COMP 364 / 464
High Performance Computing

Introduction to High Performance and Parallel Computing

Outline

- Overview of HPC
- Topics to be covered
- Theoretical background
- Parallel computing systems
- Parallel programming models
- MPI/OpenMP examples

OVERVIEW

What is High Performance Computing?

- Make a given problem run faster.
  - Faster algorithm: analyze the complexity
  - Scalar optimization: improve memory and computational efficiency (reduce data movement)
  - Parallelism: use multiple cores or multiple nodes
- Make a small problem larger.
  - Scale a small problem from a desktop to a cluster.
  - Scale a large problem from a cluster to a supercomputer.

What are we going to learn?

- Scalar optimizations:
  - Improve the memory efficiency of an application and learn why it matters.
  - Learn how to improve the instruction efficiency.
- Parallelism:
  - Vectorization: parallelism within a single core
  - Multi-threading: parallelism across cores on the same CPU or node (shared-memory)
  - Multi-processing: parallelism across CPUs or nodes (distributed-memory)

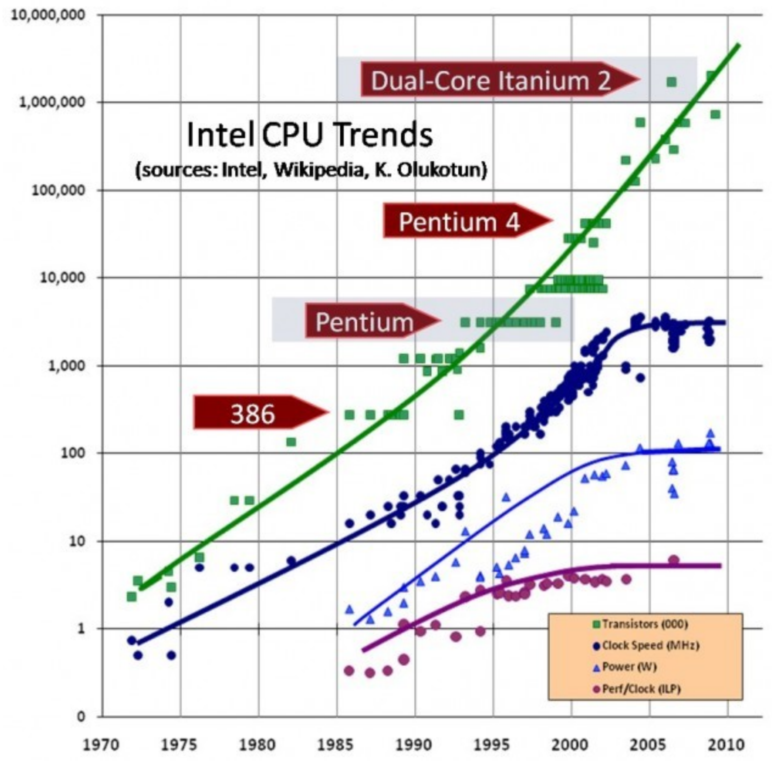
What is Parallel Computing?

- Parallel computing: use of multiple processors or computers working together on a common task.
  - Each processor works on part of the problem
  - Processors can exchange information

http://selectnews.ro/prima-achizitie-majora-de-terenuri-agricole-in-romania-fac-filantropii-americani/

Why do we need Parallel Computing?

- We've hit the 'power wall'
- Single processor performance roughly doubled every 18-24 months since mid-70's by increasing the CPU clock frequency.



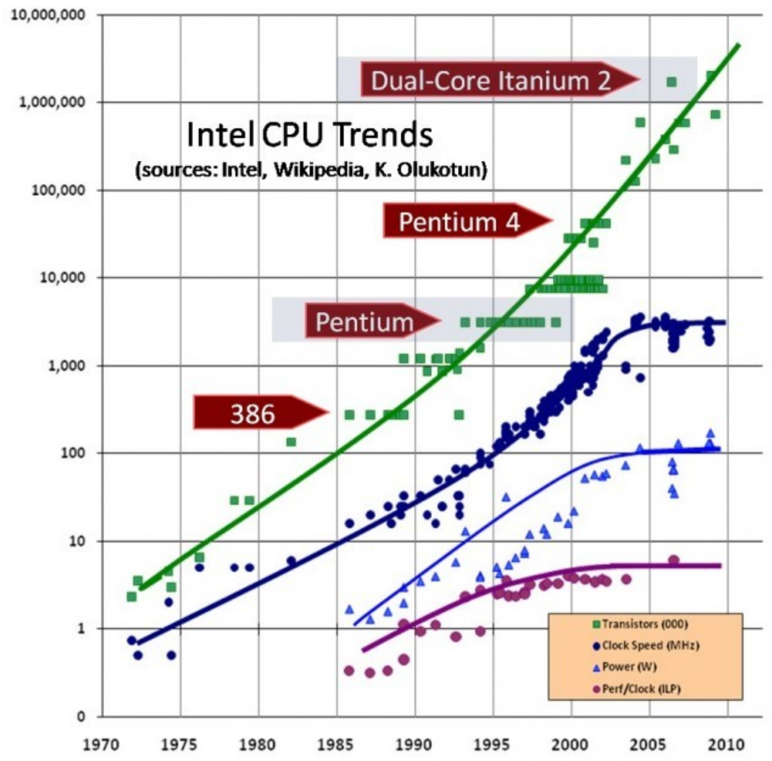
The death of CPU scaling: From one core to many — and why we're still stuck,' J. Hruska.

Why do we need Parallel Computing?

- Processor performance doubling every ~18 months is often termed "Moore's Law" after Gordon Moore, a co-founder of Intel back in 1965.
  - He observed that the # of transistors in an integrated circuit roughly doubled every 1-2 years.
  - As features shrink, they tend to be quicker -> faster CPUs.
- Other factors that are critical to the performance doubling observation:
  - Dennard scaling: power / area is roughly constant
  - This implies that the performance per power grows exponentially (roughly doubling every 1.6 years).

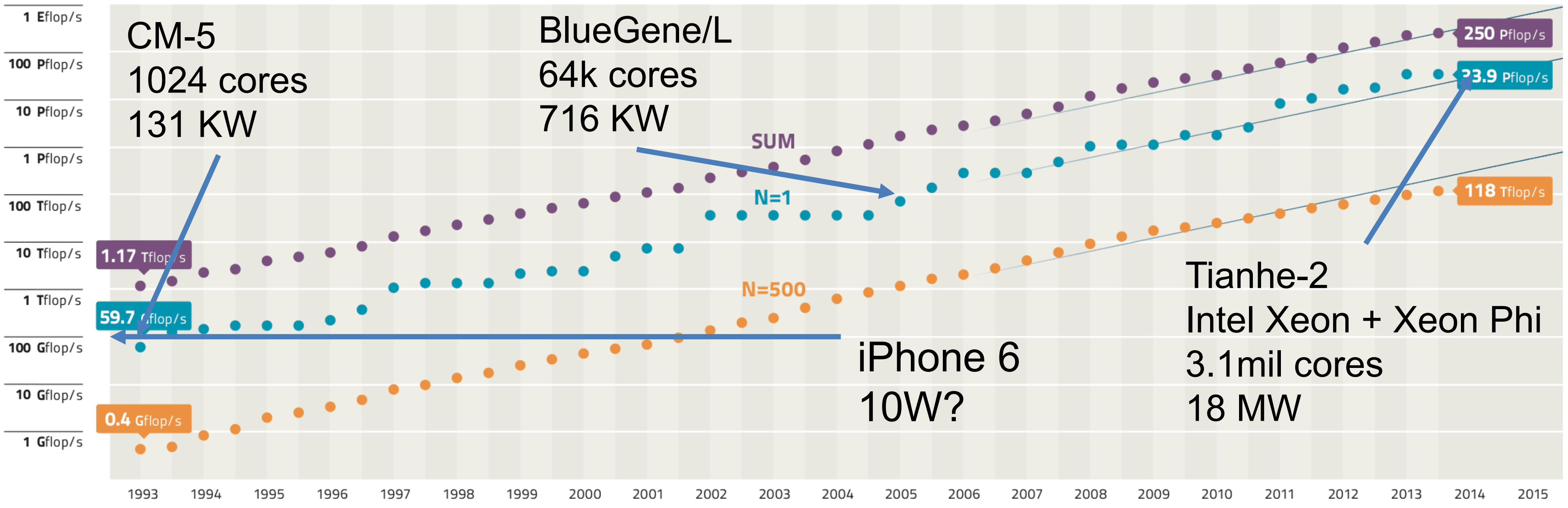
Why do we need Parallel Computing?

