Parallel Computing for
Science & Engineering
SSC 374/394c

Introduction to Parallel Computing

Instructors:
Victor Eijkhout, Research Scientist, TACC
Kent, Milfeld, Research Associate, TACC

Outline

- Overview
- Theoretical background
- Parallel computing systems
- Parallel programming models
- MPI/OpenMP examples

OVERVIEW

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

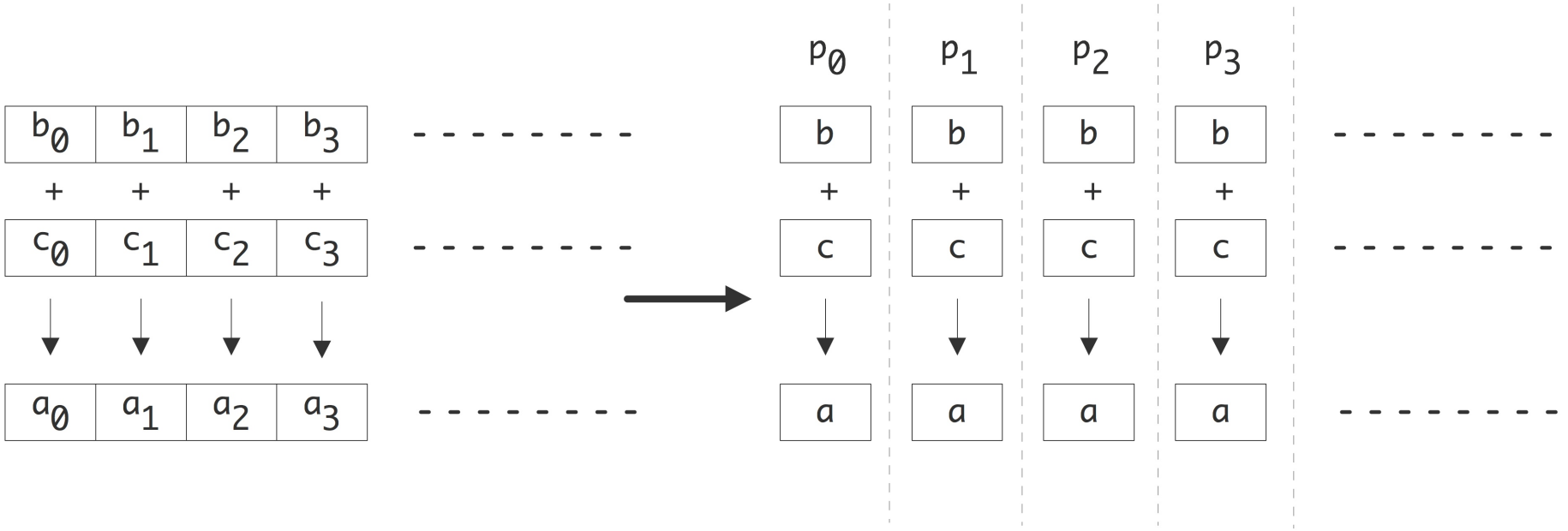
The basic idea

- 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]
```

```
a = b+c
```

Idealized version:
every process has
one array element


p0
p1
p2
p3
b0
b1
b2
b3
c0
c1
c2
c3
a0
a1
a2
a3
b
c
a

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]
```

```
a = b+c
```

Idealized version:
every process has
one array element

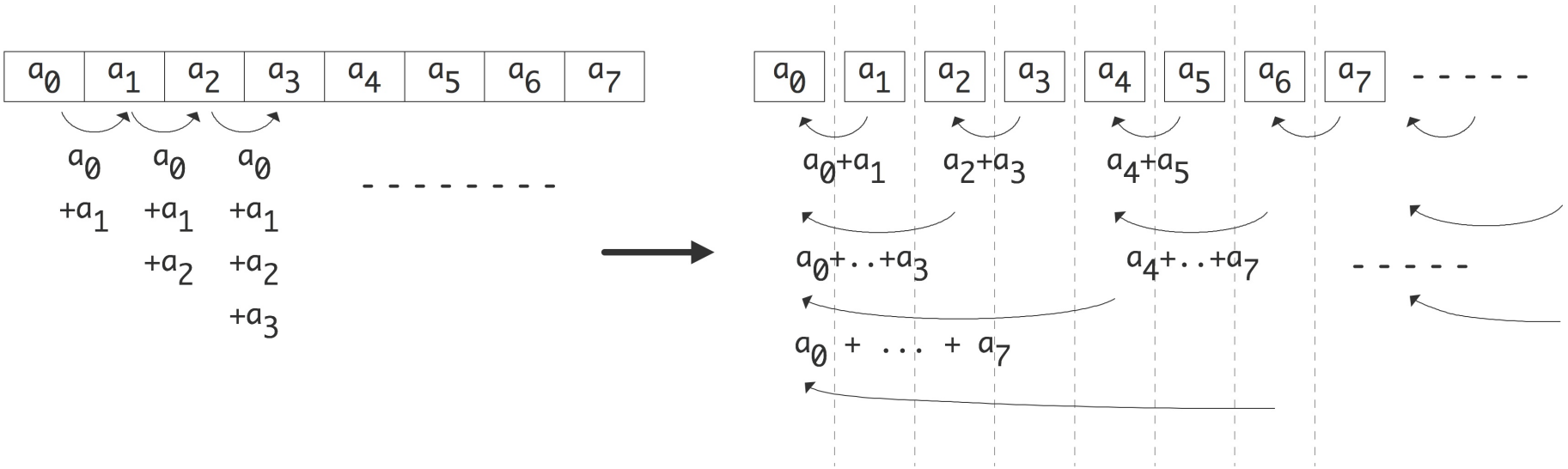
```
for (i=my_low; i<my_high; i++)
  a[i] = b[i]+c[i]
```

