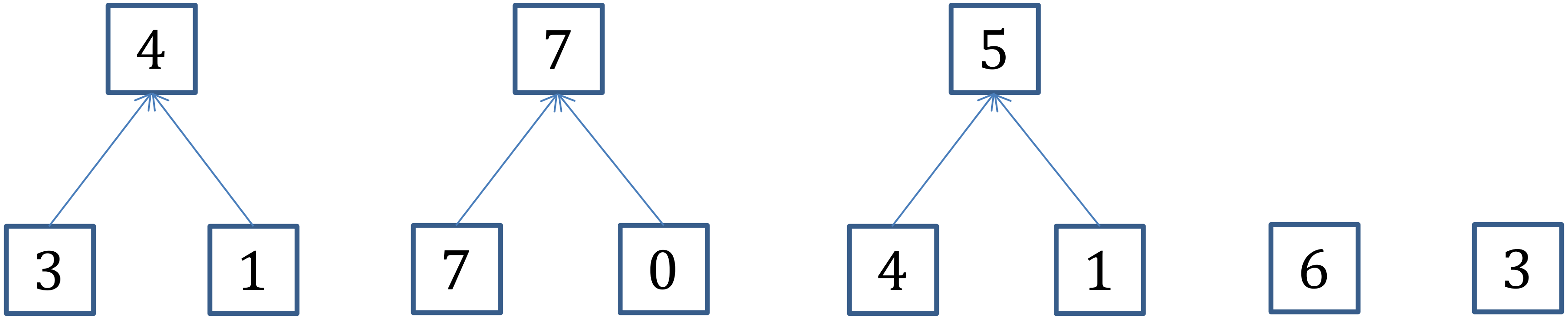
Parallelization despite Dependence – Parallel Reduction



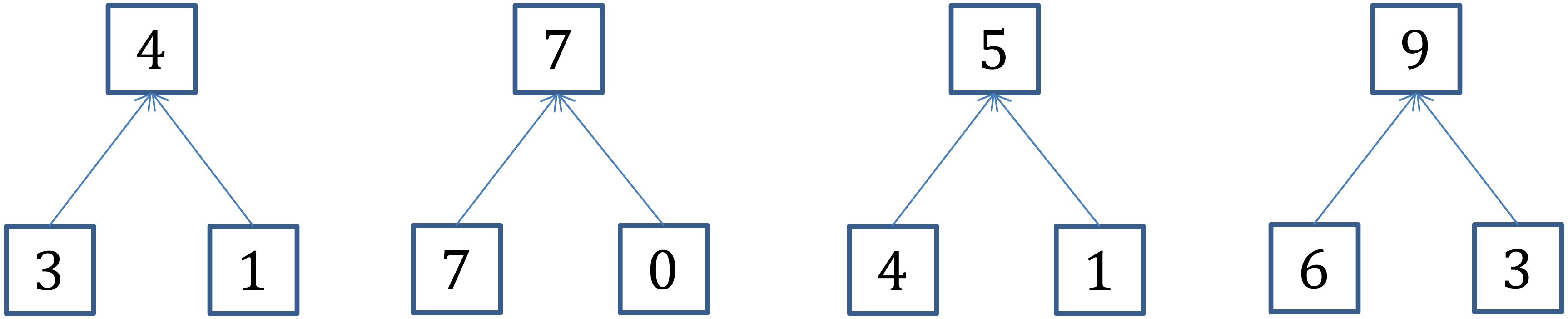
Parallelization despite Dependence – Parallel Reduction



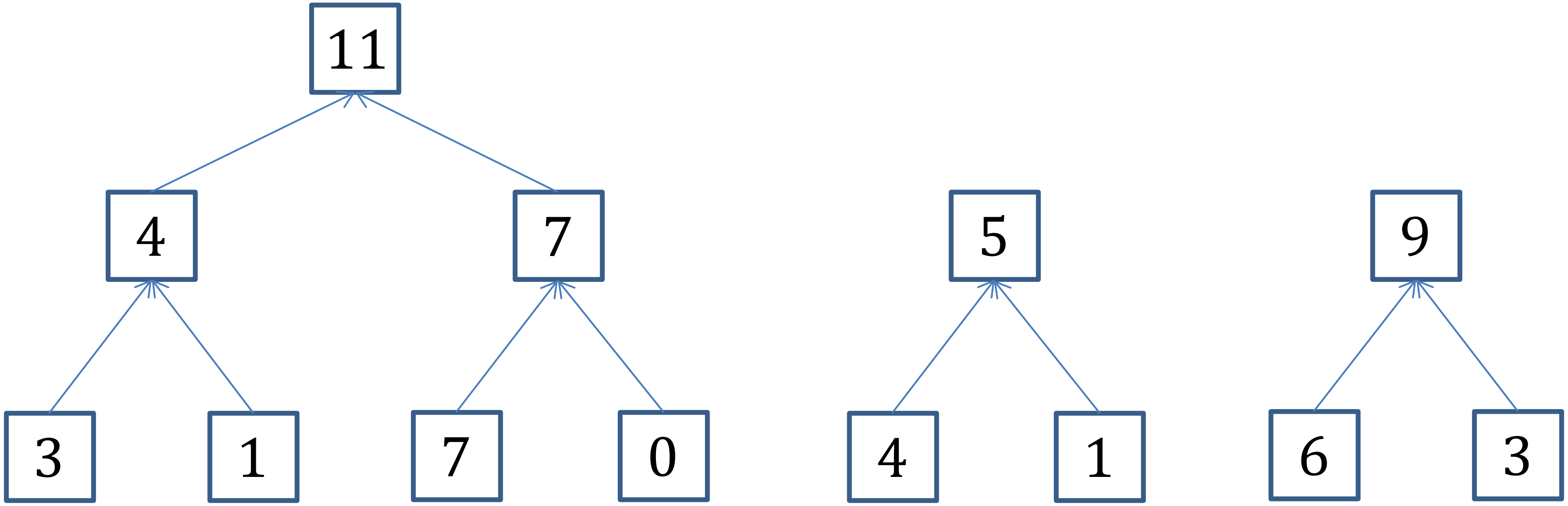
Parallelization despite Dependence – Parallel Reduction



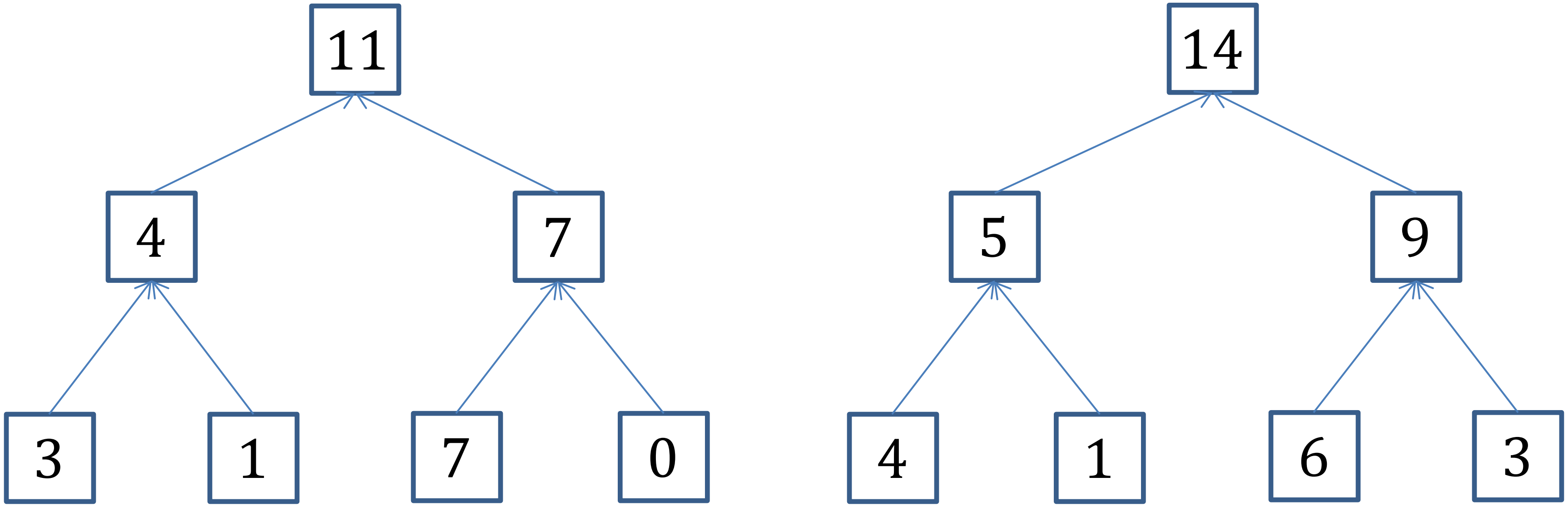
Parallelization despite Dependence – Parallel Reduction