- Dennard scaling broke down around ’06 … power not constant as the frequency increased.
- But transistors keep shrinking (for now)
- Instead of increasing frequency … replicate (parallel) and increase on-chip memory (cache)?
- Increases the net performance w/o increasing the power consumption.



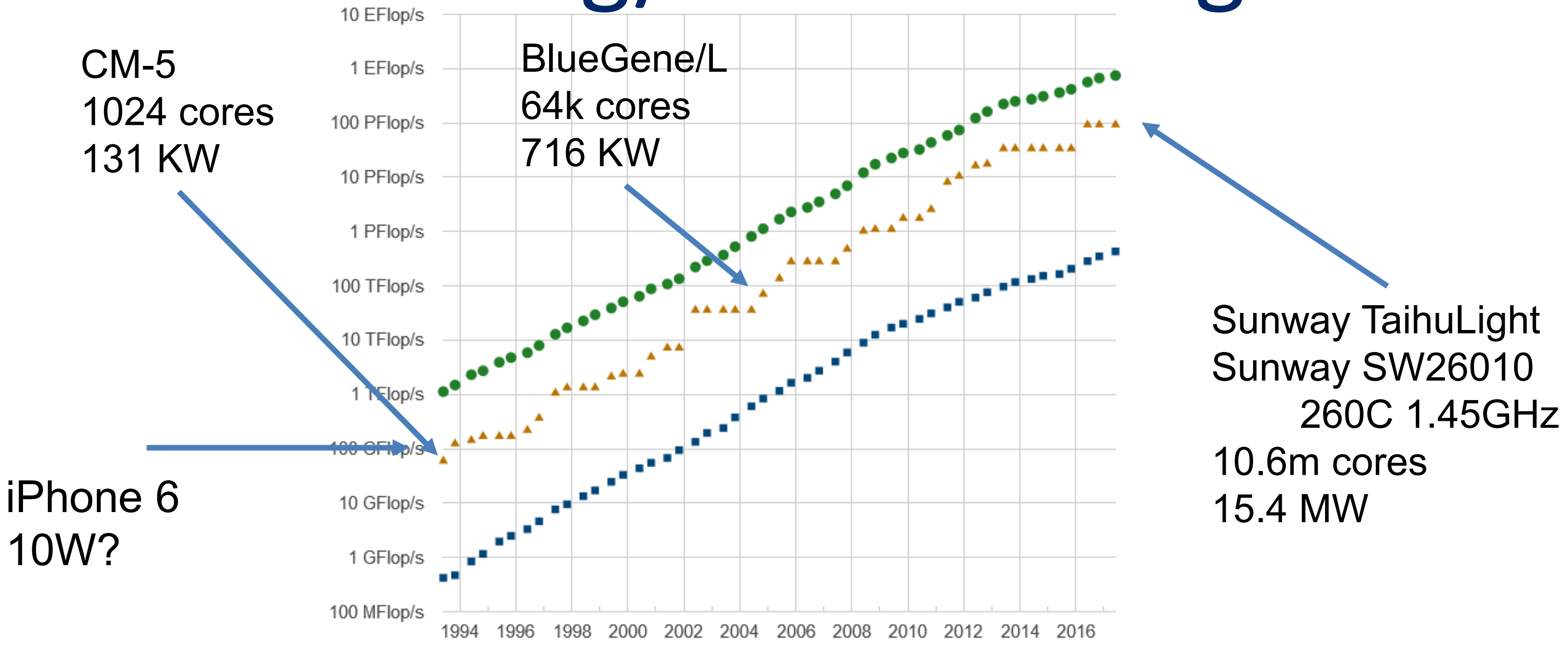
The death of CPU scaling: From one core to many — and why we’re still stuck,’ J. Hruska.

How big/fast can we go?



- Plot of Top500 winner (#1), #500, and sum of all 500 of the fastest supercomputers in the world since '93.
- Lifespan ~8 years from #1 to #500.

How big/fast can we go?



- Plot of Top500 winner (#1), #500, and sum of all 500 of the fastest supercomputers in the world since ‘93.
- Lifespan ~8 years from #1 to #500.

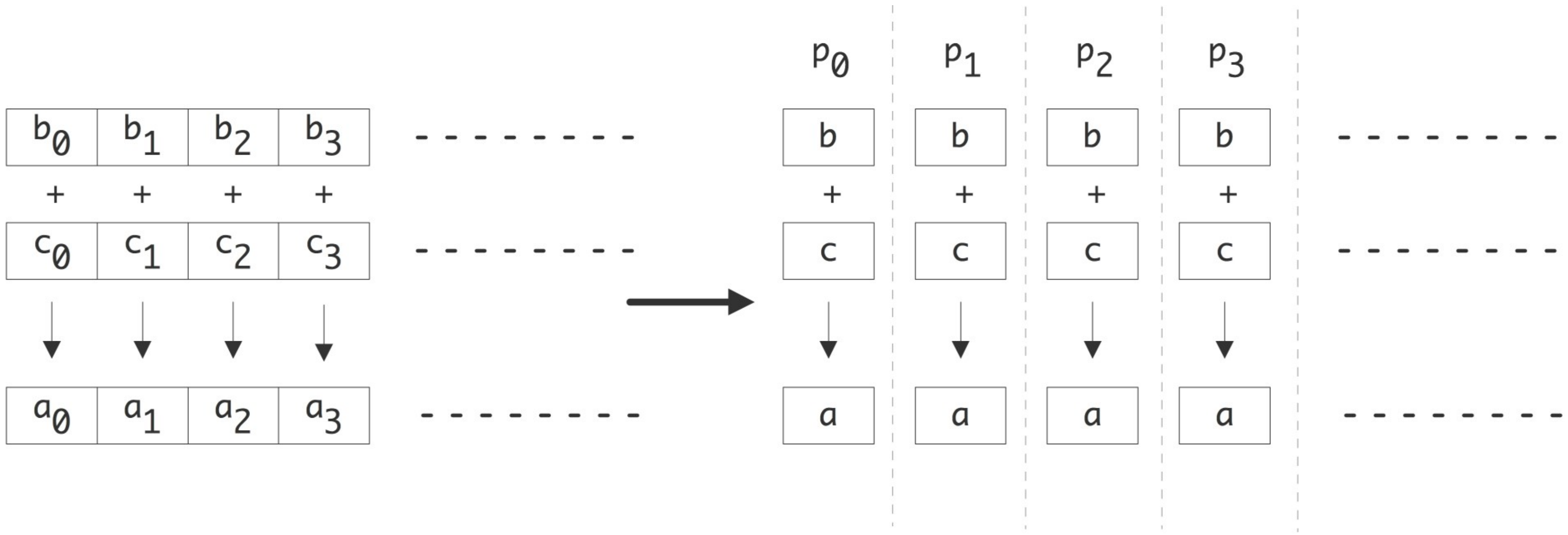
Parallelism: The Basic Idea

- Spread operations over many processors or instruction streams.
- If *n* operations take time *t* on 1 processor,
- Does this become *t/p* on *p* processors ($p <= n$)?

```
for (i = 0; i < n; ++i)
   a[i] = b[i] + c[i];
```

Idealized version: every core has one array element

The Basic Idea (Idealized Version)



The Basic Idea

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor…
- …does this become *t/p* on *p* processors (*p<=n*)?

```
for (i = 0; i < n; ++i)
  a[i] = b[i] + c[i];
```

Idealized version: every process has one array element

Slightly less ideal: each core has part of the array

```
for (i = my_begin; i < my_end; ++i)
  a[i] = b[i] + c[i];
```