Slightly less ideal:
each processor has
part of the array

The basic idea (cont'd)

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does it always become *t/p* on *p* processors (*p*<=*n*)?

```
s = sum( x[i], i=0,n-1 )
```


a0 a1 a2 a3 a4 a5 a6 a7
a0 +a1
a0 +a1 +a2
a0 +a1 +a2 +a3
- - - - - - - -
a0 a1 a2 a3 a4 a5 a6 a7 - - - - -
a0+a1
a2+a3
a4+a5
a0+..+a3
a4+..+a7
- - - - -
a0 + ... + a7

The basic idea (cont'd)

- Spread operations over many processors
- If *n* operations take time *t* on 1 processor,
- Does it always become *t/p* on *p* processors (*p*<=*n*)?

```
s = sum( x[i], i=0,n-1 )
```

```
for (p=0; p<n/2; p++)
  x[2p,0] = x[2p]+x[2p+1]
for (p=0; p<n/4; p++)
  x[4p,1] = x[4p]+x[4p+2]
for ( .. p<n/8 .. )

Et cetera
```

Conclusion: n operations can be done with n/2 processors, in total time $\log_2 n$

Theoretical question: can addition be done faster?

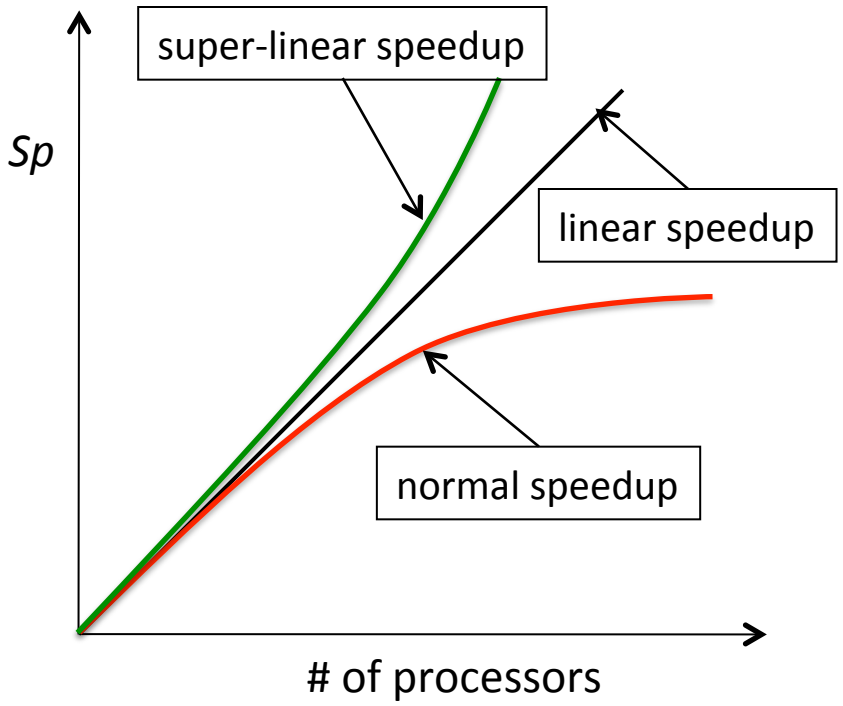
Practical question: can we even do this?

THEORETICAL BACKGROUND

Speedup & Parallel Efficiency

- Speedup: $S_p = \frac{T_s}{T_p}$
  - *p* = # of processors
  - *Ts* = execution time of the sequential algorithm
  - *Tp* = execution time of the parallel algorithm with *p* processors
  - *Sp= P* (linear speedup: ideal)
- Parallel efficiency

