Parallel Computing on Clusters and SMP Machines: A Comparison

SFU, Jun. 2000

1. Introduction:
   - Methods of communication between processes in parallel computing
   → Why use MPI?

2. Beowulf versus SMP
   distributed memory ↔ shared memory
   Amdahl's law

3. Case Studies:
   - Monte-Carlo Simulations
   - Partial Differential Equations
     → domain decomposition
   - Fast Fourier Transforms

4. Conclusion
   - What kind of parallel programs can you run on a Beowulf?
   - For which programs do you need an SMP machine?
   - Future developments

Communication Between Processes

Central problem in parallel computing:

How can results that were calculated on one processor be accessed by another processor?

$\Rightarrow$ transfer of data required

different methods available:

OpenMP:
- compiler directives inserted into the program
- works exclusively for shared-memory architecture (SMP)

HPF (High-Performance Fortran):
- Fortran statements
- designed for shared memory

MPI (Message Passing Interface):
- Set of standardized library routines
- works for distributed and shared memory architectures
- supports clusters of SMPs

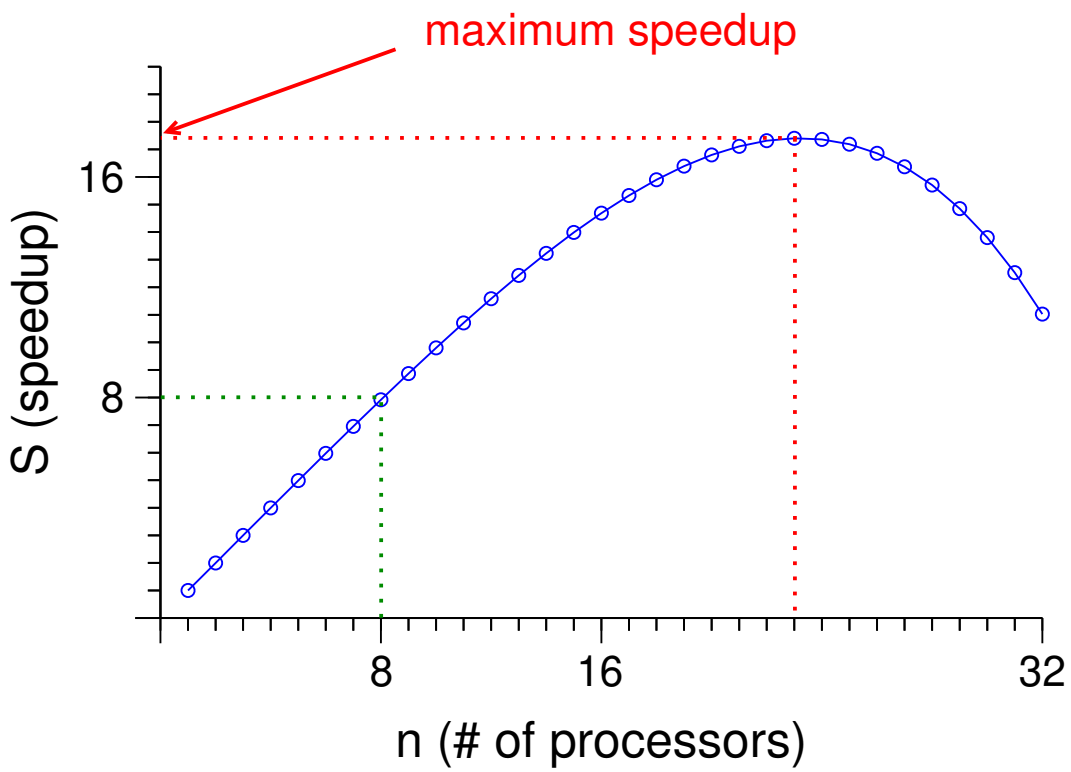
PVM (Parallel Virtual Machine):
- Similar to MPI (predecessor) [obsolete?]
- not a standard

do not use "autoparalellization"

Beowulf vs. SMP

speedup of a parallel program: $S = T_1/T_n$
(T_n CPU time of the program using n processors).

Typical behaviour:



maximum depends on the method of communication (shared memory, ethernet, ...), problem

SMP: Symmetric Multi-Processor machine

shared memory → all processors can access all memory, communication is fast.

Beowulf Cluster: pile of PCs, each with own set of memory chips, memory is distributed

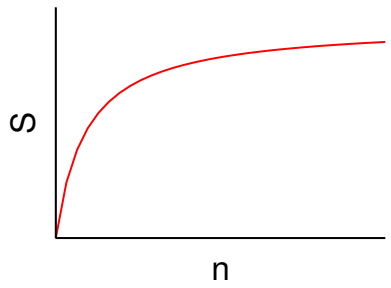
→ data must be transferred to other processors over (usually) switched fast ethernet (100 Mbit/s) → communication is slow.