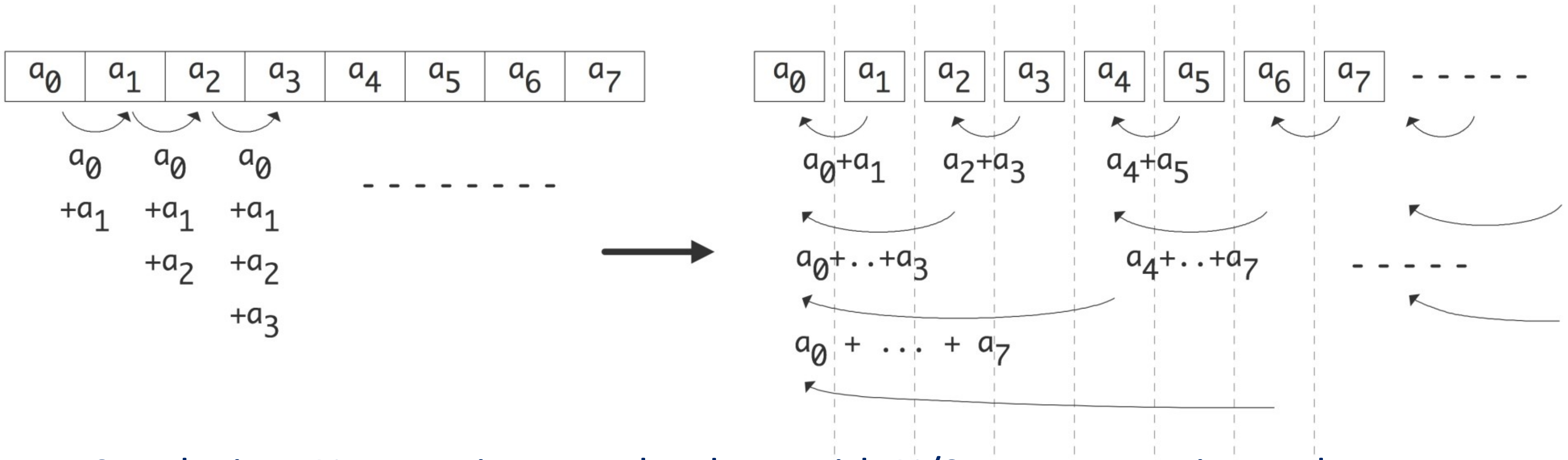
The Basic Idea (…)

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor…
- …does this become *t/p* on *p* processors (*p*<=*n*)?

```
sum = 0.0;
for (i = 0; i < n; ++i)
  sum = sum + a[i];
```

Can we do this in parallel?

The Basic Idea (Continued)



a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
a_0 $+a_1$
a_0 $+a_1$ $+a_2$
a_0 $+a_1$ $+a_2$ $+a_3$
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
a_0+a_1
a_2+a_3
a_4+a_5
$a_0+..+a_3$
$a_4+..+a_7$
$a_0 + ... + a_7$

Conclusion: N operations can be done with N/2 processors, in total time log2(N).

Theoretical question: can addition be done faster?

Practical question: can we even do this?

The Basic Idea (…)

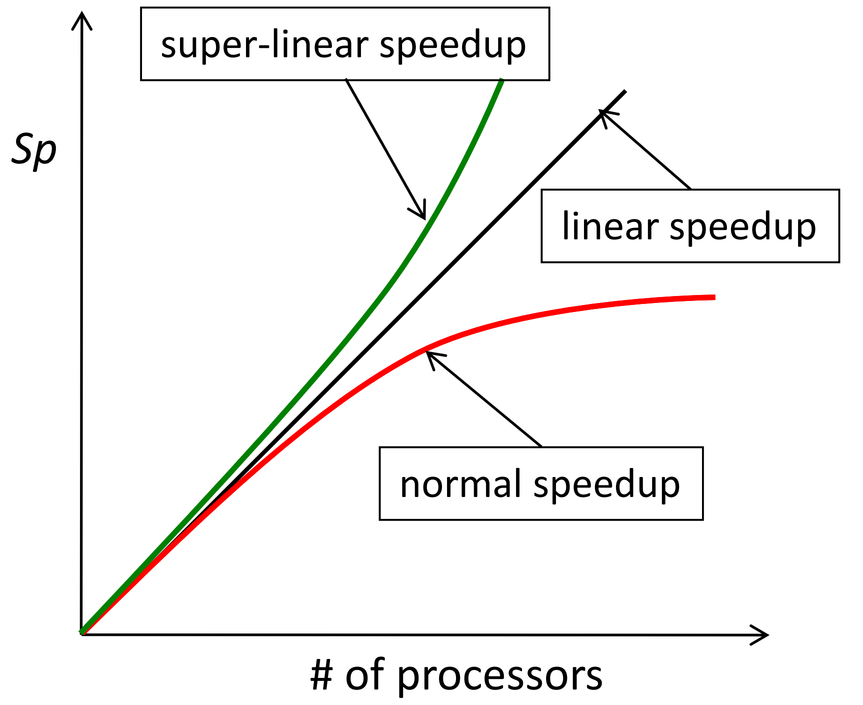
- Spread operations over many processors
- If *n* operations take time *t* on 1 processor…
- …does this become *t/p* on *p* processors (*p<=n*)?

```
for (k = 0; k < log2(n); ++k) {
  s = pow(2,k); /* stride = 2^k */
  p = 2*s; /* increment = 2*s */
  for (i = 0; i < n; i += p)
    a[i] = a[i] + a[i+s];
}
```

THEORETICAL BACKGROUND

Speedup & Parallel Efficiency

- Speedup: $S_p = \dfrac{T_s}{T_p}$
  - *p* = parallelism (e.g., # of cores)
  - *Ts* = execution time of the sequential algorithm
  - *Tp* = execution time of the parallel algorithm with *p* cores
  - *Sp= P* (linear speedup: ideal)



- Parallel efficiency: $E_p = \dfrac{S_p}{p} = \dfrac{T_s}{pT_p}$

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections
- Other Considerations
  - Time to re-write code

Amdahl's Law

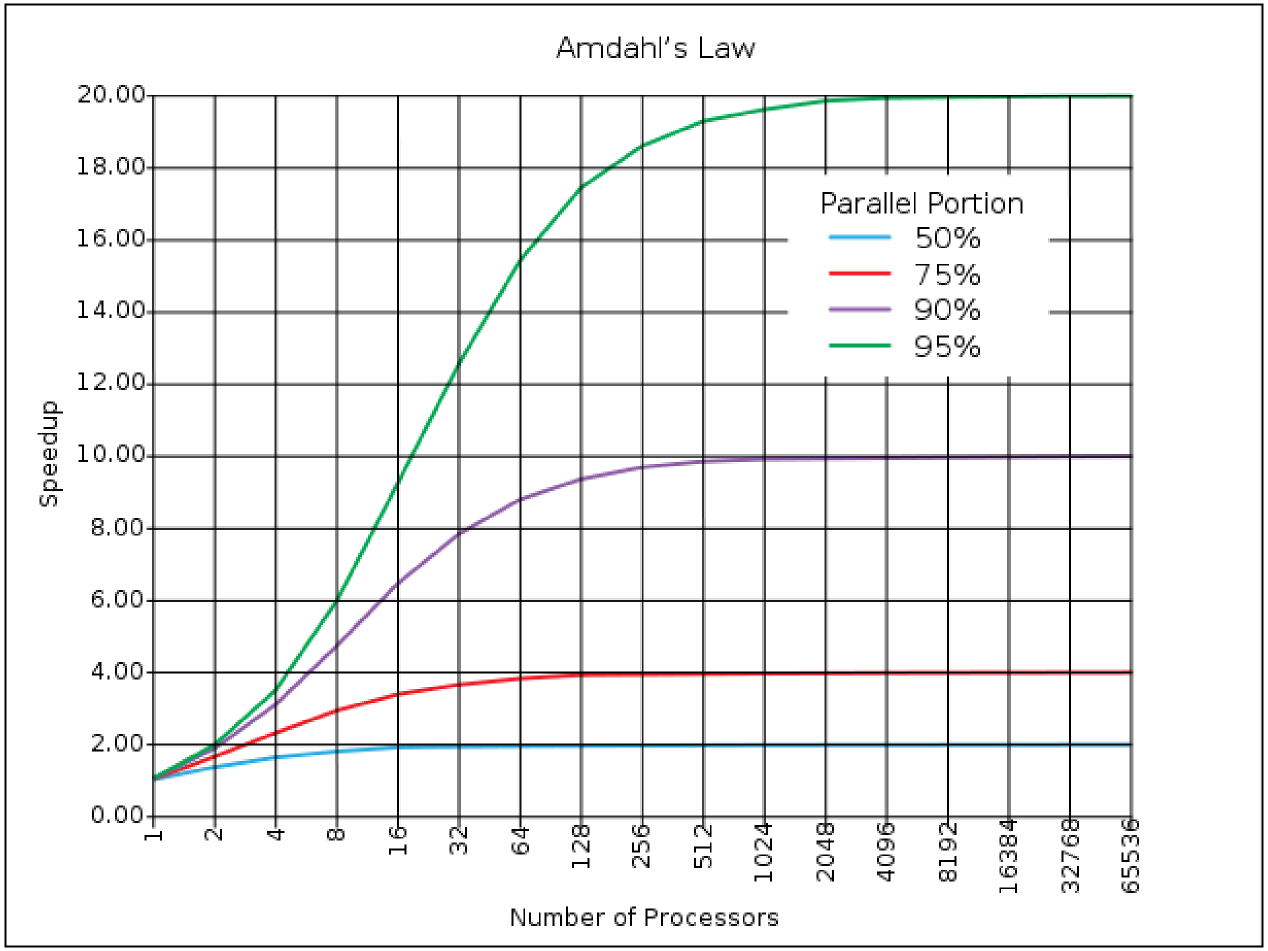
- All parallel programs contain parallel sections and serial sections
- Serial sections limit the parallel effectiveness (efficiency)
- Amdahl's Law states this formally
  - Effect of parallelism on speed up