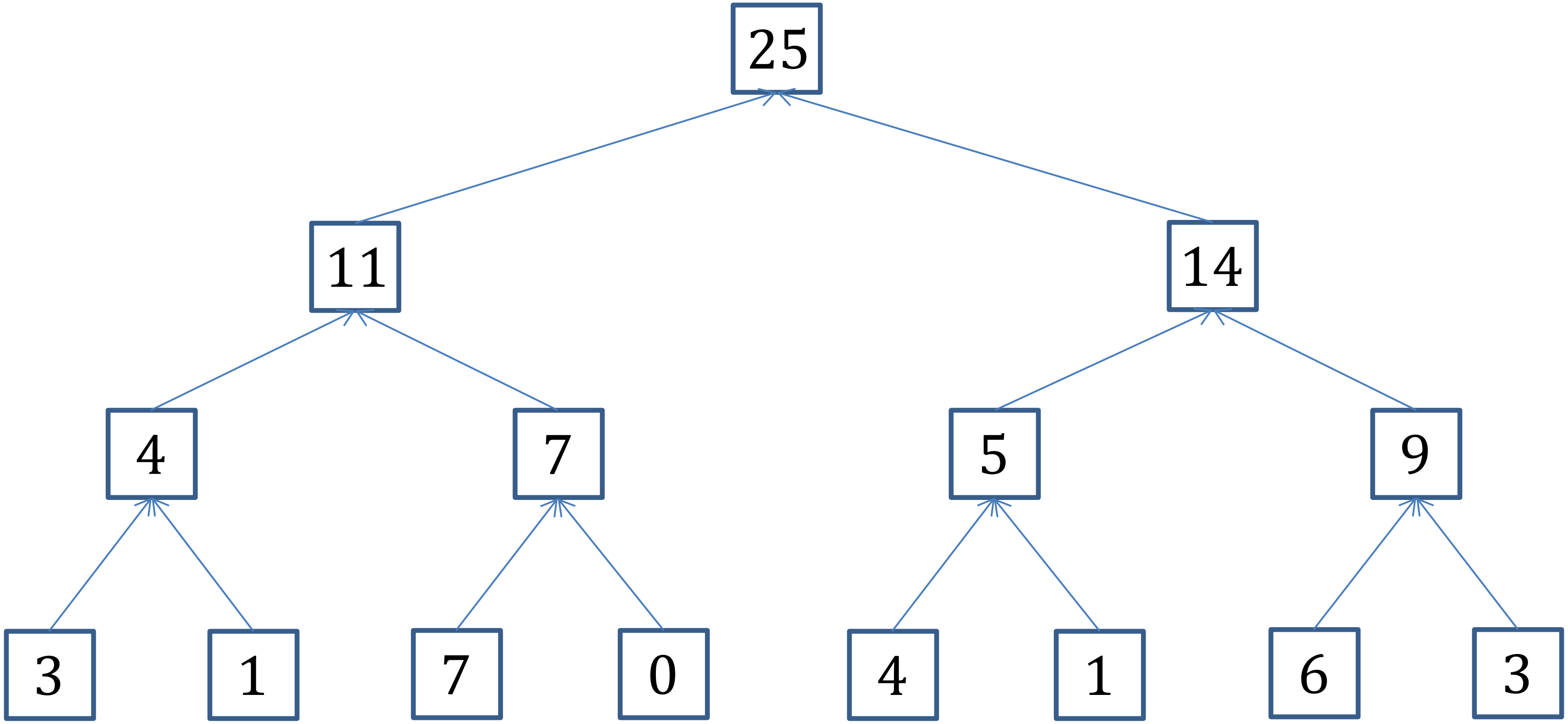
Parallelization despite Dependence – Parallel Reduction



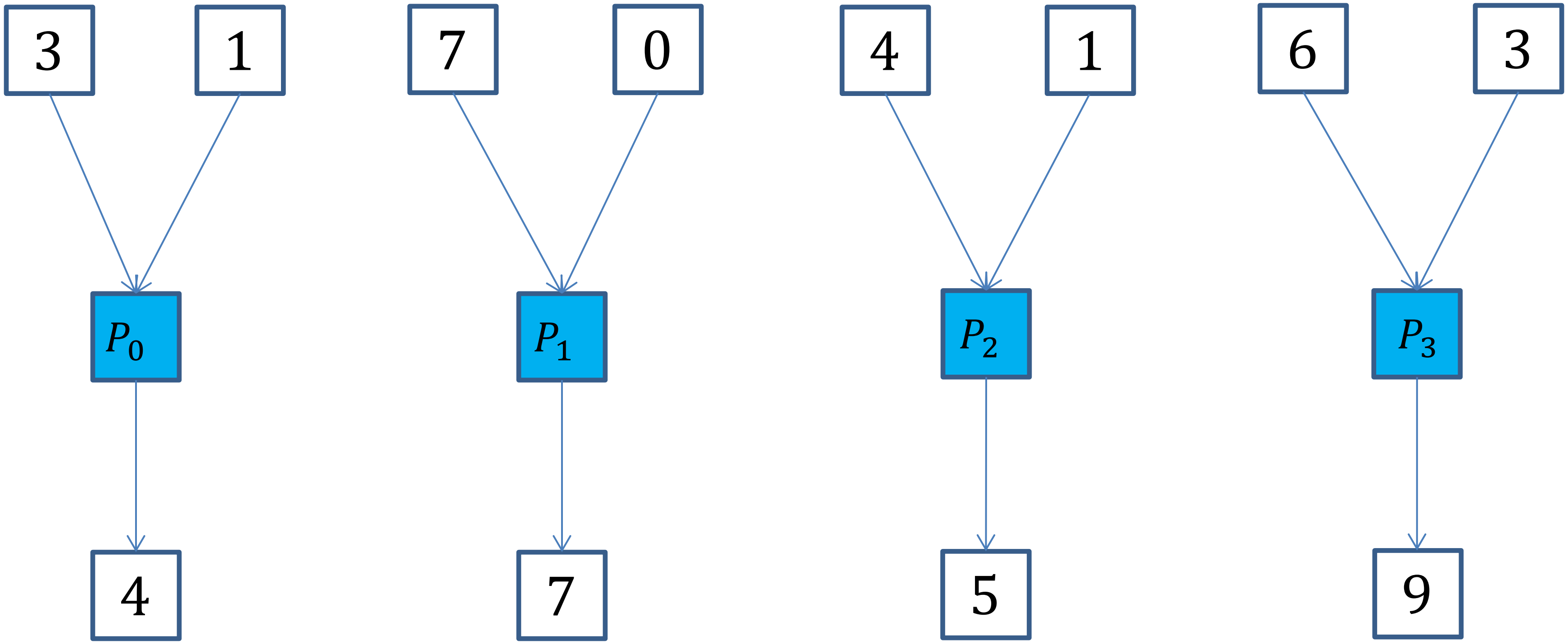
Parallelization despite Dependence – Parallel Reduction



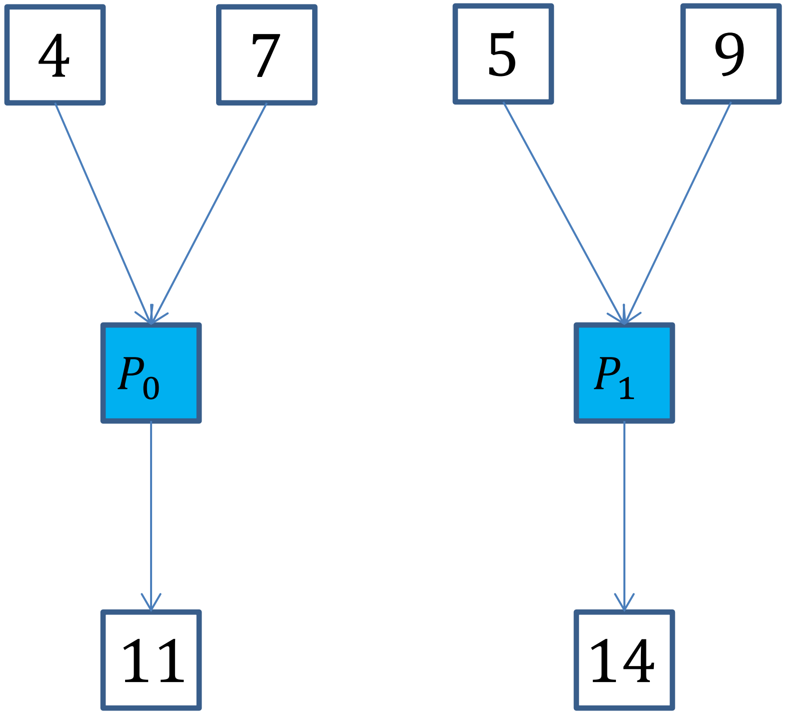
Parallelization despite Dependence – Parallel Reduction



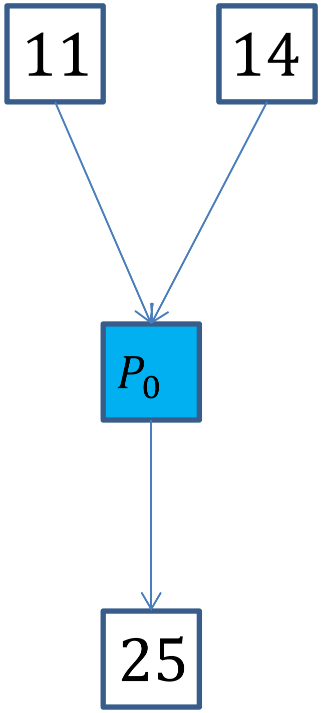
Mapping Tree Reduction to Processing Units



Mapping Tree Reduction to Processing Units



Mapping Tree Reduction to Processing Units



The Stencil Pattern – Neighborhoods

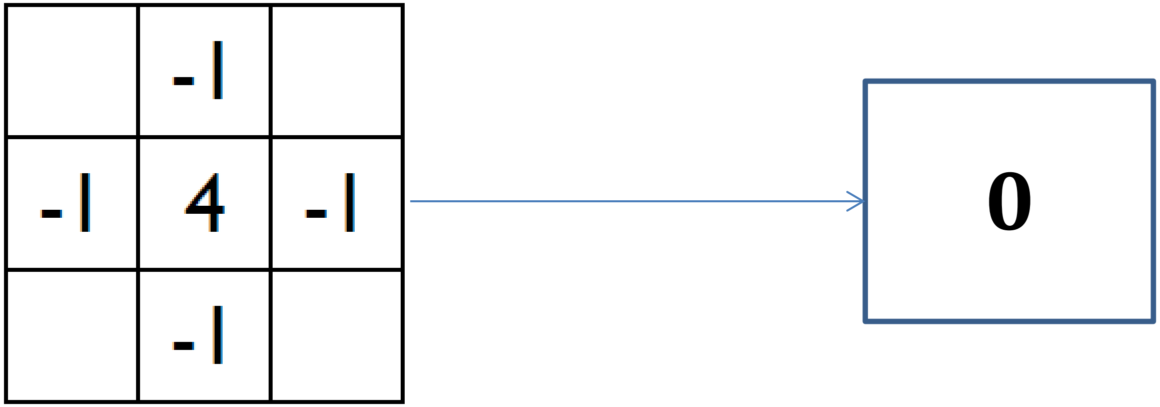
- *Stencil*
  - “A map in which each output depends on a *neighborhood* of inputs specified using a set of fixed offsets relative to the output position.”

Ref: McCool, Michael, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

The Stencil Pattern – Neighborhoods

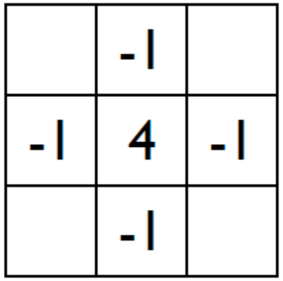
- *Stencil*
  - “A map in which each output depends on a *neighborhood* of inputs specified using a set of fixed offsets relative to the output position.”