Amdahl's Law

Estimate for the speedup: $T_n = t_s + t_p/n$

with: t_s cpu time for serial part of the program
t_p cpu time for part executed in parallel

$$S = \frac{T_1}{T_n} = \frac{t_s + t_p}{t_s + t_p/n} \xrightarrow{n\to\infty} \frac{t_s + t_p}{t_s}$$

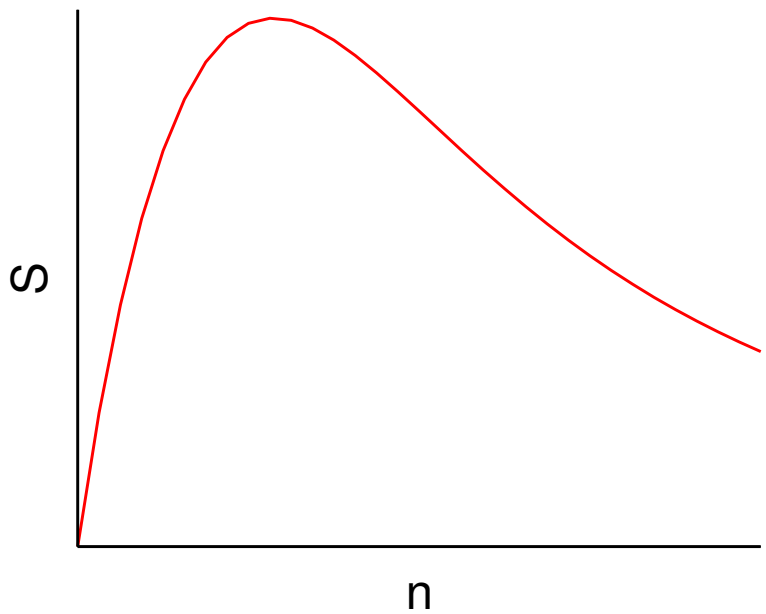


Amdahl's law

more realistic: include cost of communication

$$t_{\text{com}} = t_{cs} + t_{cn}n + t_{cp}n(n-1)$$

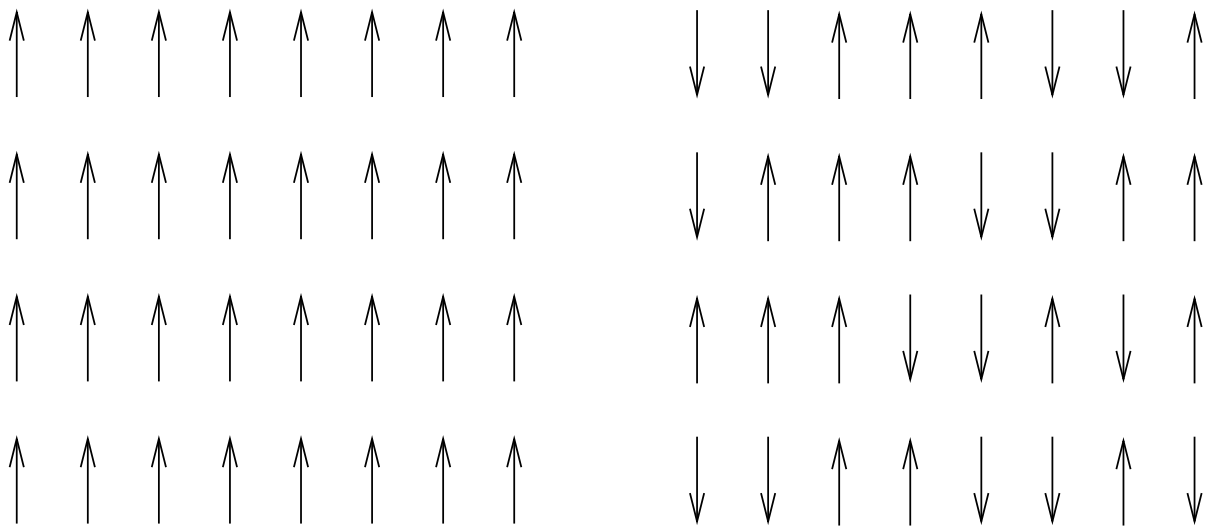
$$S = \frac{T_1}{T_n} = \frac{t_s + t_p}{t_s + t_p/n + t_{cs} + t_{cn}n + t_{cp}n(n-1)}$$



Monte-Carlo Simulations

Example:
Ising model (describes a magnetic material)

low T: $M = 1$ high T: $M = 0$



What is the configuration for arbitrary T?