$$S_P \leq \frac{T_S}{T_P} = \frac{1}{f_s + \frac{f_p}{P}} \rightarrow \frac{1}{f_s}, p \rightarrow \infty$$

where

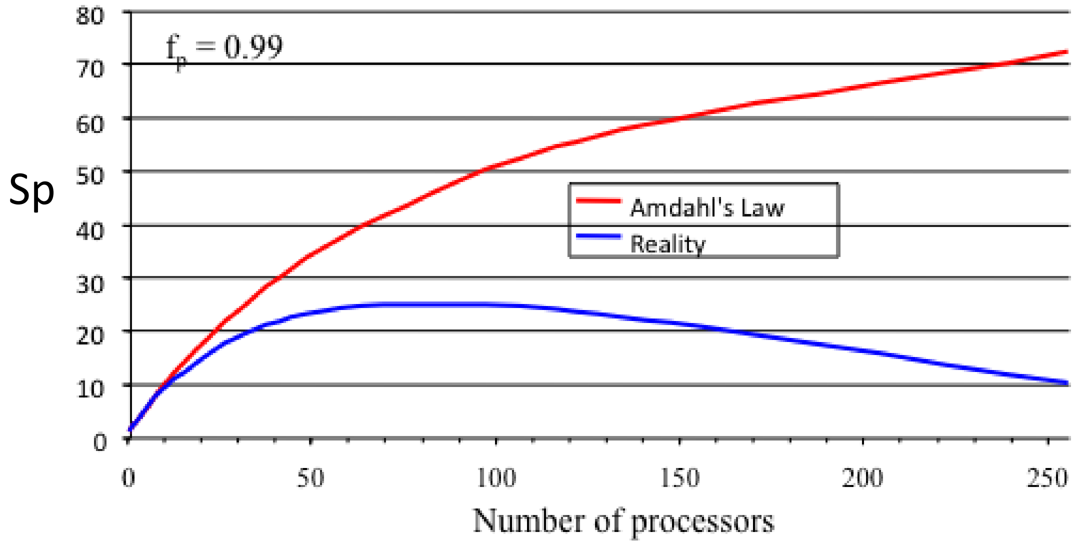
- f_s = serial fraction of code
- f_p = parallel fraction of code
- P = number of processors

Amdahl's Law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Practical Limits: Amdahl's Law vs. Reality

- In reality, the situation is even worse than predicted by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Cost of Communication
  - I/O



Gustafson's Law

- Effect of multiple processors on run time of a problem with a *fixed amount of parallel work per processor.*

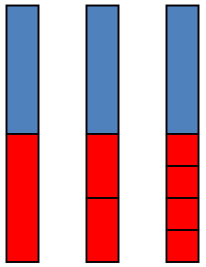
$$S_P \leq P - \alpha \square (P-1)$$

  - α is the fraction of non-parallelized code where the parallel work per processor is fixed (not the same as f_p from Amdahl's)
  - P is the number of processors

Comparison of Amdahl and Gustafson

Amdahl : fixed work

$f_p = 0.5$



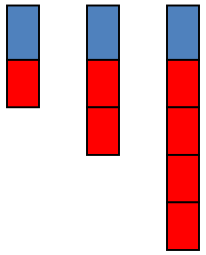
$$S \leq \frac{1}{f_s + f_p / N}$$

$$S_2 \leq \frac{1}{0.5 + 0.5/2} = 1.3$$

$$S_4 \leq \frac{1}{0.5 + 0.5/4} = 1.6$$

Gustafson : fixed work per processor

$\alpha = 0.5$



$$S_p \leq P - \alpha \square (P-1)$$

$$S_2 \leq 2 - 0.5(2-1) = 1.5$$

$$S_4 \leq 4 + 0.5(4-1) = 2.5$$

Scaling: Strong vs. Weak

- We want to know how quickly we can complete analysis on a particular problem size by increasing the core (processing element – PE) count
  - Amdahl's Law
  - Known as "strong scaling"

- We want to know if we can analyze a larger problem in approximately the same amount of time by increasing the PE count
  - Gustafson's Law
  - Known as "weak scaling"

PARALLEL SYSTEMS

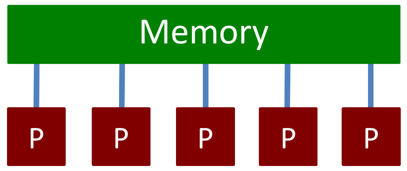
Classification #1: Instruction Streams

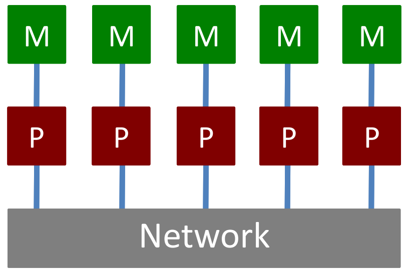
Hardware Classification

SISD	Single Instruction / Single Data	von Neumann 1-instruction-at-a-time
SIMD	Single Instruction / Multiple Data	Array processors, **vector** pipelines, SSE/AVX instructions
MIMD	Multiple Instruction / Multiple Data	Every processor has its own data and instruction stream
SPMD	Single Program / Multiple Data	Like MIMD, but all the same executable
SIMT	Single Instruction / Multiple Thread	Like SIMD, but not entirely synchronized: GPUs

Classification #2: Memory Model

Shared and Distributed Memory

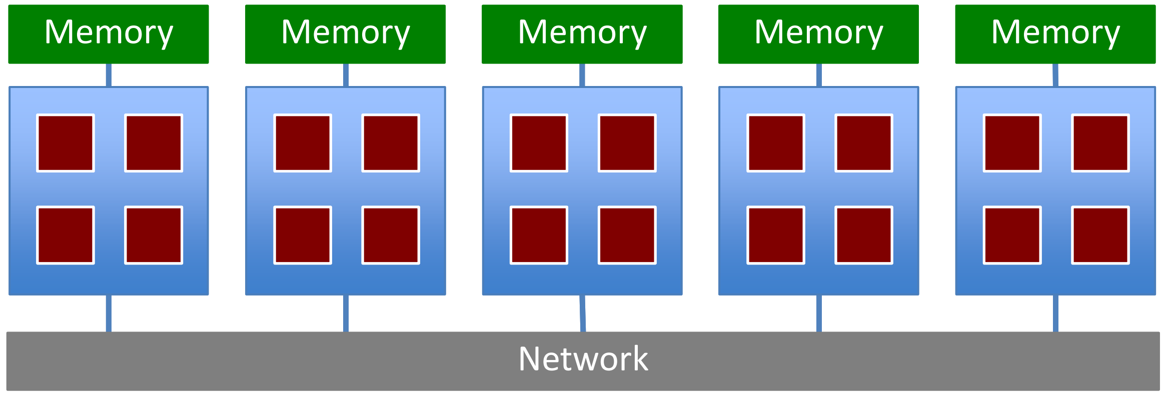




- All processors have access to a pool of shared memory
- Access times vary from CPU to CPU in NUMA systems
- Example: SGI Altix (SMP), multicore processors