Ref: McCool, Michael, James Reinders, and Arch Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

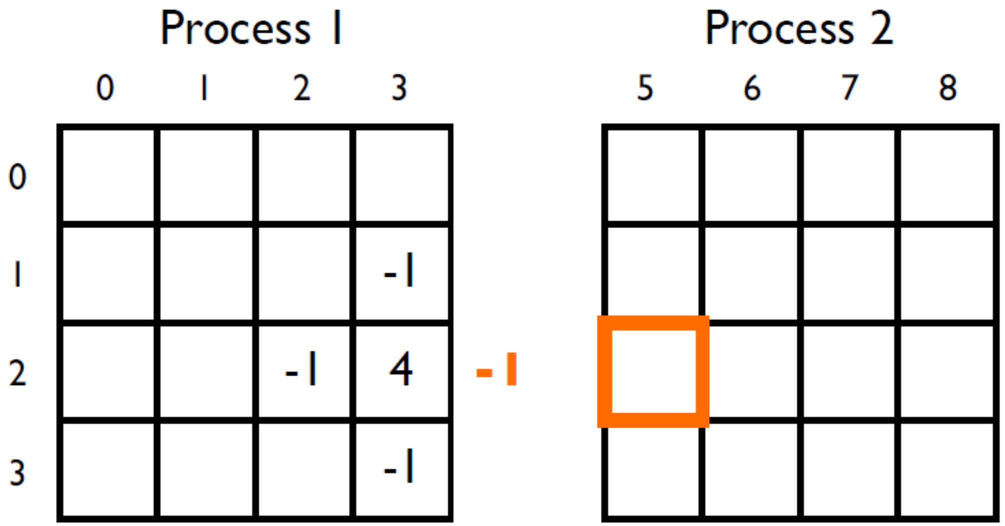


	-1	
-1	4	-1
	-1	

The Stencil Pattern



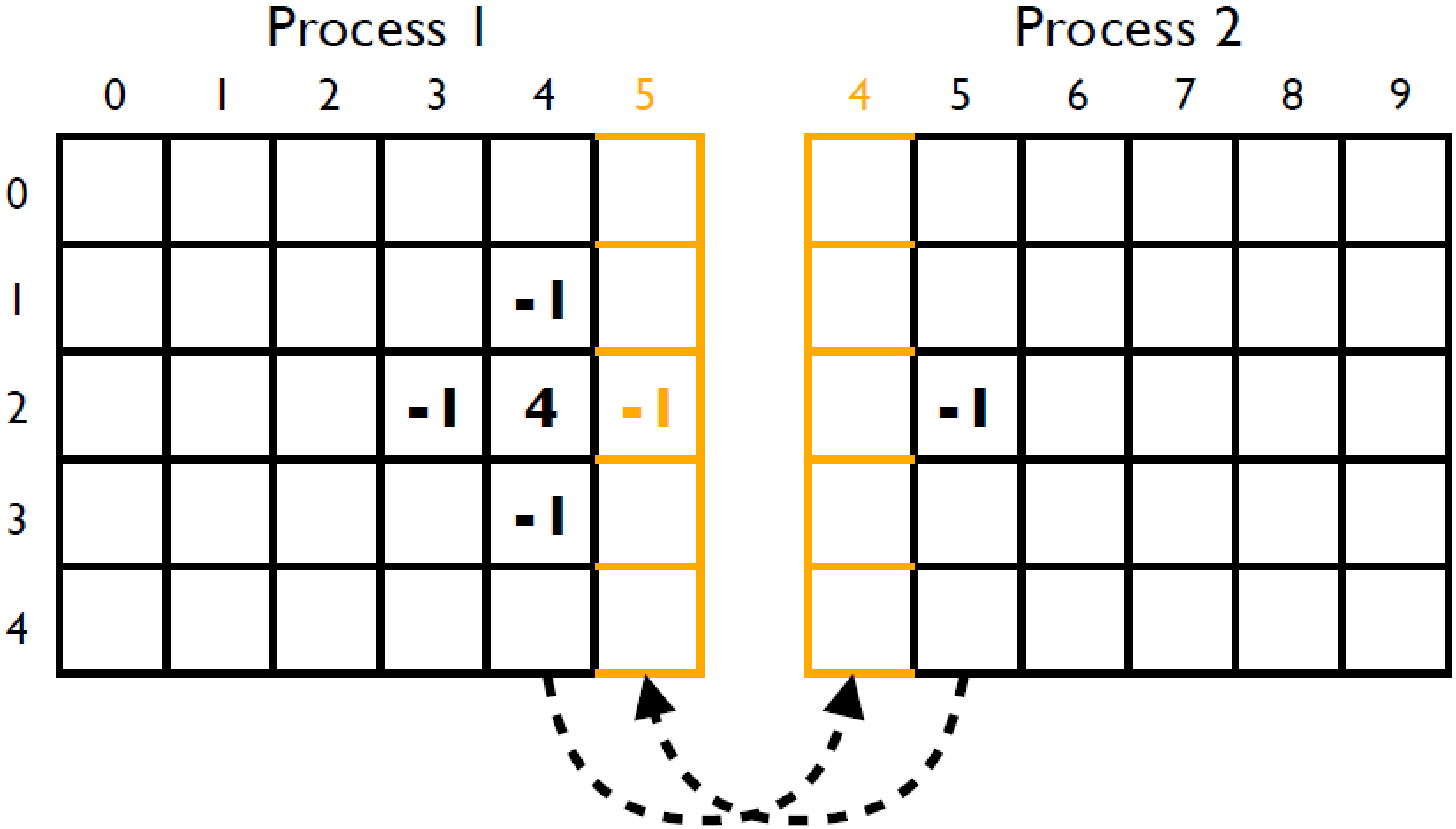
(a) 5-Point Stencil



(b) Stencil that needs a cell from its neighbor

Reference: Kjolstad, Fredrik Berg, and Marc Snir. "Ghost cell pattern.“ *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010.

The Stencil Pattern – Halo/Ghost Cells



Reference: Kjolstad, Fredrik Berg, and Marc Snir. "Ghost cell pattern.“ *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010.

Concurrency and Resource Sharing

- Multiple processes trying to access a shared resource, e.g., concurrent read and write to the same memory location
  - Leads to *Resource Contention*

Concurrency and Resource Sharing

- Multiple processes trying to access a shared resource, e.g., concurrent read and write to the same memory location
  - Leads to *Resource Contention*
- This makes concurrent/parallel programming hard
  - May lead to *Race Condition,* i.e., the final result of an operation depends on the order of execution

Concurrency and Resource Sharing

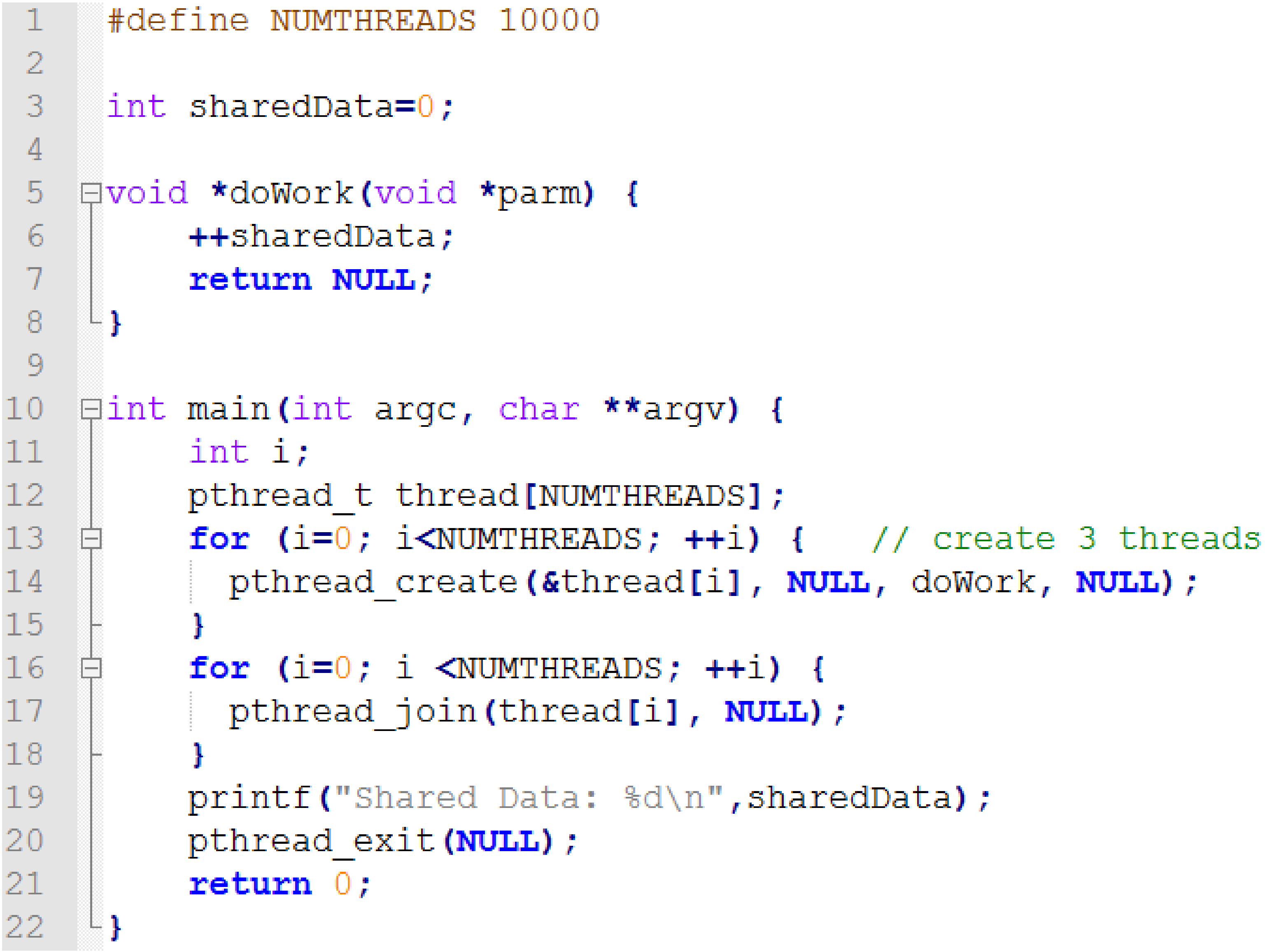
- Multiple processes trying to access a shared resource, e.g., concurrent read and write to the same memory location
  - Leads to *Resource Contention*
- This makes concurrent/parallel programming hard
  - May lead to *Race Condition,* i.e., the final result of an operation depends on the order of execution
    - *Deadlock:* Two or more concurrent tasks are unable to proceed, because each is waiting for one of the others to do something.