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$



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

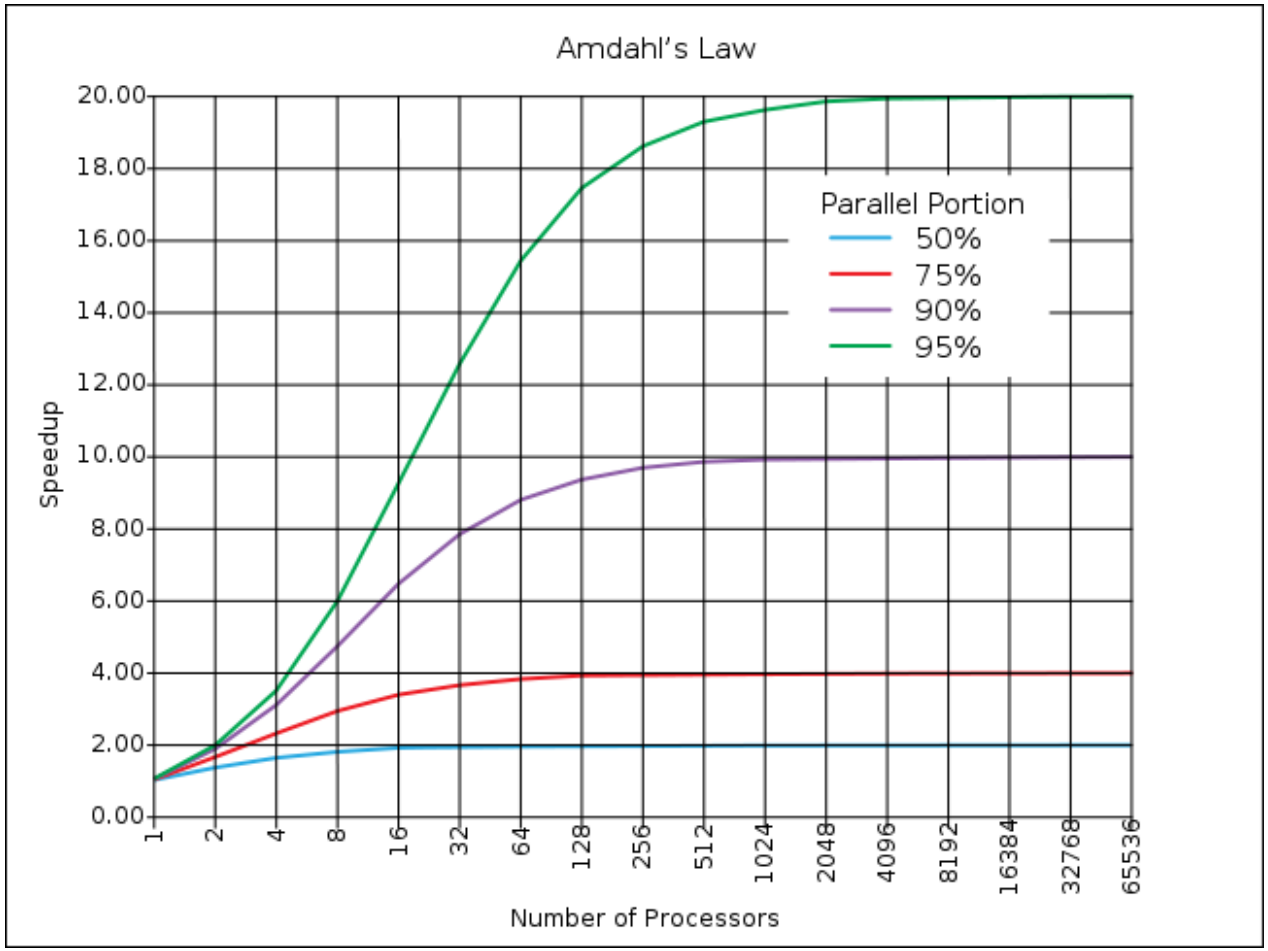
- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness
- Amdahl's Law states this formally
  - Effect of multiple processors on speed up

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

where

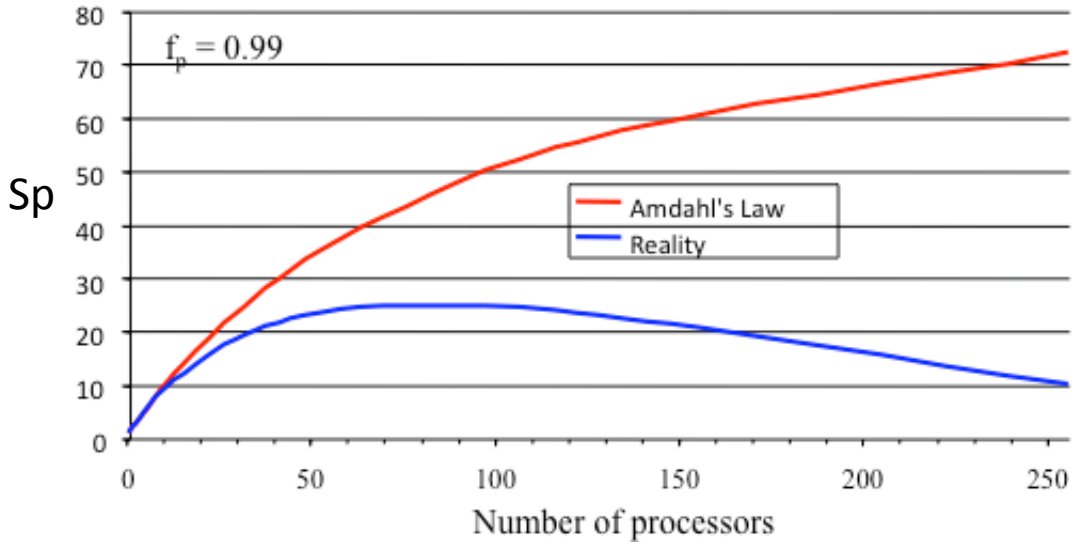
- f_s = serial fraction of code
- f_p = parallel fraction of code
- P = number of processors

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 Communications
  - I/O



Gustafson's Law

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

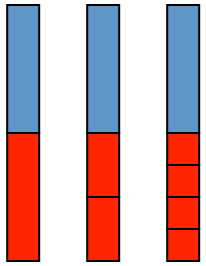
$$S_P \leq P - \alpha \cdot (P - 1)$$

  - *a* 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

$f_p = 0.5$



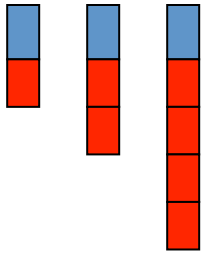
$$S \le \frac{1}{f_s + f_p / N}$$

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

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

Gustafson : fixed work per processor

$\alpha = 0.5$



$$S_p \le P - \alpha \cdot (P - 1)$$

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

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

Scaling: Strong vs. Weak

- We want to know how quickly we can complete analysis on a particular data set by increasing the PE count
  - Amdahl's Law
  - Known as "strong scaling"
- We want to know if we can analyze more data 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