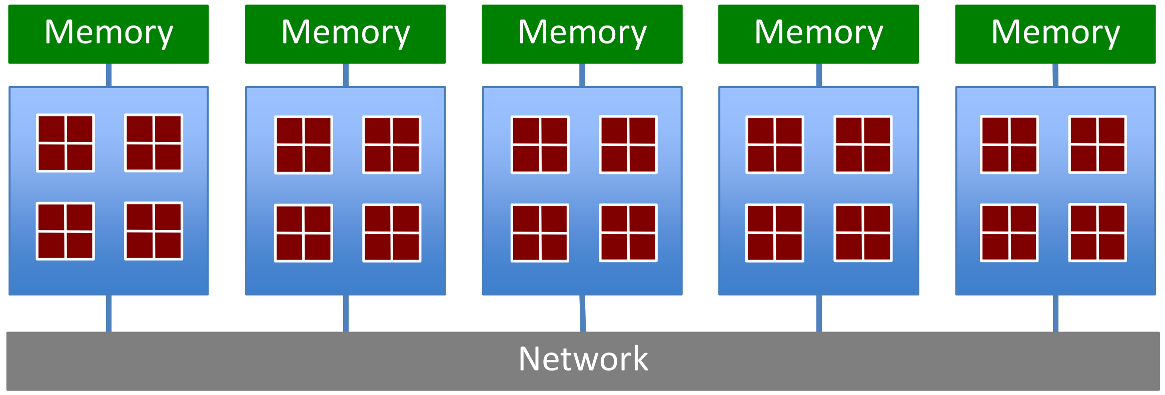
- Memory is local to each processor
- Data exchange by message passing over a network
- Example: Clusters with single-socket blades

Hybrid systems



- A limited number, N, of processors have access to a common pool of shared memory
- To use more than N processors requires data exchange over a network
- Example: Cluster with multi-socket blades

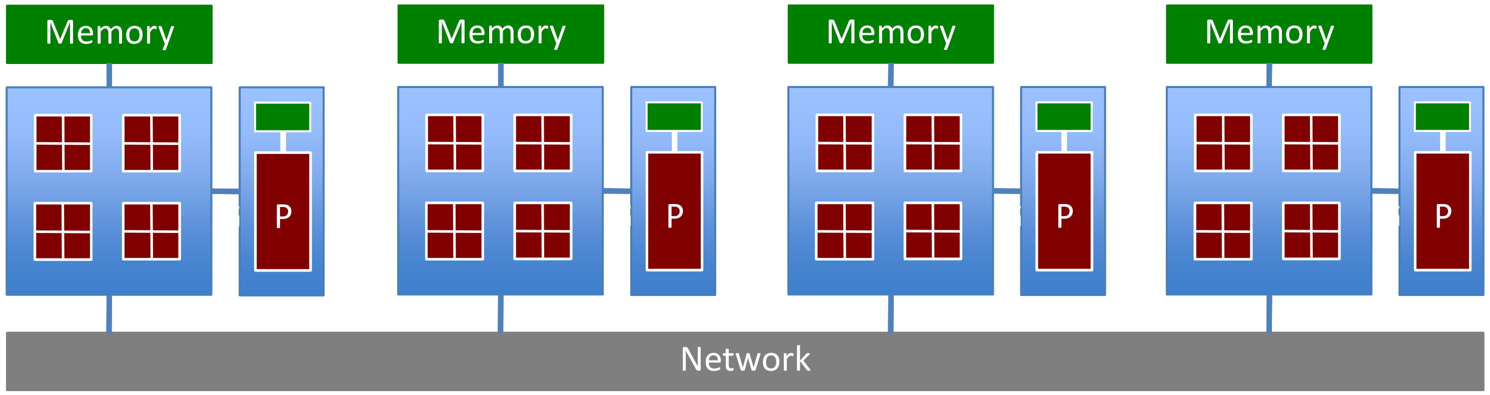
Multi-core Systems



- Extension of hybrid model

- Communication details increasingly complex
  - Cache access
  - Main memory access
  - Quick Path / Hyper Transport socket connections
  - Node to node connection via network

Coprocessor Systems

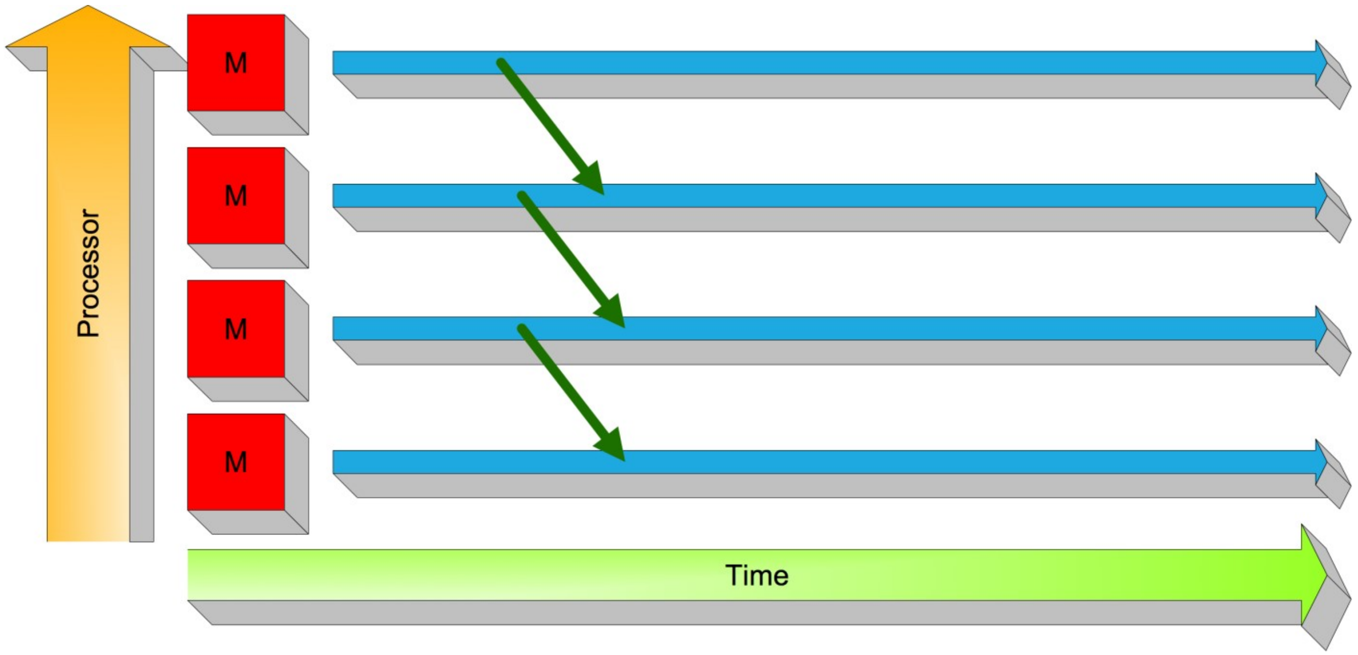


- Calculations made in both CPUs and coprocessors (GPU, MIC)
- Programmability is tricky: two different processor types
- Requires specific libraries and compilers (GPU: CUDA, OpenCL, MIC: OpenMP)

Classification #3: Process Dynamism

“Process-based” Parallelism

- MIMD & SPMD: one process per processor/core, lives for the life of the run
- Great for distributed memory: task creation and migration is hard.