1. start with some initial configuration
2. select particle
3. flip particle according to some probability
   $\rightarrow$ random process
4. iterate 2, 3
5. measure total magnetization M and add to running average: $M_{\text{ave}} = \sum M$.
6. iterate 2 - 5
7. print M_{ave}/(no. of measurements)

Typical layout of a Monte-Carlo program:

- black: serial code run on a single processor
- green: code run in parallel on all processors
- red: communication between processors
- blue: comment

Serial program

```
program monte_carlo
...
<read input data>
...
for mc=1 to mc=nrun do
   <generate initial configuration>
   <change configuration: random process>
   <measure quantities of interest>
   <update running average of these quantities>
end do
...
<print results>
...
```

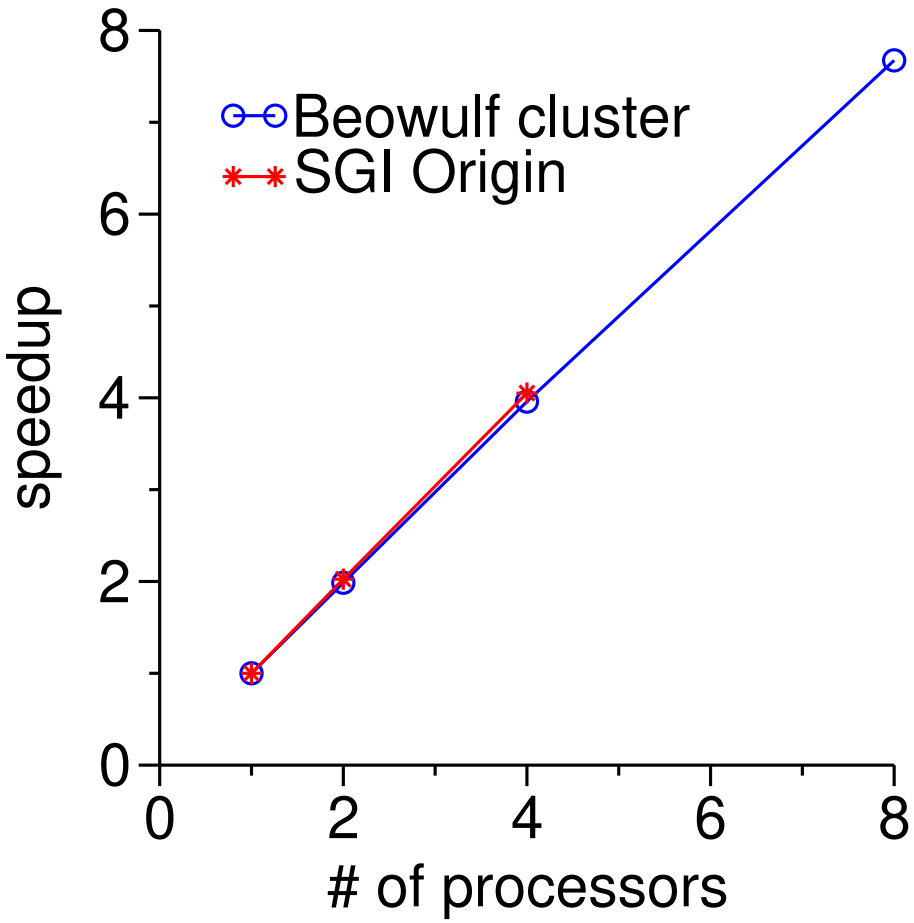
Parallel version of the same program

```
program monte_carlo
...
<read input data>
...
<broadcast input data to all processors>
for mc=1 to mc=nrun/np do
   <generate initial configuration>
   <change configuration: random process>
   <measure quantities of interest>
   <update running average of these quantities>
end do
<collect averages on master processor>
 ...
<print results>
...
```

- program basically 100% parallel
- speedup = # of processors
- communication hardware unimportant