Concurrency and Resource Sharing

- Multiple processes trying to access a shared resource, e.g., concurrent read and write to the same memory location
  - Leads to *Resource Contention*
- This makes concurrent/parallel programming hard
  - May lead to *Race Condition,* i.e., the final result of an operation depends on the order of execution
    - *Deadlock:* Two or more concurrent tasks are unable to proceed, because each is waiting for one of the others to do something.
    - *Starvation:* A runnable task is overlooked indefinitely. Even though it is able to proceed, it is never chosen to run.

The *Minion-Banana* Tutorial on Concurrency and Resource Contention ☺

Race Condition



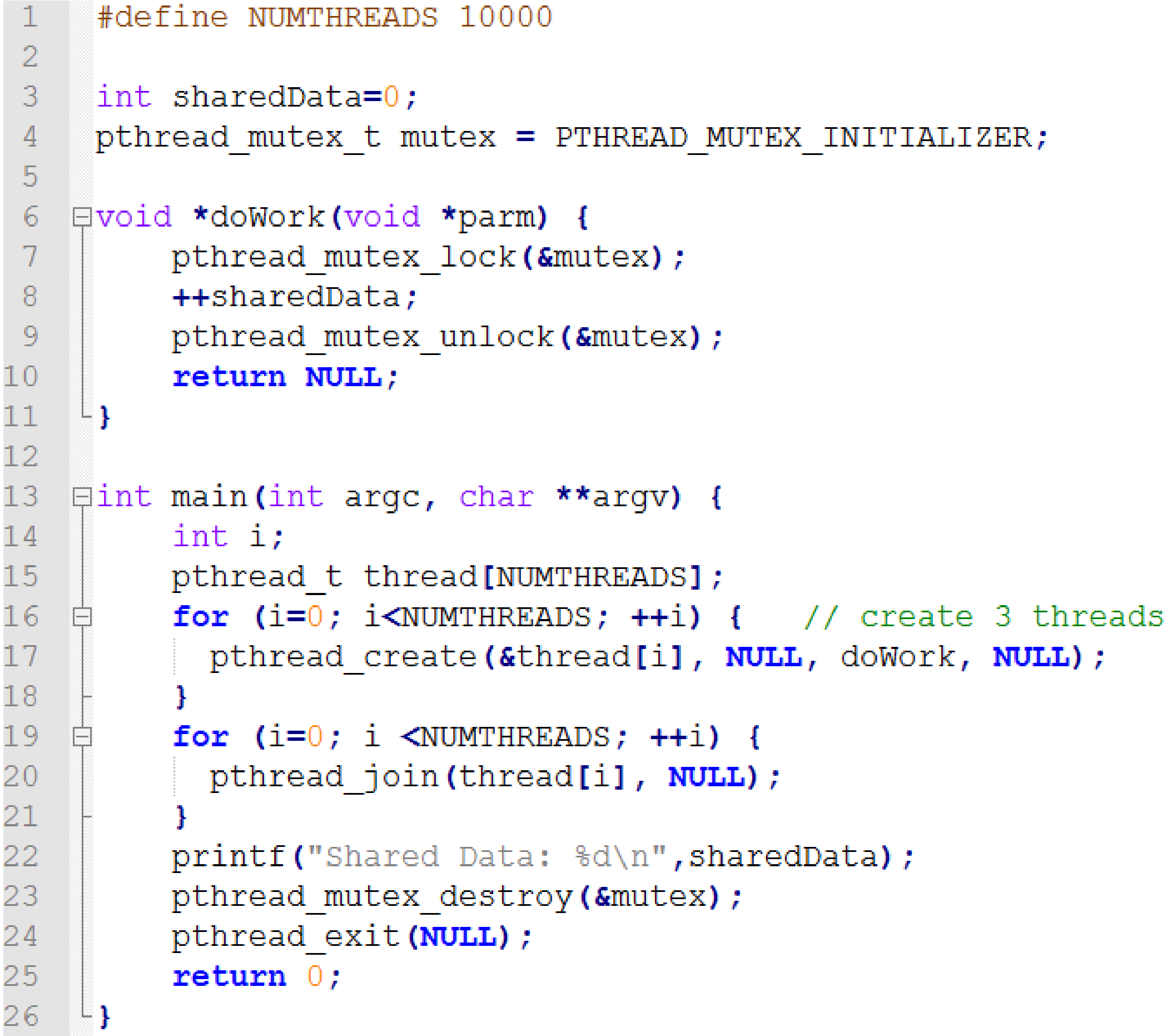
```c
#define NUMTHREADS 10000

int sharedData=0;

void *doWork(void *parm) {
    ++sharedData;
    return NULL;
}

int main(int argc, char **argv) {
    int i;
    pthread_t thread[NUMTHREADS];
    for (i=0; i<NUMTHREADS; ++i) {   // create 3 threads
      pthread_create(&thread[i], NULL, doWork, NULL);
    }
    for (i=0; i <NUMTHREADS; ++i) {
      pthread_join(thread[i], NULL);
    }
    printf("Shared Data: %d\n",sharedData);
    pthread_exit(NULL);
    return 0;
}
```

Mutex



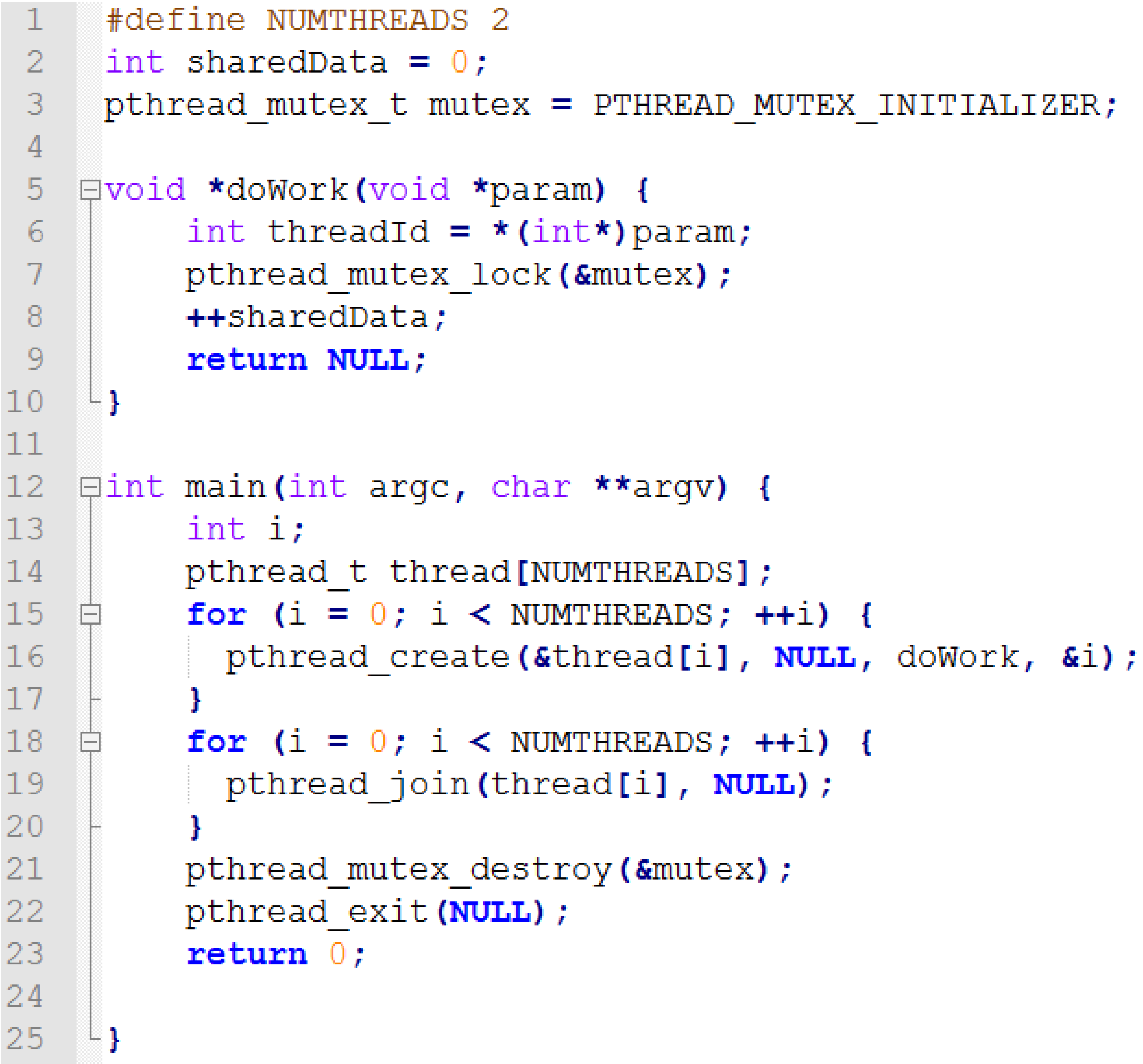
```c
#define NUMTHREADS 10000

int sharedData=0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *doWork(void *parm) {
    pthread_mutex_lock(&mutex);
    ++sharedData;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(int argc, char **argv) {
    int i;
    pthread_t thread[NUMTHREADS];
    for (i=0; i<NUMTHREADS; ++i) {    // create 3 threads
      pthread_create(&thread[i], NULL, doWork, NULL);
    }
    for (i=0; i <NUMTHREADS; ++i) {
      pthread_join(thread[i], NULL);
    }
    printf("Shared Data: %d\n",sharedData);
    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
    return 0;
}
```