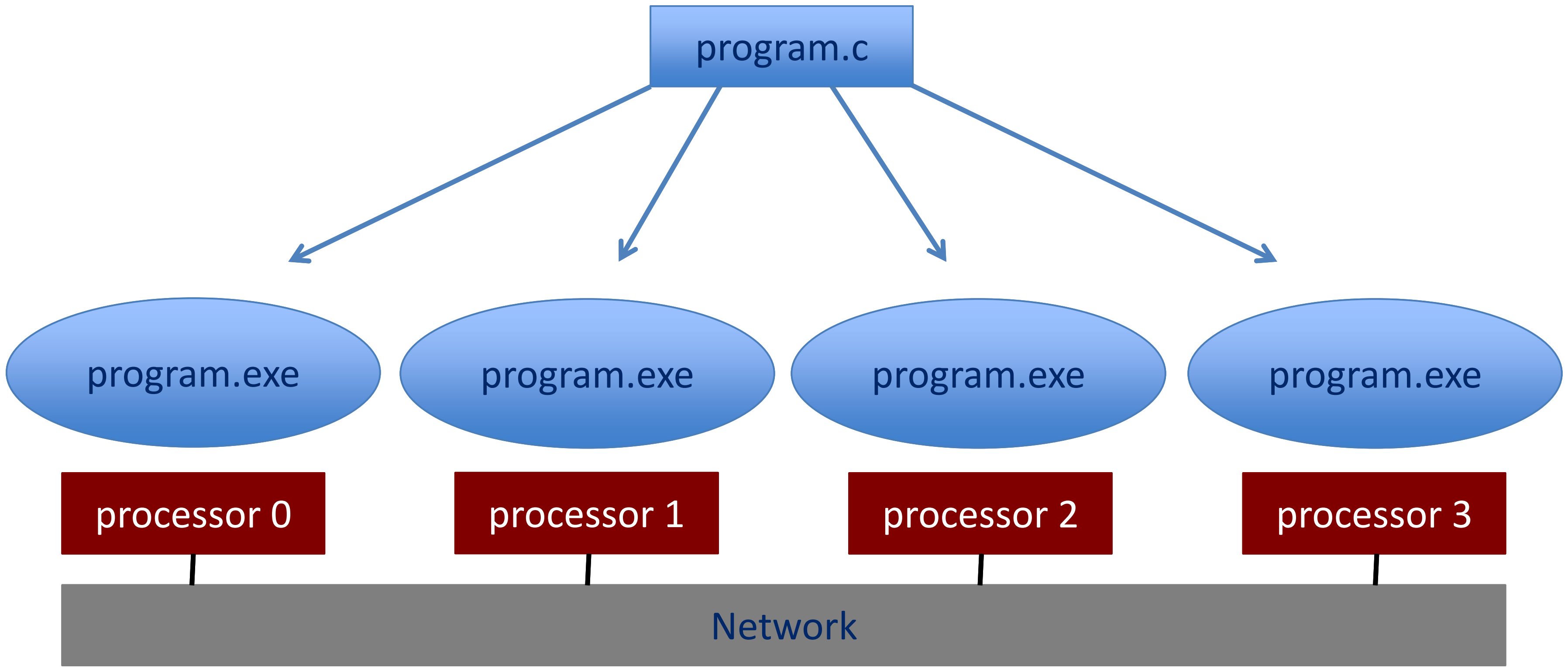


Single Program Multiple Data

- SPMD: dominant programming model for shared and distributed memory machines.
  - One source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code start simultaneously and communicate and sync with each other periodically

- MPMD: not often used (climate models, multi-physics engineering models)

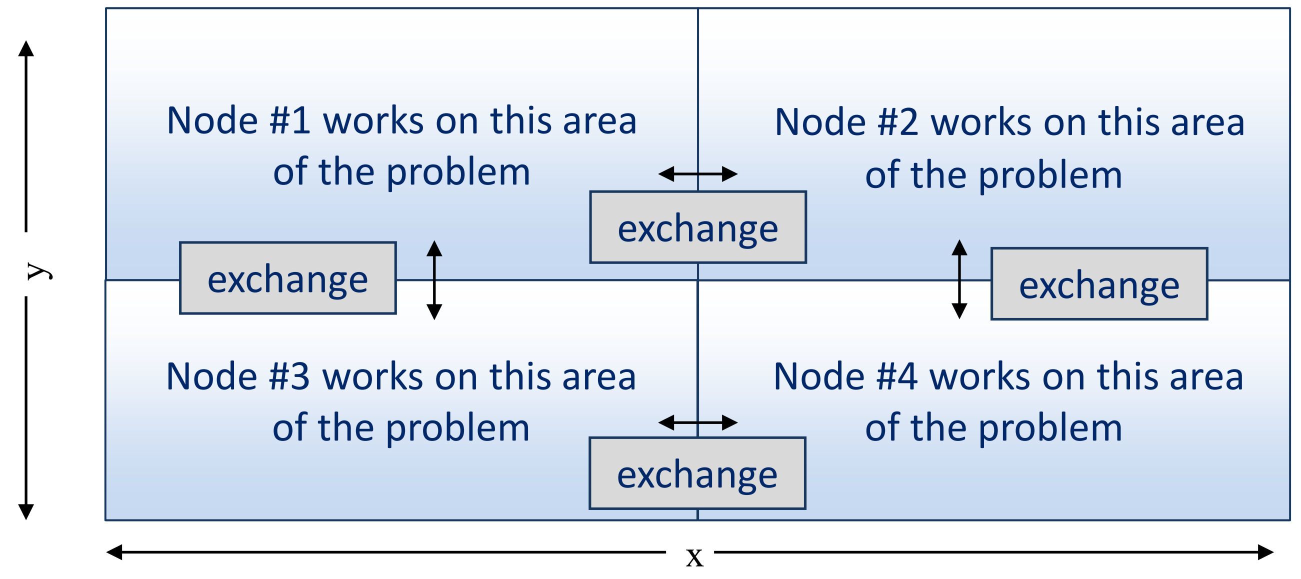
SPMD Model


program.c
program.exe
program.exe
program.exe
program.exe
processor 0
processor 1
processor 2
processor 3
Network

Data Decomposition

- For distributed memory systems, the spatial domain or particles are decomposed to the individual compute nodes
  - Each node works on its section of the problem.
  - Nodes can exchange information, the less the better.