“Old school” hardware classification

SISD old-fashioned (von Neumann) one-instruction-at-a-time

SIMD array processors, vector pipelines, SSE/AVX instructions

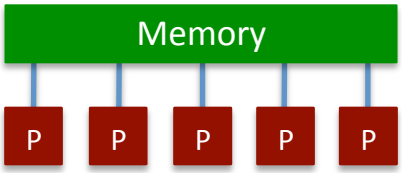
MIMD every processor its own data and instruction stream

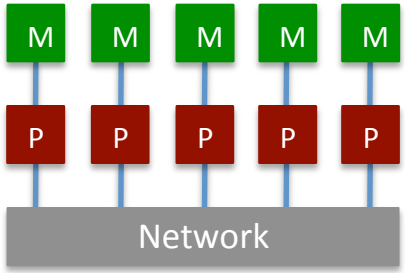
SPMD like MIMD, but all the same executable

SIMT like SIMD, but not entirely synchronized: GPUs

Classification #2: memory model

Shared and distributed memory

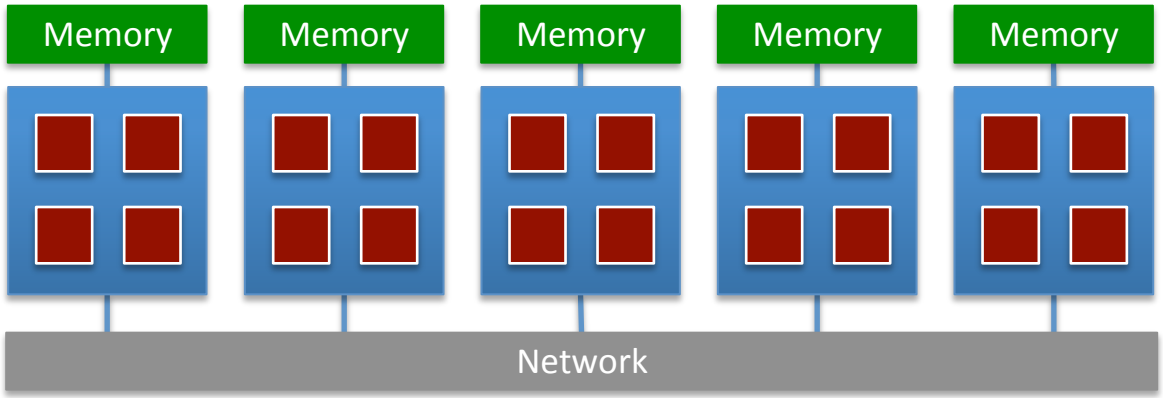




- All processors have access to a pool of shared memory
- Access times vary from CPU to CPU in NUMA systems
- Example: SGI Altix (SMP), multicore processors

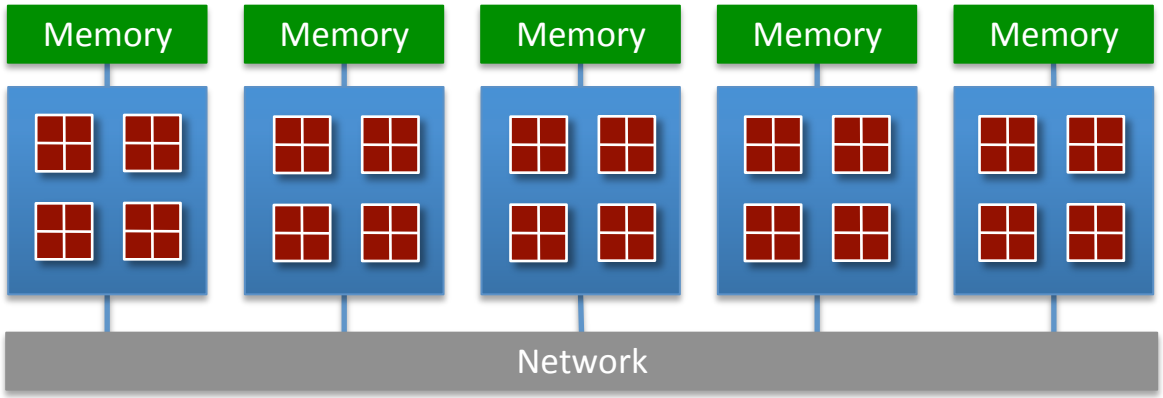
- Memory is local to each processor
- Data exchange by message passing over a network
- Example: Clusters with single-socket blades