Starvation



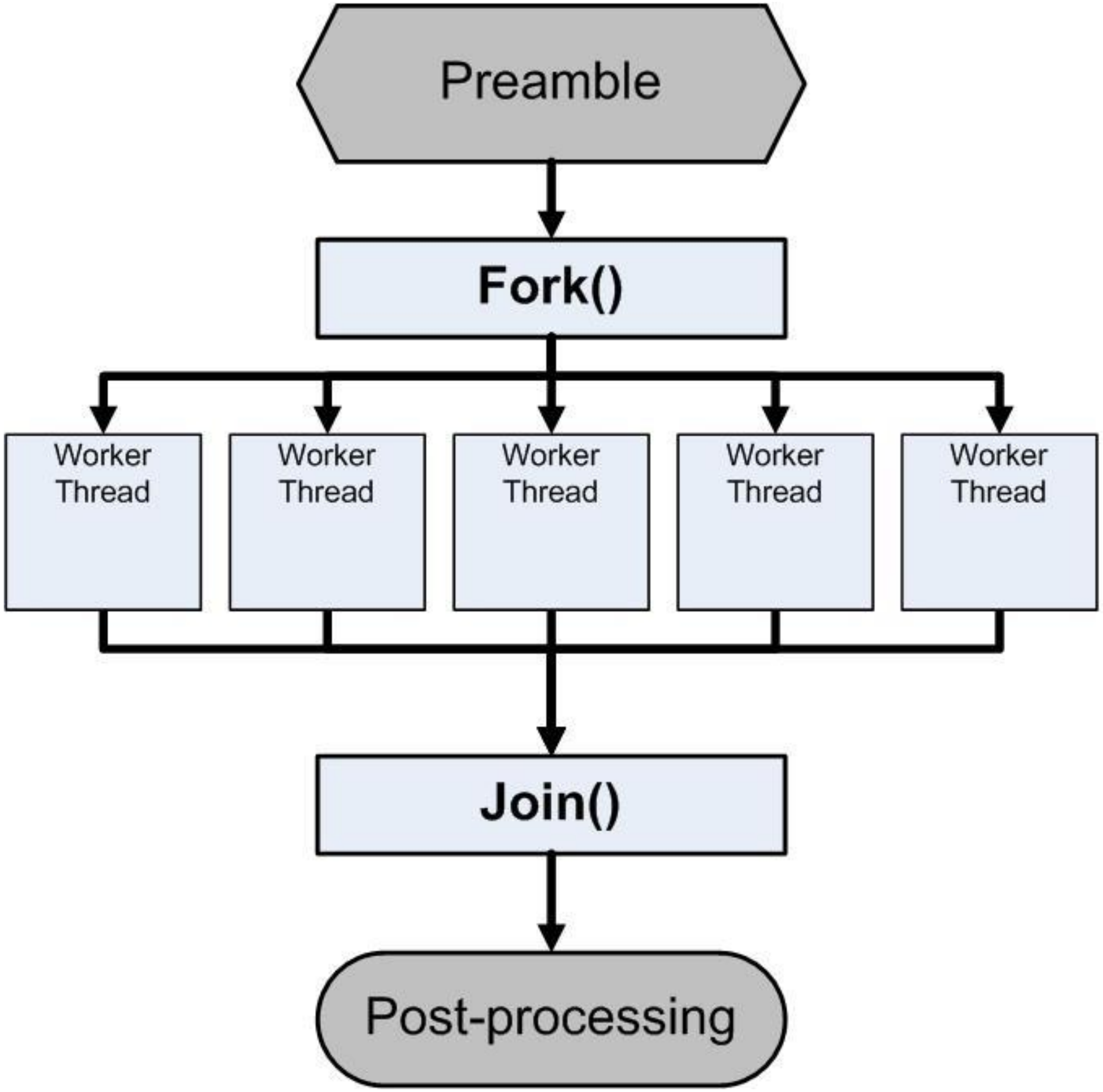
```c
#define NUMTHREADS 2
int sharedData = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *doWork(void *param) {
    int threadId = *(int*)param;
    pthread_mutex_lock(&mutex);
    ++sharedData;
    return NULL;
}

int main(int argc, char **argv) {
    int i;
    pthread_t thread[NUMTHREADS];
    for (i = 0; i < NUMTHREADS; ++i) {
      pthread_create(&thread[i], NULL, doWork, &i);
    }
    for (i = 0; i < NUMTHREADS; ++i) {
      pthread_join(thread[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
    return 0;

}
```

The Fork-Join Pattern

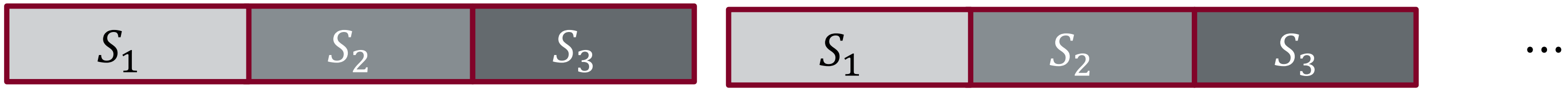


Ref: Mark B Friedman. Parallel Scalability Isn't Child's Play, Part 2: Amdahl's Law vs. Gunther's Law. Developer Division Performance Engineering blog, MSDN blogs.