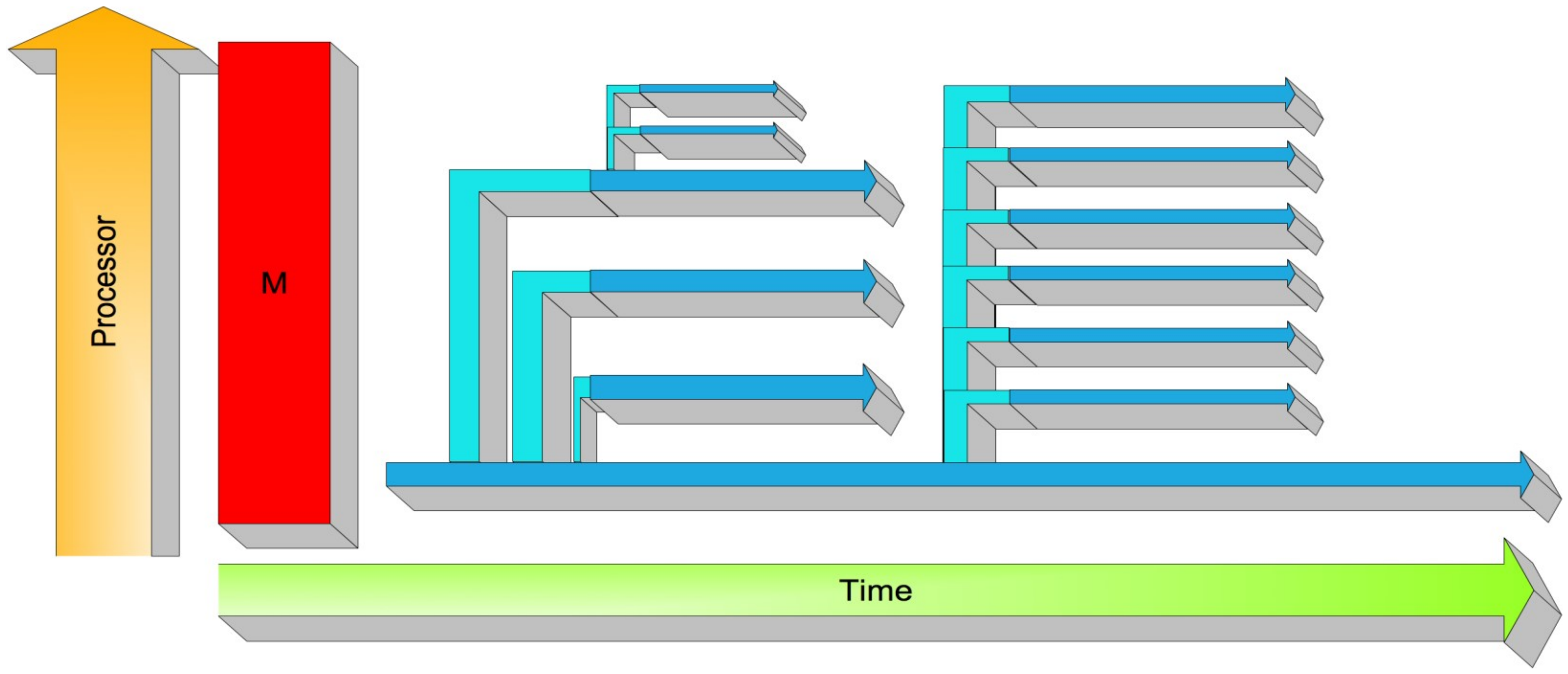
Grid of Problem to be solved



Node #1 works on this area of the problem
Node #2 works on this area of the problem
Node #3 works on this area of the problem
Node #4 works on this area of the problem
exchange
exchange
exchange
exchange
y
x

"Task-based" Parallelism

- Threading models: tasks can be created at will, placed on whatever processor/core is free
- Great on shared memory



Dynamic Thread Creation

- Old: pthreads
- Newer: Cilk+ (Intel), OpenMP (open standard), Java Threads

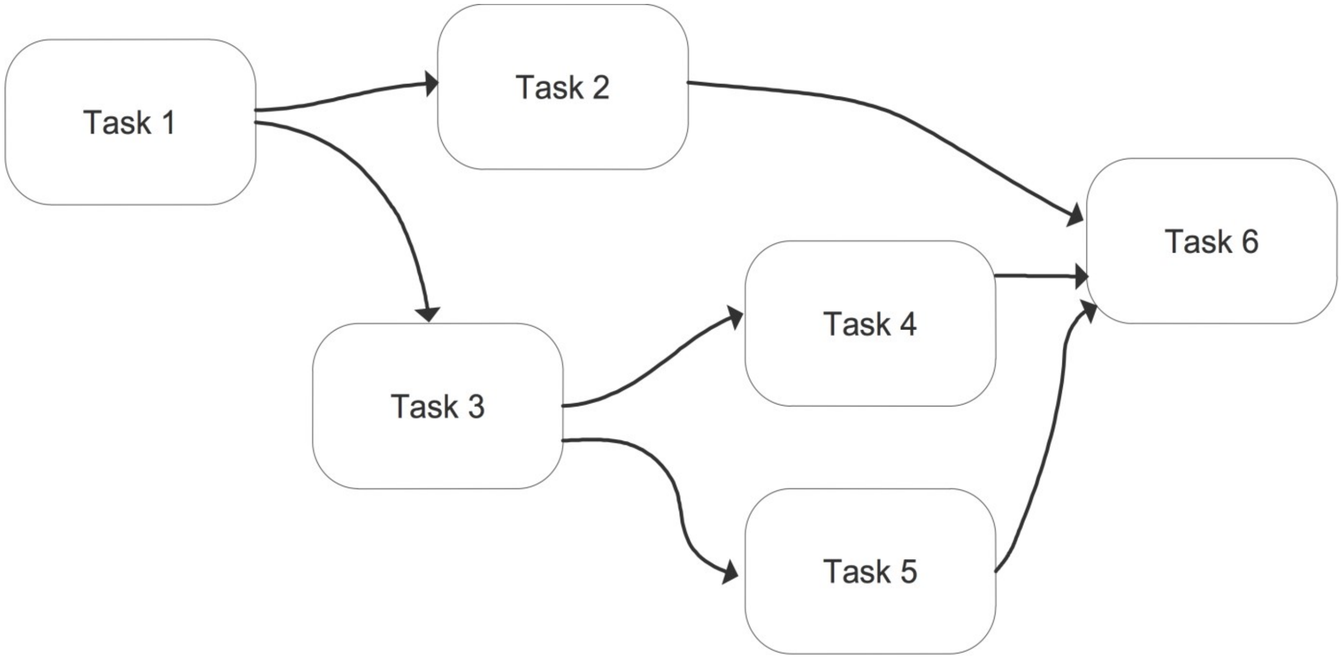
```
int sum = 0; /* Global!!! */
void adder(){ sum = sum+1; }

int main() {
  int i;
  pthread_t threads[NTHREADS];
  for (i=0; i<NTHREADS; i++)
    pthread_create
      (threads+i,NULL,&adder,NULL);
  for (i=0; i<NTHREADS; i++)
    pthread_join(threads[i],NULL);
```

```
cilk int fibonacci(int n){
  if (n<2) return 1;
  else {
    int rst=0;
    rst += spawn fibonacci(n-1);
    rst += spawn fibonacci(n-2);
    sync;
  return rst;
 }
```

General Tasks

- Can be realized with OpenMP tasks
- Dedicated task graph packages: CnC, Quark, SuperMatrix, TBB



OpenMP Example: Parallel Loop

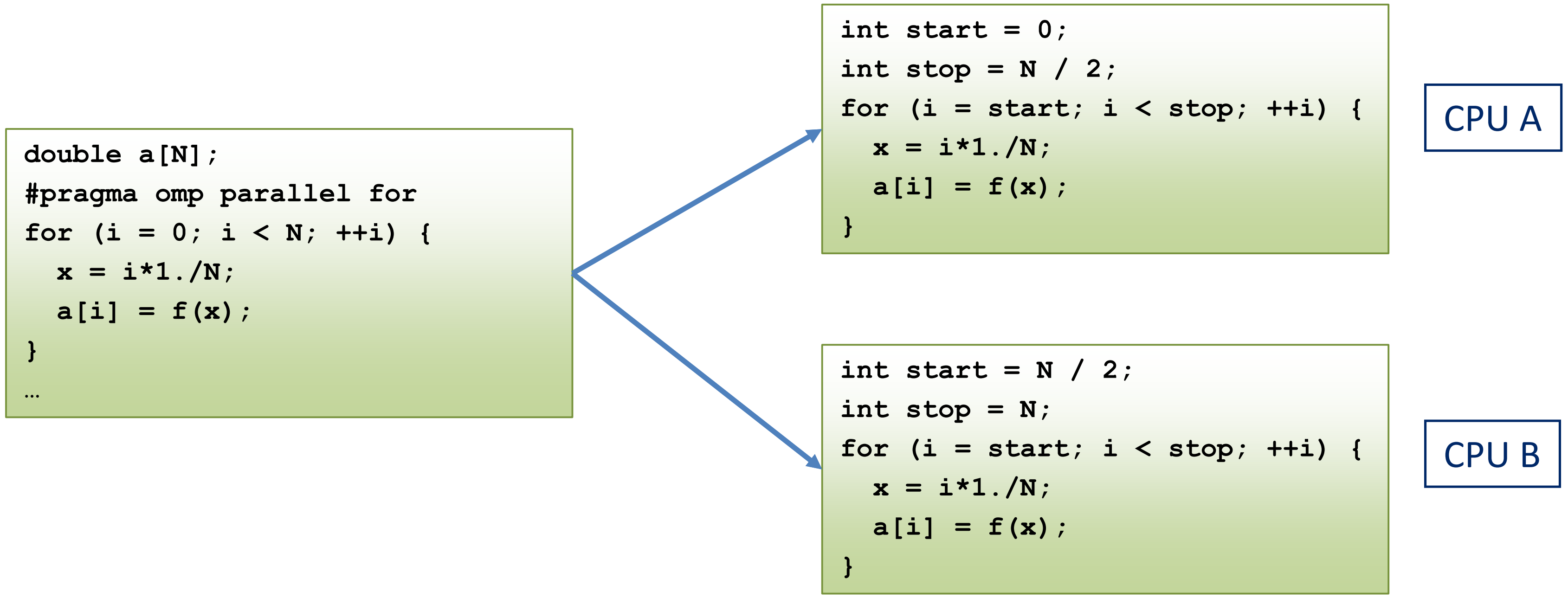
```
#pragma omp parallel for
for (i = 0; i < N; ++i)
   b[i] = a[i] + c[i];
```