Hybrid systems



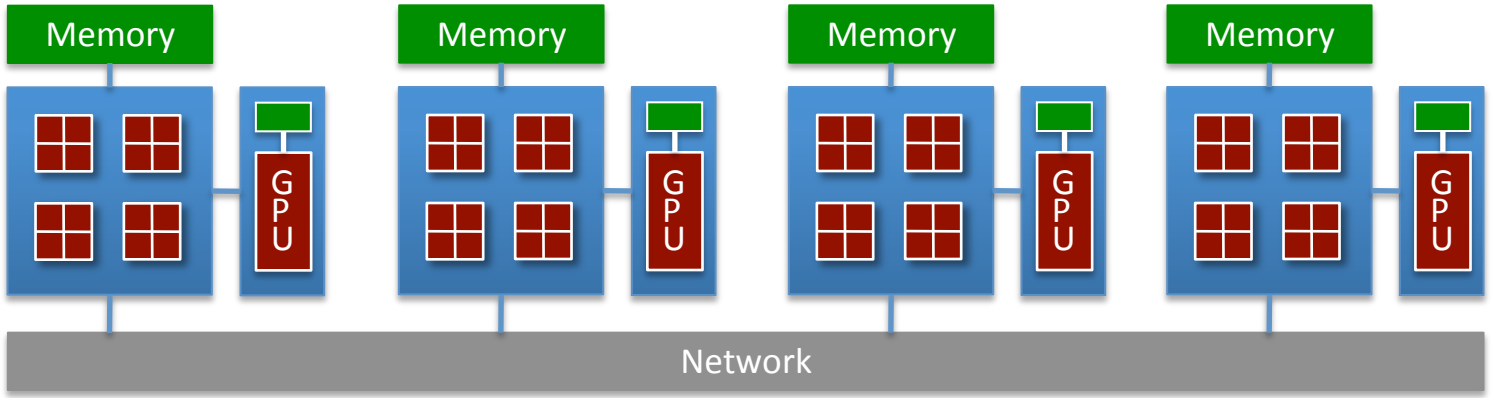
- A limited number, N, of processors have access to a common pool of shared memory
- To use more than N processors requires data exchange over a network
- Example: Cluster with multi-socket blades

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

Co-processor Systems

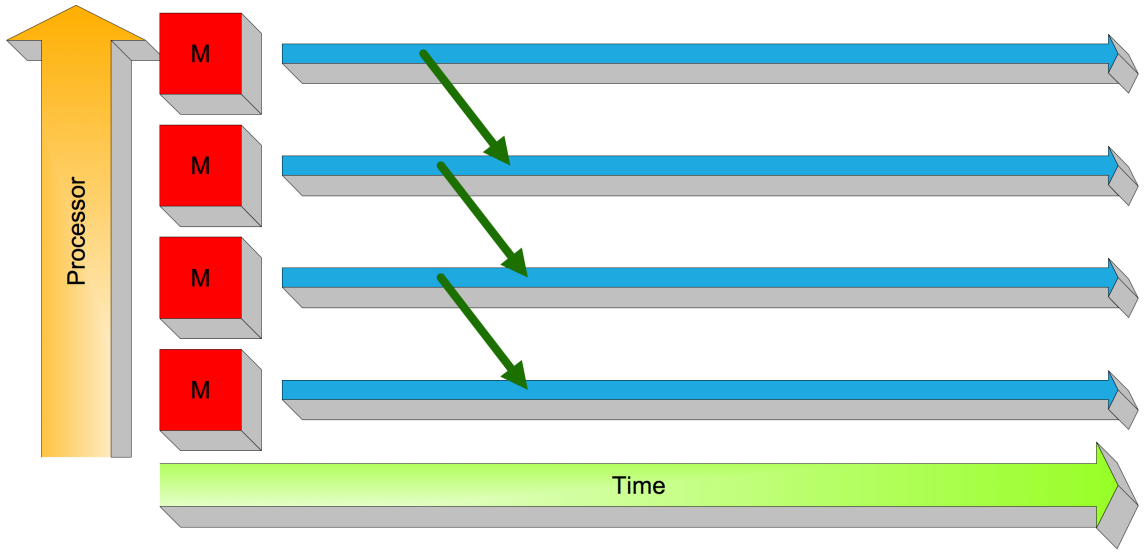


- Calculations made in both CPUs and co-processors (GPU, MIC)
- Programmability is tricky: two different processor types
- Requires specific libraries and compilers (GPU: CUDA, OpenCL, MIC: OpenMP)

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.