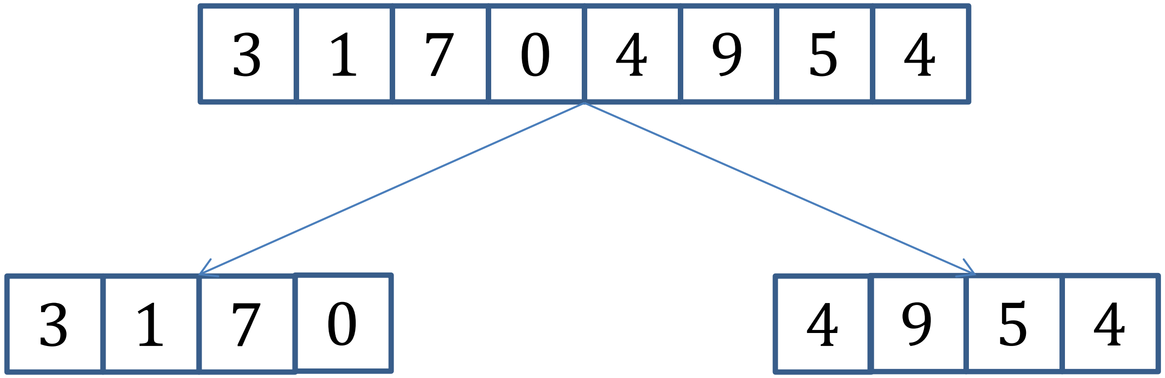
Parallel Pipeline Pattern

3	1	7	0	4	9	5	4

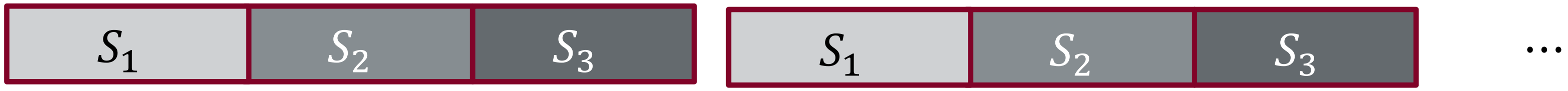
Serial processing of stages



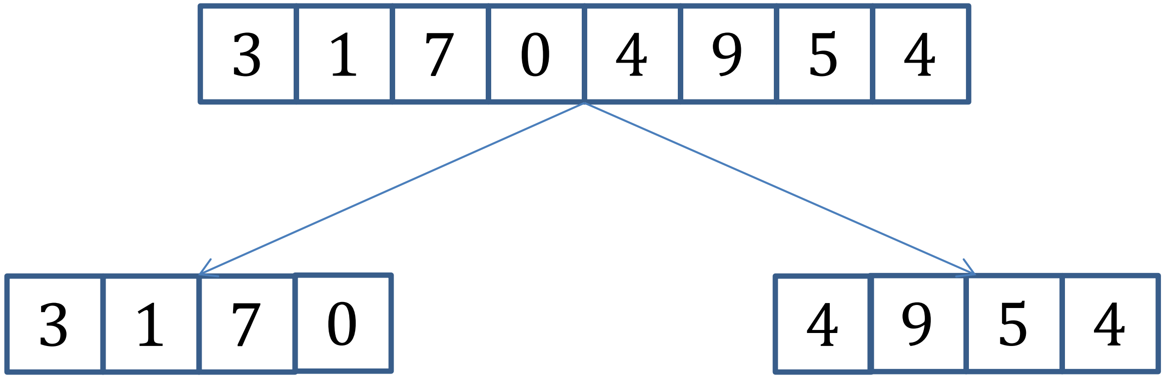
Parallel Pipeline Pattern



Serial processing of stages



Parallel Pipeline Pattern

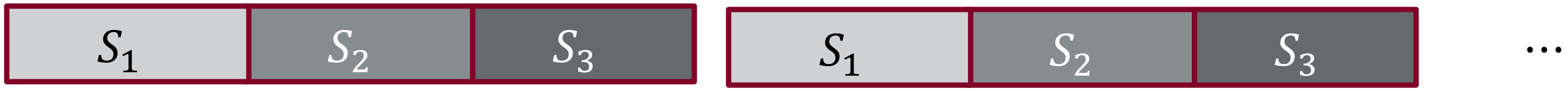


3	1	7	0	4	9	5	4

3	1	7	0

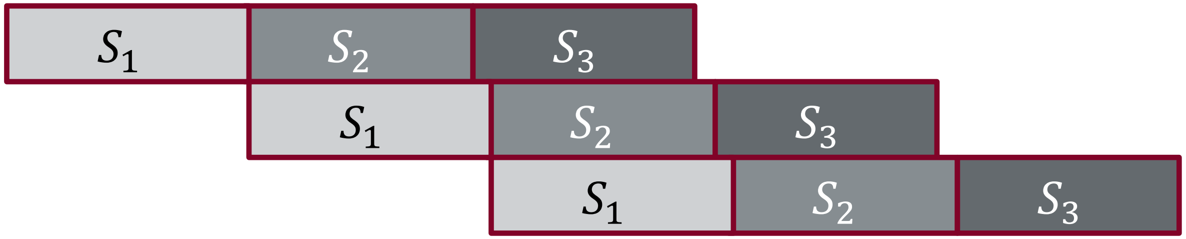
4	9	5	4

Serial processing of stages



S_1 S_2 S_3 S_1 S_2 S_3 …

Pipelined processing of stages



S_1 S_2 S_3

S_1 S_2 S_3

S_1 S_2 S_3

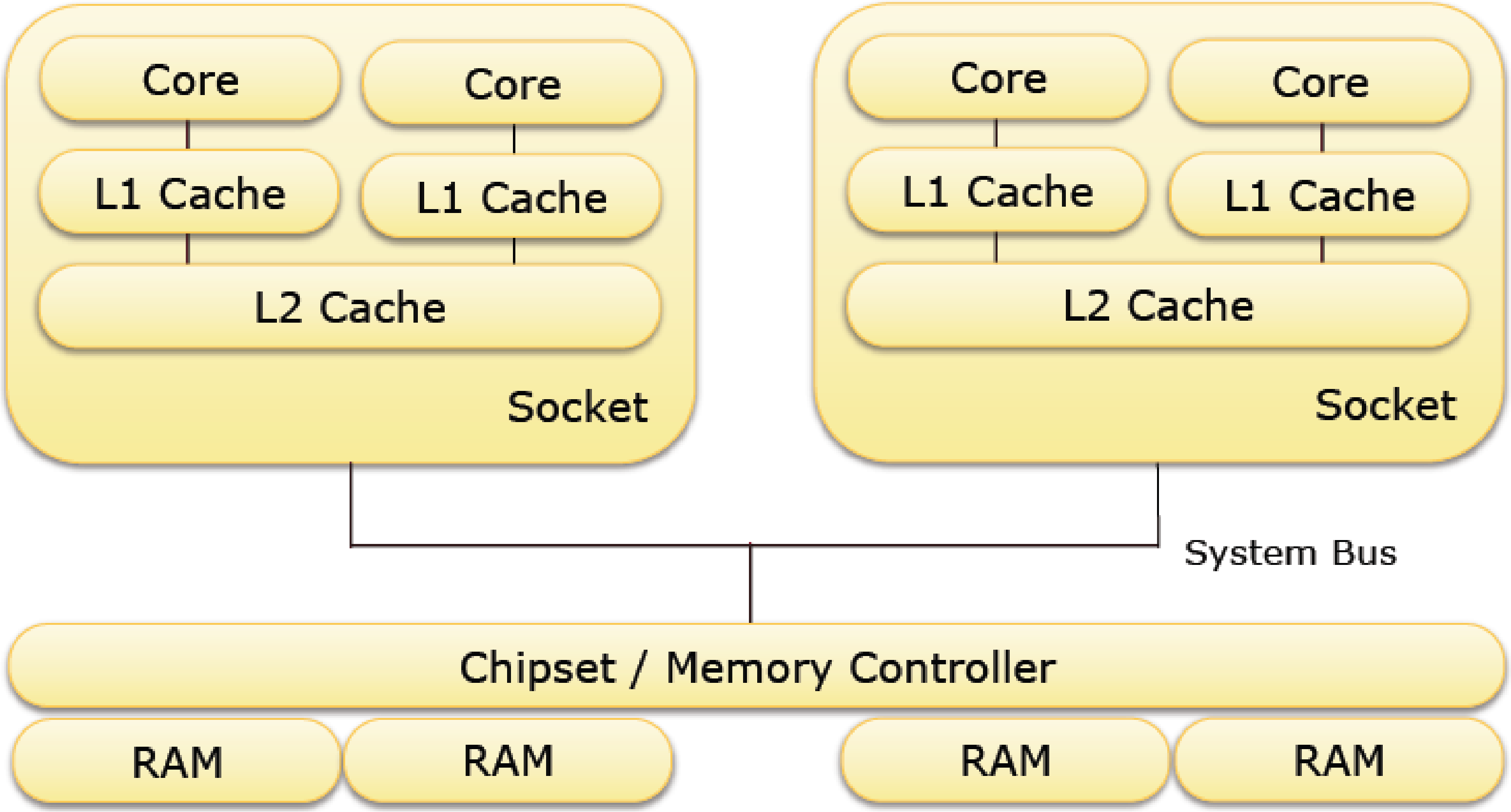
Part - III

Software Performance Optimization

"I really hate this damned machine;
I wish that they would sell it.
It never does quite what I want
but only what I tell it.“

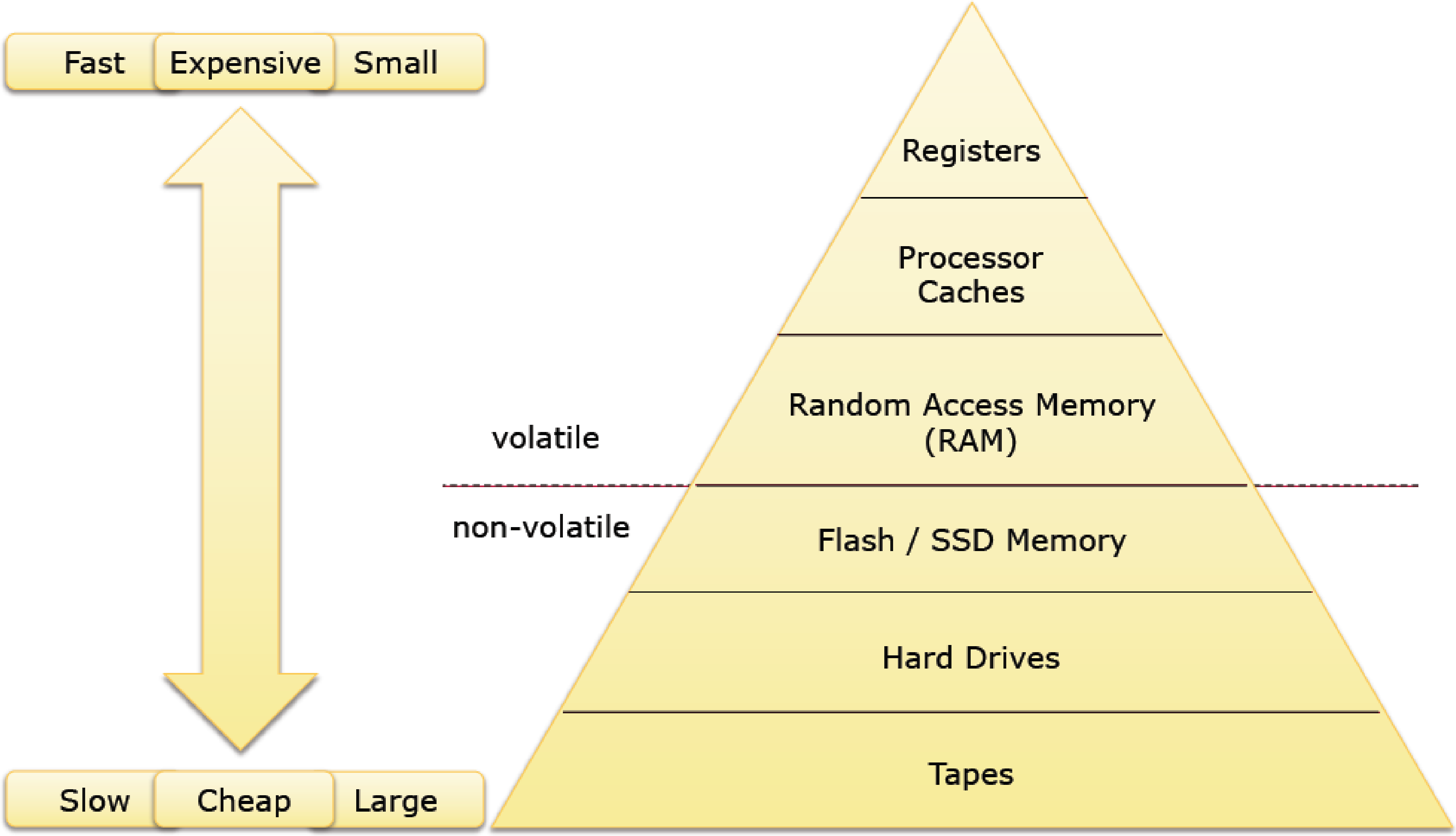
A Programmer’s Lament

Parallel Hardware Architecture and code Optimization



- Code must optimally utilize this complex hierarchy

Parallel Hardware Architecture and code Optimization



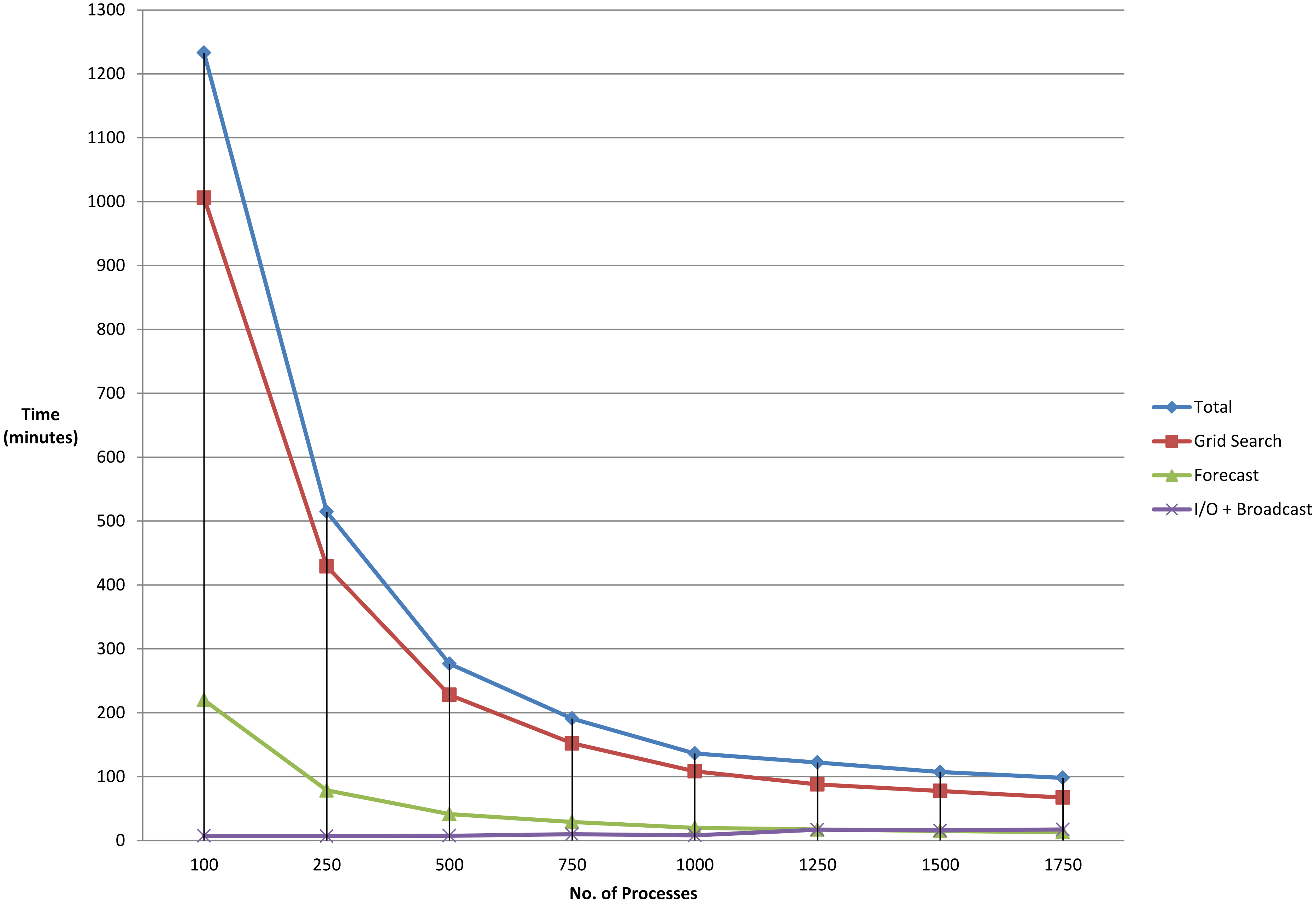
Part II – Motivation

Escaping the Ivory Tower Approach to Parallel Computing

Real Scientist: You suck!
Fahad: I know … I'm conscious, self-aware, …
Real Scientist : No, I mean you *really* suck!
Fahad: Oh, gee … thanks?
Real Scientist: I see all this parallel programming crap, but where are your applications? … How do I use all this stuff?
Fahad: ☹