- Easy parallelism: tasks correspond to loop iterations
- Actually, tasks are *groups* of iterations!
- The directive specifies that (1) a parallel region with K threads is about to be entered and (2) the loop immediately following should be executed in parallel.
- By default, the iteration space is divided into K contiguous chunks of approximately equal size … N / K.
- For codes that spend the majority of their time executing the content of simple loops, the PARALLEL FOR directive can result in significant parallel performance.
- OpenMP also has a general task mechanism though it's not this easy.

Different World Views

Shared Memory Data Access

- One code will run on 2 CPUs
- Program has array of data to be operated on by 2 CPUs so array is split into two parts.



```
double a[N];
#pragma omp parallel for
for (i = 0; i < N; ++i) {
  x = i*1./N;
  a[i] = f(x);
}
…
```

```
int start = 0;
int stop = N / 2;
for (i = start; i < stop; ++i) {
  x = i*1./N;
  a[i] = f(x);
}
```

```
int start = N / 2;
int stop = N;
for (i = start; i < stop; ++i) {
  x = i*1./N;
  a[i] = f(x);
}
```

Distributed Memory Data Access

- Since each CPU has local address space: local indexing only
- Must translate between local and global index space.

CPU A

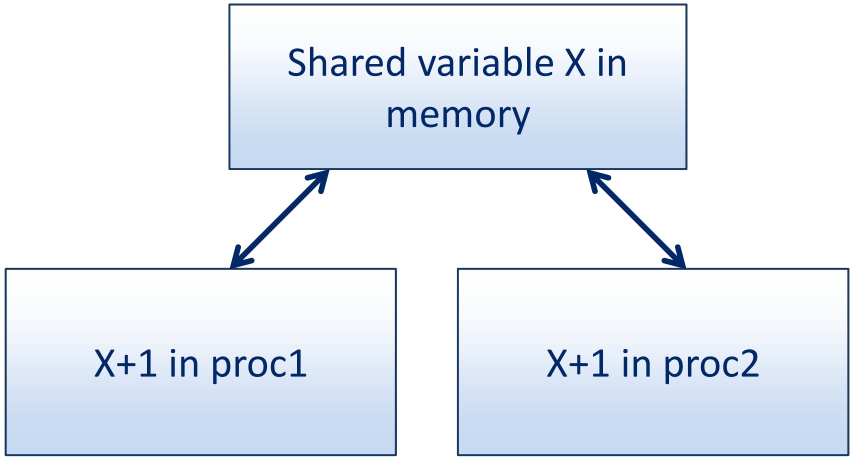
```
int start = 0;
int stop = N / 2;
double a[N/2];
for (i = start; i < stop; ++i) {
  x = i*1.0/N;
  a[i-start] = f(x);
}
```

CPU B

```
int start = N / 2;
int stop = N;
double a[N/2];
for (i = start; i < stop; ++i) {
  x = i*1.0/N;
  a[i-start] = f(x);
}
```

Accessing Shared Variables

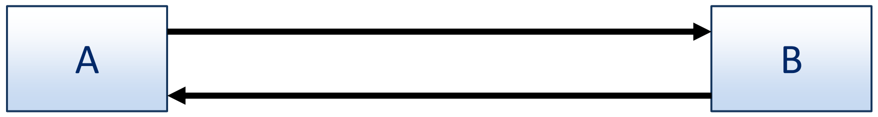
- If multiple processors want to write to a shared variable at the same time, there could be conflicts:
  - Process 1 and 2
  - read X
  - compute X+1
  - write X
  - Is the answer X, X+1, X+2?
  - Known as *race* condition.



- Programmer, language, and/or architecture must provide ways of resolving conflicts

Message Passing Communication

- Processes in message passing programs communicate by passing messages



- Basic message passing primitives
  - Send (parameters list)
  - Receive (parameters list)
  - Parameters depend on the library used

MPI: Sends and Receives

- Message Passing Interface (MPI) programs must send and receive data between the processors (communication)

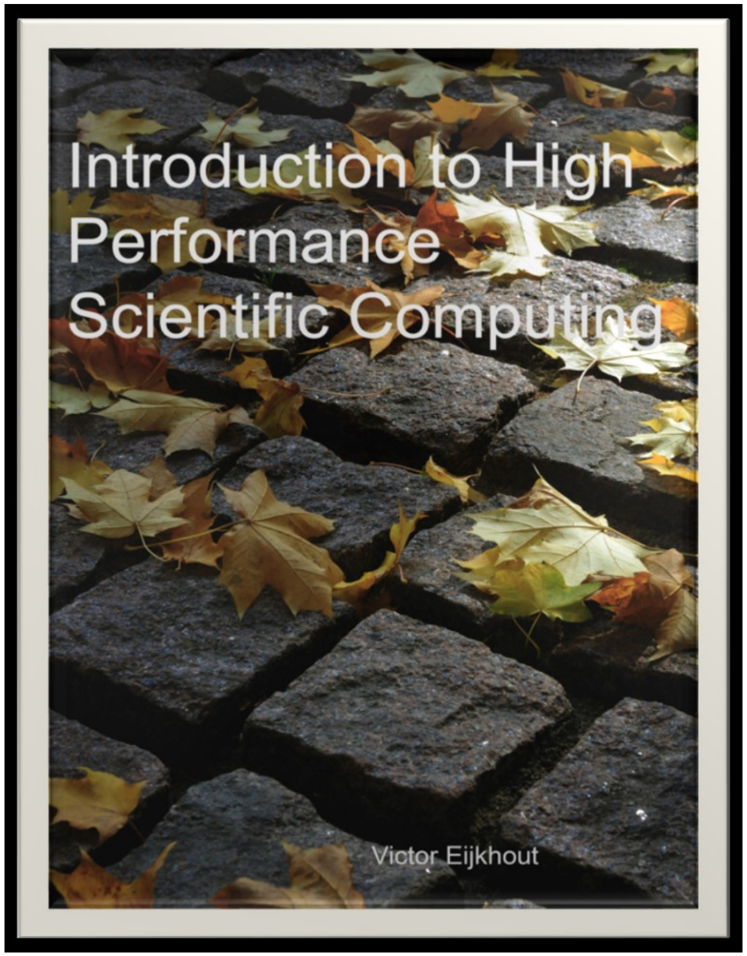
- The most basic calls in MPI (besides the three initialization and one finalization calls) are:
  - MPI_Send
  - MPI_Recv

- These calls are blocking: the source processor issuing the send/receive cannot move to the next statement until the target processor issues the matching receive/send.

Final Thoughts

- Systems with multiple shared memory processors are very common for reasons of economics and engineering.
- Going forward, this means that the most practical programming paradigms to learn are
  - Pure MPI
  - OpenMP + MPI (or Threads + MPI) … MPI+X

Further reading



- General page: http://www.tacc.utexas.edu/~eijkhout/istc/istc.html
- Direct download: http://tinyurl.com/EijkhoutHPC

- Content presented here is based on content from "Parallel Computing for Science and Engineering course materials by The Texas Advanced Computing Center, 2013. Available under a Creative Commons Attribution Non-Commercial 3.0 Unported License"

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License