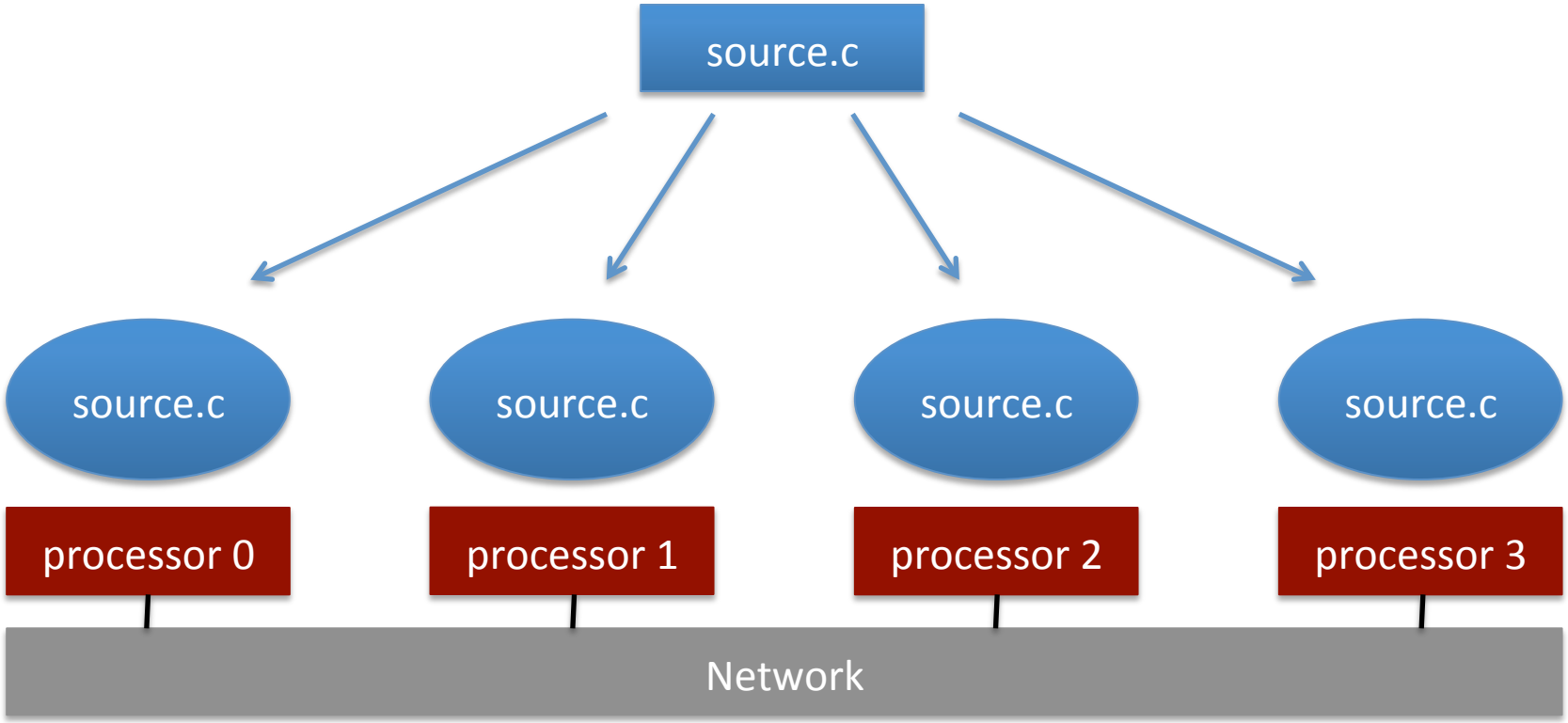


Single Program Multiple Data

- 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

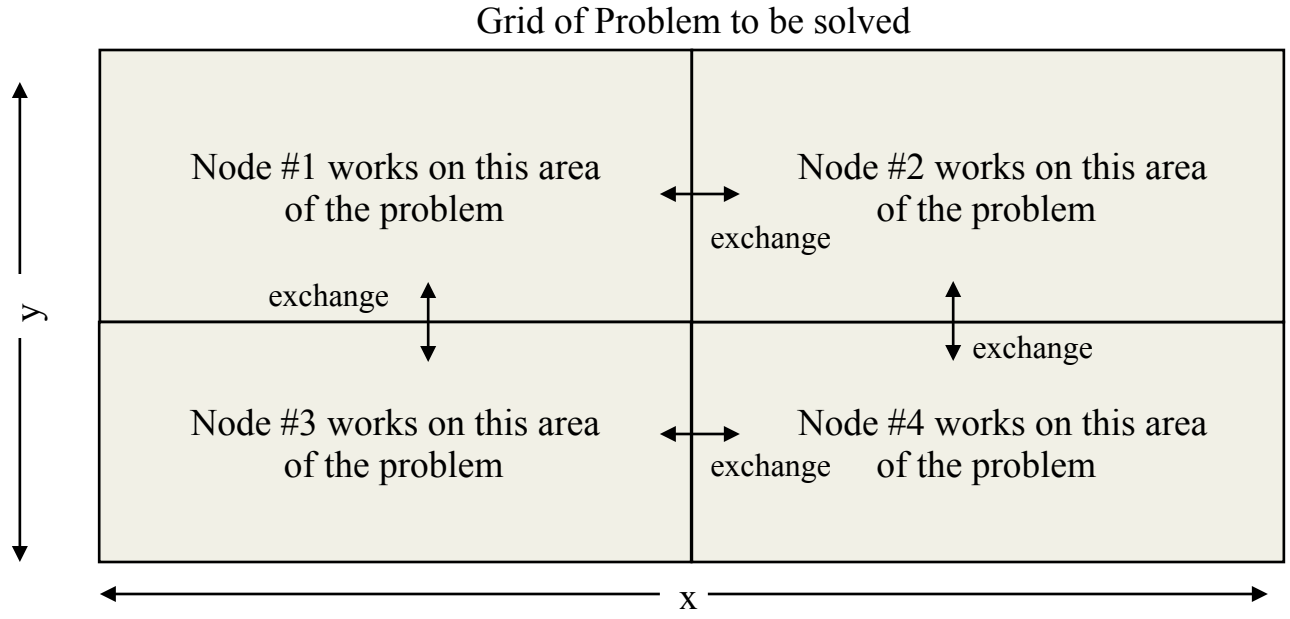
- MPMD: not often used (climate models), kinda tricky

SPMD Model


source.c
source.c
source.c
source.c
source.c
processor 0
processor 1
processor 2
processor 3
Network

Data Decomposition

- For distributed memory systems, the 'whole' grid or sum of particles is decomposed to the individual nodes
  - Each node works on its section of the problem
  - Nodes can exchange information