Simulation of Seismic Events following Earthquakes

- Collaborative project with the *GFZ German Research Center for Geosciences*
    - The *2011 earthquake off the Pacific coast of Tōhoku, Japan*
    - 250 days of data used in the numerical simulation
    - Prediction of the *No. and location of aftershocks*, e.g., *Fukushima*


1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
Time (minutes)
100
250
500
750
1000
1250
1500
1750
No. of Processes
Total
Grid Search
Forecast
I/O + Broadcast

Simulation of Seismic Events following Earthquakes

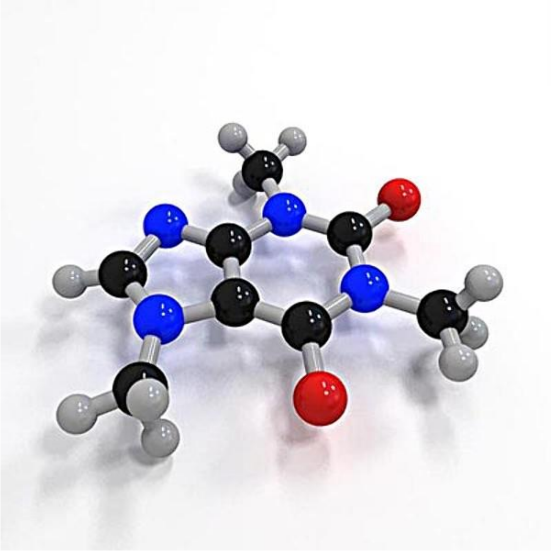
- Distributed-memory Parallelization using MPI
  - *1000-core FutureSOC cluster* used for performance evaluation
  - Successful scaling up to *1750* processes
  - Simulation run that would have taken *~6 days* can now be done in *~1.5 hours*

Enumeration of Elementary Flux Modes in Metabolic Networks

Guinea Pig

Enumeration of Elementary Flux Modes in Metabolic Networks

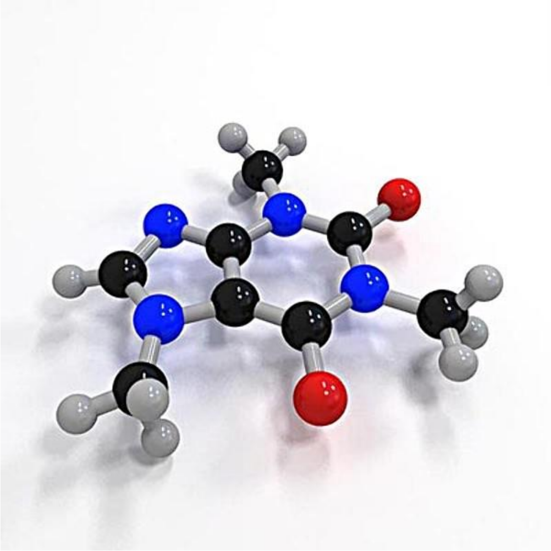
Guinea Pig



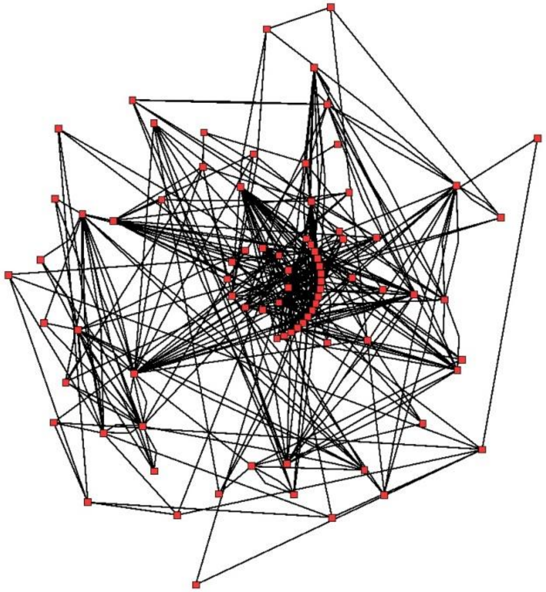
Caffeine Molecule

Enumeration of Elementary Flux Modes in Metabolic Networks

Guinea Pig



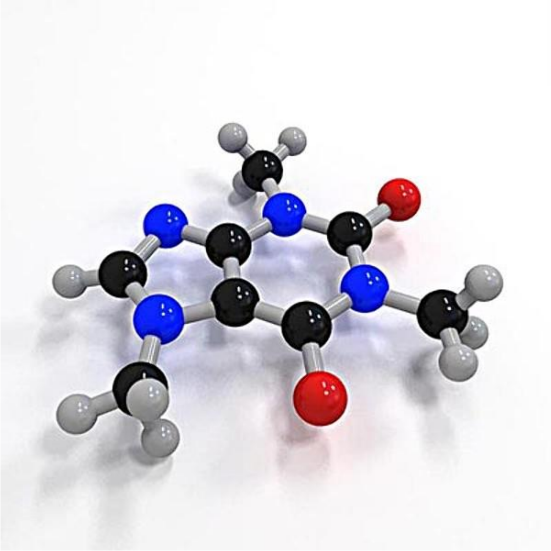
Caffeine Molecule



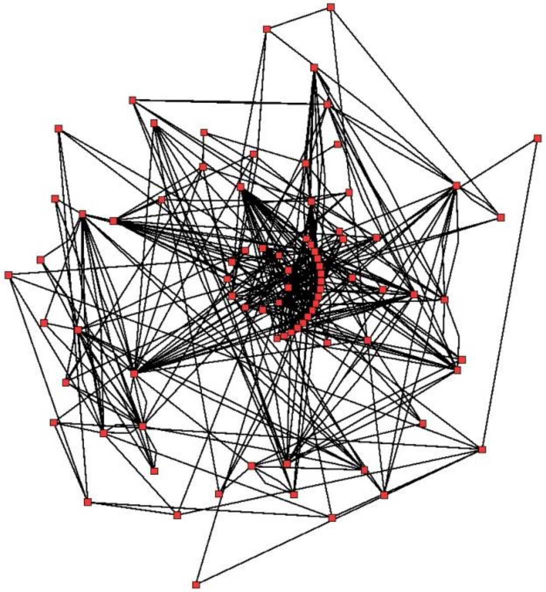
Plant Metabolic Network

Enumeration of Elementary Flux Modes in Metabolic Networks

Guinea Pig



Caffeine Molecule



Plant Metabolic Network

Enumeration of Elementary Flux Modes in Metabolic Networks

- Objective
  - To understand structural properties of *Metabolic Networks*
  - Understand the range of *Metabolic Pathways* in the network
  - Gain insight into the *contribution of pathways* to the overall metabolic network behavior
- Potential Applications
  - Treatments for metabolic disorders such as *Diabetes*
  - Therapies for *Cancer* patients
  - Metabolic Engineering: Production of *Biofuels*

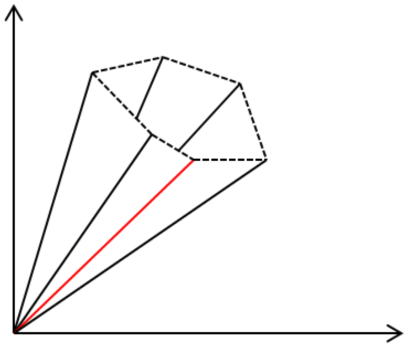
Enumeration of Elementary Flux Modes in Metabolic Networks

$$\frac{dX_1}{dt} = 1 \cdot v_1 - 1 \cdot v_2 - 1 \cdot v_4$$

$$\frac{dX_2}{dt} = 2 \cdot v_2 - 1 \cdot v_3$$

$$\begin{tikzcd}
X_0 \arrow[r, "v_1"] & X_1 \arrow[r, "v_2"] \arrow[d, "v_4"] & 2X_2 \arrow[r, "v_3"] & X_3 \\
 & X_4 & &
\end{tikzcd}$$

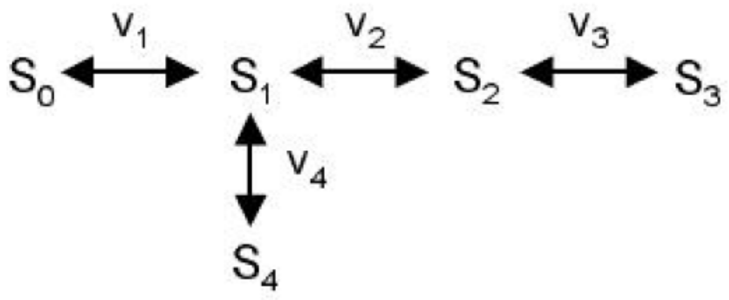
$$\Rightarrow \begin{pmatrix} \frac{dX_1}{dt} \\ \vdots \\ \frac{dX_m}{dt} \end{pmatrix} = \begin{pmatrix} 1 & -1 & \cdots & S_{1n} \\ \vdots & \ddots & & \vdots \\ S_{m1} & \cdots & & S_{mn} \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

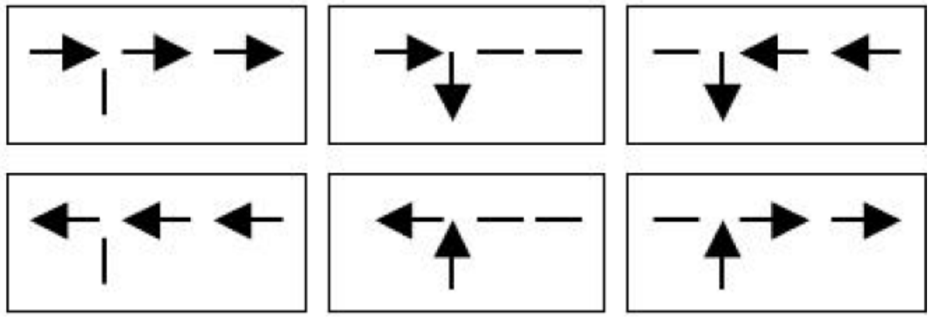


- □ S is the stoichiometric matrix
- □ At steady state, $0 = S \cdot v$
- □ Also, some reactions are irreversible, i.e., $v_i \geq 0$
- □ This makes the solution space a *convex polyhedral cone*