Typical Data Decomposition

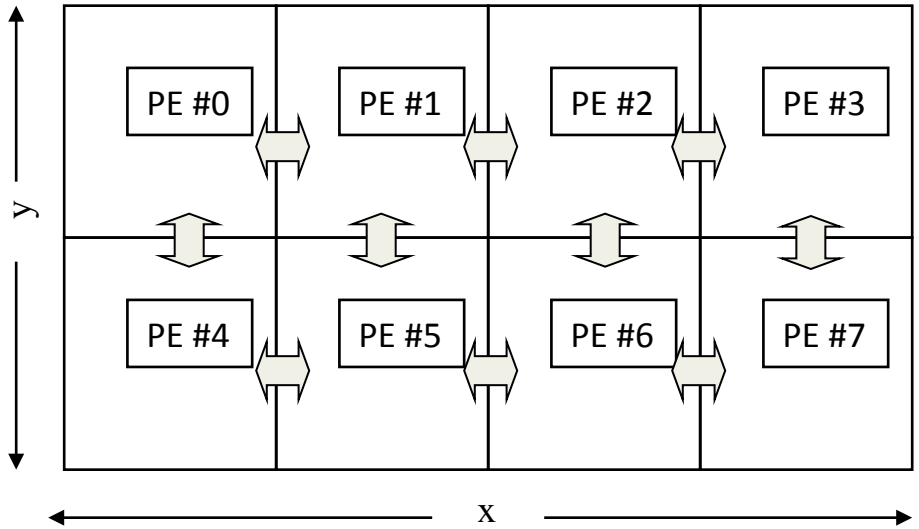
- Example: integrate 2-D propagation problem:

Starting partial differential equation:

$$\frac{\partial \Psi}{\partial t} = D \cdot \frac{\partial^2 \Psi}{\partial x^2} + B \cdot \frac{\partial^2 \Psi}{\partial y^2}$$

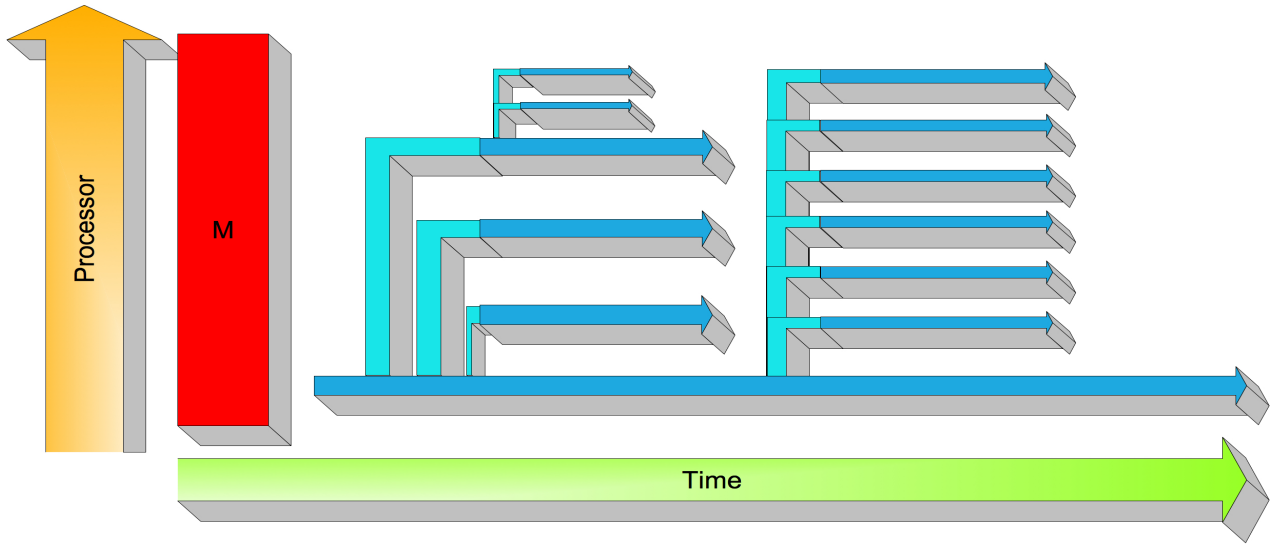
Finite Difference Approximation:

$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} = D \cdot \frac{f_{i+1,j}^{n} - 2f_{i,j}^{n} + f_{i-1,j}^{n}}{\Delta x^2} + B \cdot \frac{f_{i,j+1}^{n} - 2f_{i,j}^{n} + f_{i,j-1}^{n}}{\Delta y^2}$$



“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)

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

OpenMP Example: Parallel Loop

```
!$OMP PARALLEL DO
  do i=1,128
    b(i) = a(i) + c(i)
  end do
!$OMP END PARALLEL DO
```

- Easy parallelism: tasks correspond to loop iterations
- (Actually, tasks are *groups* of iterations)
- The first directive specifies that the loop immediately following should be executed in parallel. The second directive specifies the end of the parallel section (optional).
- For codes that spend the majority of their time executing the content of simple loops, the PARALLEL DO directive can result in significant parallel performance.
- OpenMP also has a general task mechanism

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.

```
program:
…
if CPU=a then
   low_limit=1
   upper_limit=50
elseif CPU=b then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
            upper_limit
  x = i*1./upper_limit
  A(I) = f(x)
end do
...
end program
```

CPU A

```
program:
…
low_limit=1
upper_limit=50
do I= low_limit,
          upper_limit
  x = i*1./upper_limit
  A(I) = f(x)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit,
          upper_limit
  x = i*1./upper_limit
  A(I) = f(x)
end do
…
end program
```

Distributed memory data access

- Since each CPU has local address space: local indexing only

CPU A

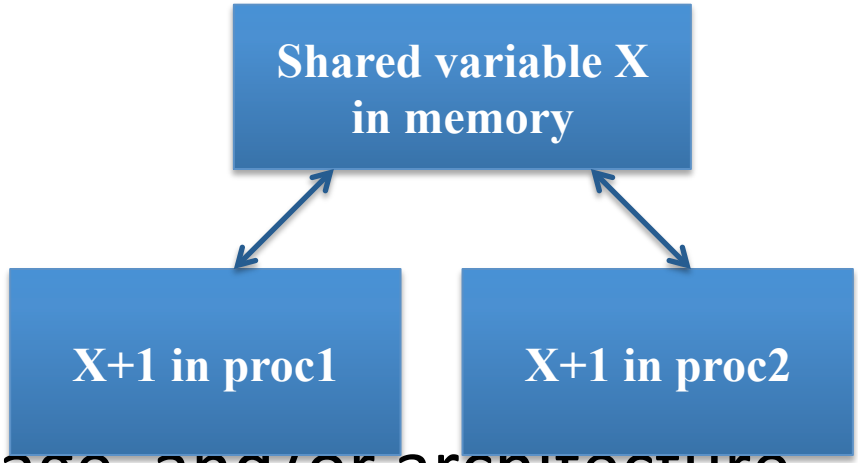
```
program:
…
low_limit=1
upper_limit=50
do I= low_limit, upper_limit
   x = i*1./upper_limit
   A(I-low_limit+1) = f(x)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit, upper_limit
   x = i*1./upper_limit
   A(I-low_limit+1) = f(x)
end do
…
end program
```

Accessing Shared Variables

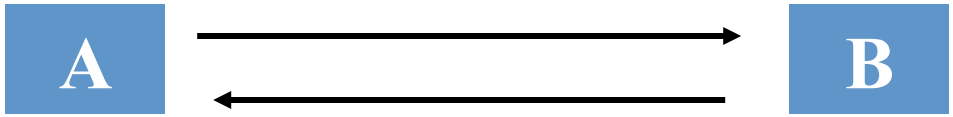
- If multiple processors want to write to a shared variable at the same time, there could be conflicts :
  - Process 1 and 2
  - read X
  - compute X+1
  - write X



- 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 (parameter list)
- Parameters depend on the library used

MPI: Sends and Receives

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

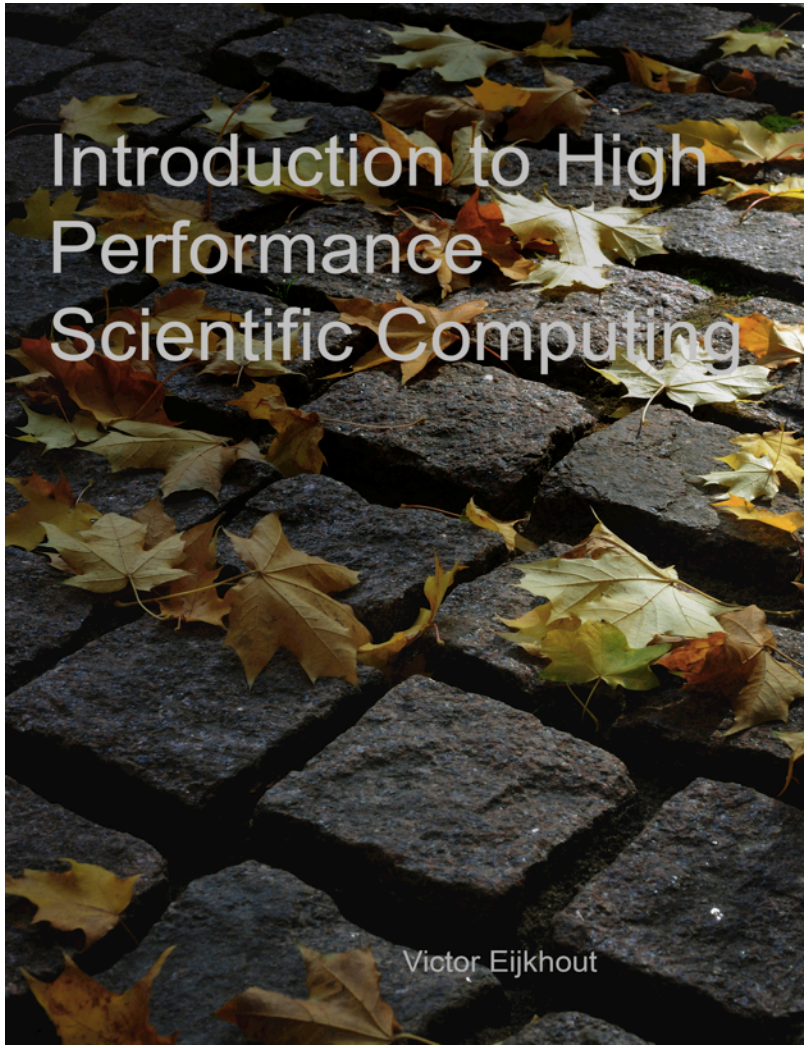
- The most basic calls in MPI (besides the three initialization and one finalization calls) are:
  - MPI_Send
  - MPI_Recv

- 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 nodes are becoming common for reasons of economics and engineering.
- Going forward, this means that the most practical programming paradigms to learn are
  - Pure MPI , and
  - OpenMP + MPI

Further reading



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

Title

- Text

```
code
```