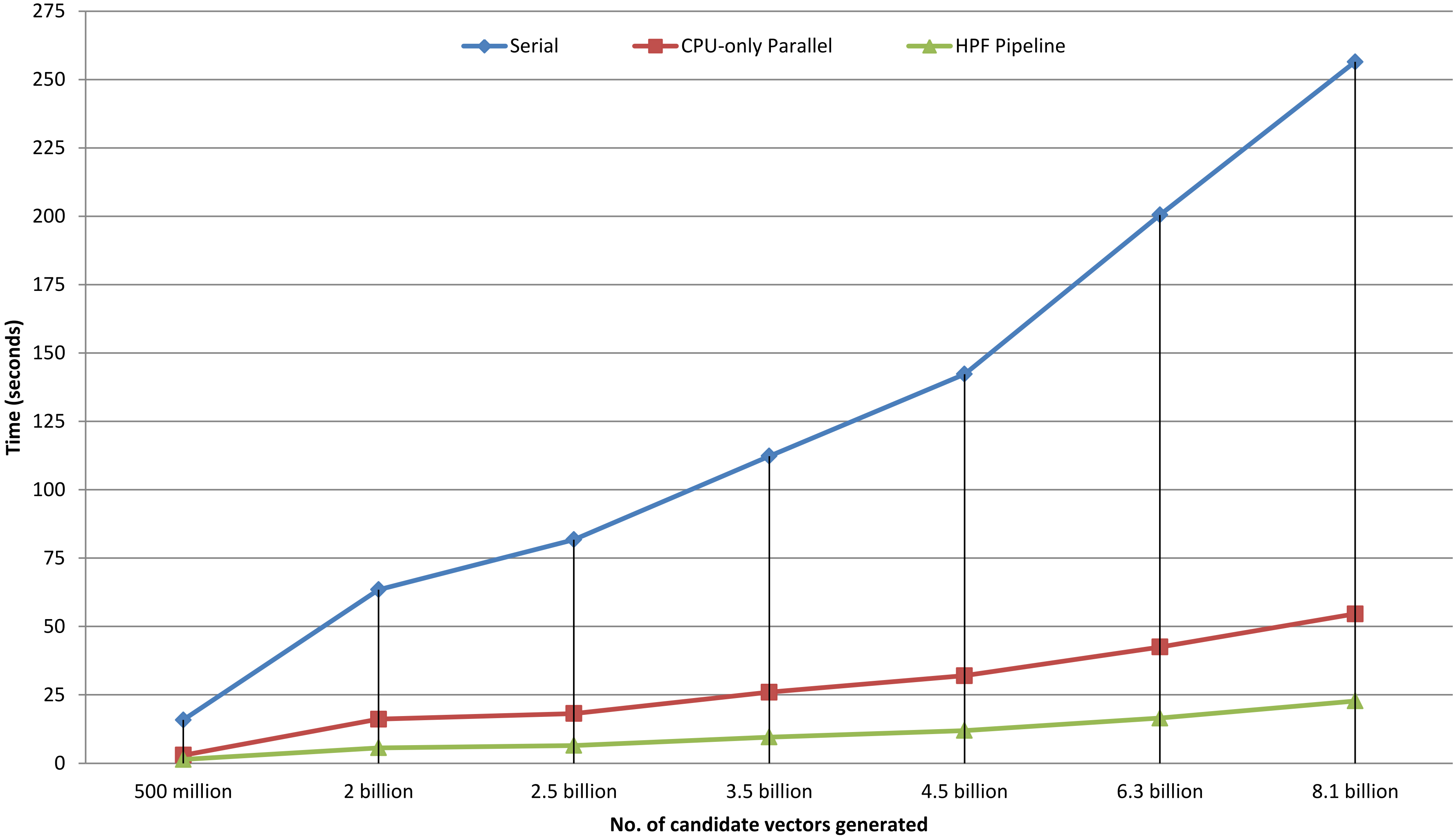
Enumeration of Elementary Flux Modes in Metabolic Networks

- Elementary Flux Mode (EFM)
  - *is a minimal set of reactions that can operate at steady state*
  - EFMs span the feasible space of *flux distributions*
  - Any feasible flux distribution can be represented as a linear combination of EFMs
- Example





Enumeration of Elementary Flux Modes in Metabolic Networks



Recap

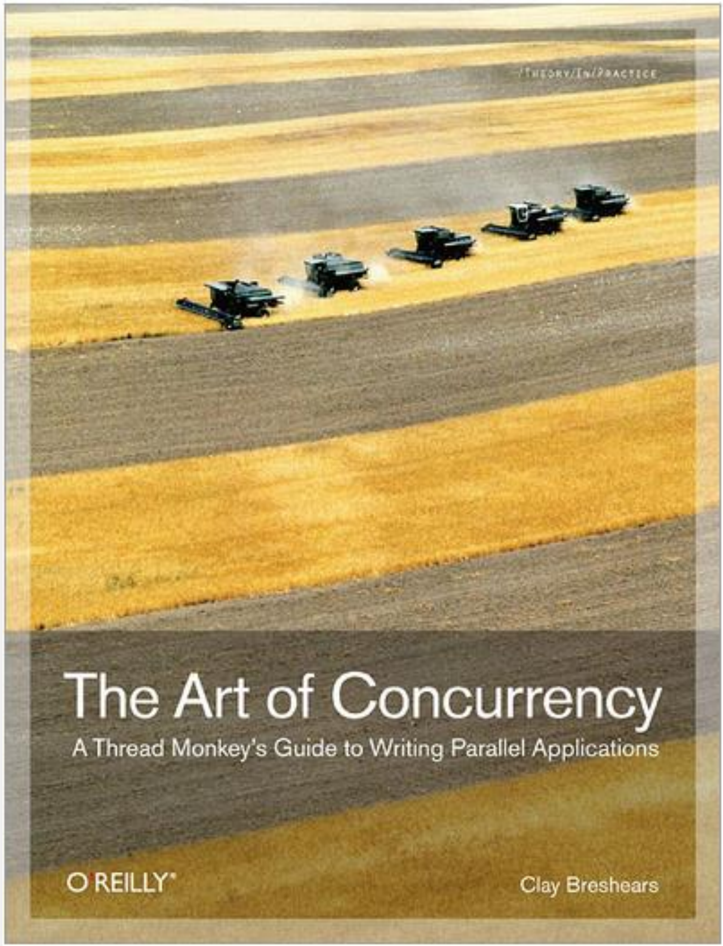
- *Concurrency* vs. *Parallelism*
- *Shared-memory* vs. *Distributed-memory* Parallelism
- *Hotspots*
- Patterns in Parallel Programming: *Geometric Decomposition*, *Map*, *Reduce*, *Stencil*, *Fork-Join*, *Pipeline*, …
- *Resource contention*, *Race condition*, *Deadlock*, *Starvation*
- *Software Performance Optimization*

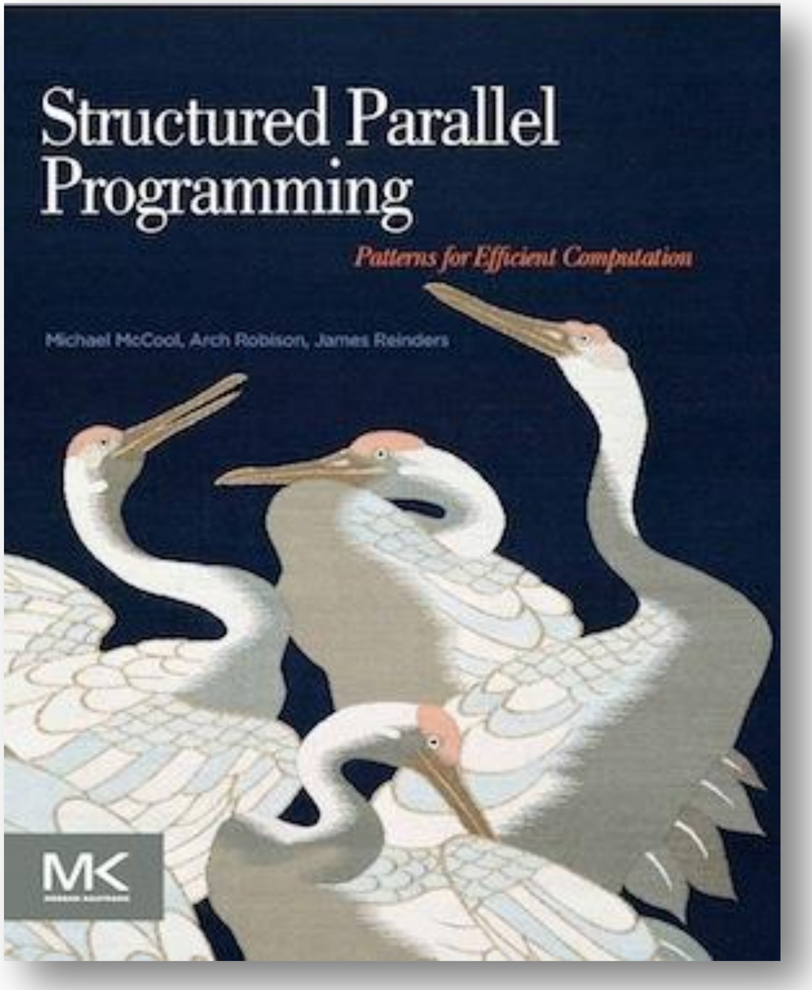
Recap

- *Concurrency* vs. *Parallelism*
- *Shared-memory* vs. *Distributed-memory* Parallelism
- *Hotspots*
- Patterns in Parallel Programming: *Geometric Decomposition*, *Map*, *Reduce*, *Stencil*, *Fork-Join*, *Pipeline*, …
- *Resource contention*, *Race condition*, *Deadlock*, *Starvation*
- *Software Performance Optimization*

Thank You!

Literature




Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch Robison, James Reinders
MK

Free MOOC – *Parallel Programming Concepts*, OpenHPI