Speedup for Monte-Carlo Program

$2d$ Ising model

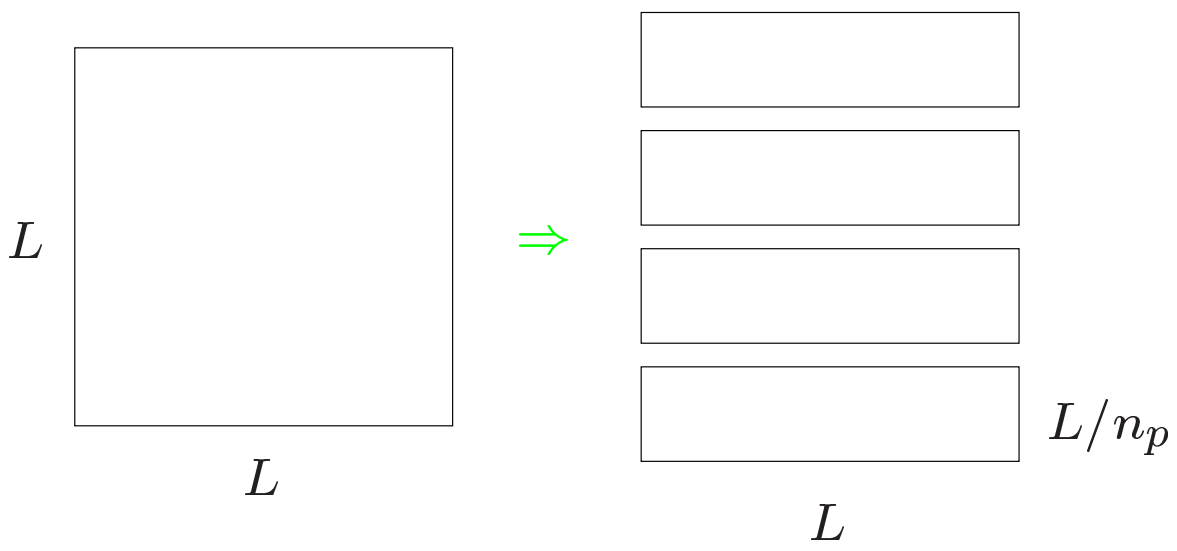


Partial Differential Equations

Domain Decomposition

- split system into several parts ($\rightarrow$ domains)
- assign one domain to each processor
- communication required at the boundaries of the domains

System size $L \times L$, n_p processors



Example: Allan-Cahn equation

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \nabla^2 f(\mathbf{r},t) + f(\mathbf{r},t) - f(\mathbf{r},t)^3$$

Discretization:

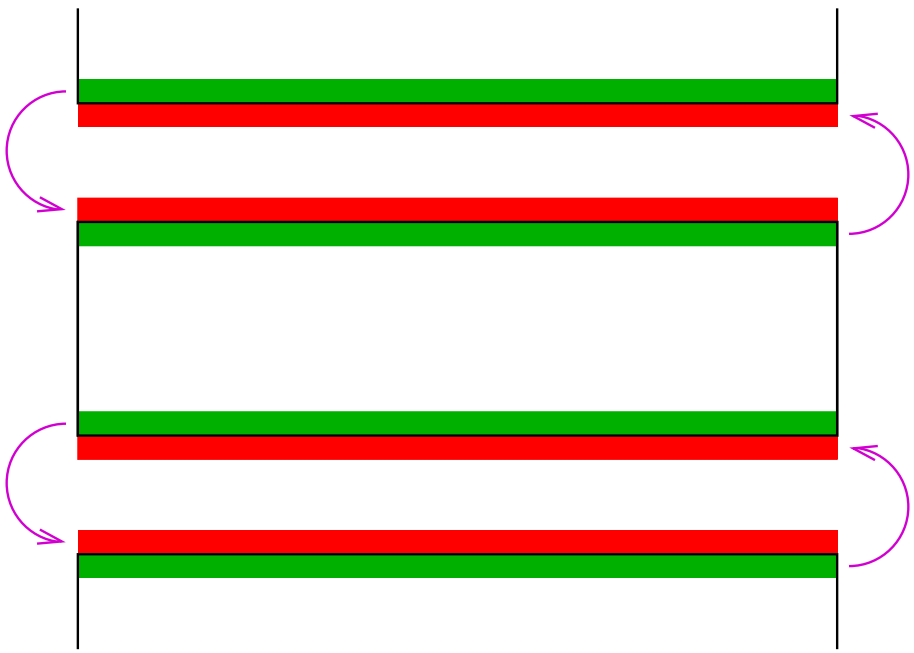
Euler method for time stepping:

$$\frac{\partial f(\mathbf{r},t)}{\partial t} = \text{rhs}(t) \rightarrow \frac{f(\mathbf{r},t+\Delta t) - f(\mathbf{r},t)}{\Delta t} = \text{rhs}(t)$$

Finite differences for ∇^2:

$$\nabla^2 f(\mathbf{r},t) \rightarrow$$
$$\frac{1}{\Delta x^2}\Big[f(x+\Delta x,y,t) + f(x-\Delta x,y,t)$$
$$+ f(x,y+\Delta x,t) + f(x,y-\Delta x,t) - 4f(x,y,t)\Big]$$

requires communication at the boundaries of the domain:



$\Rightarrow$ two send-receive operations in each time step.

Cost of Communication

No. of data to be transferred in each timestep:

$$N_{\text{trans}} = 2 \times L \times n_p$$

Total CPU-time:

$$t_{\text{cpu}} \sim L^2$$

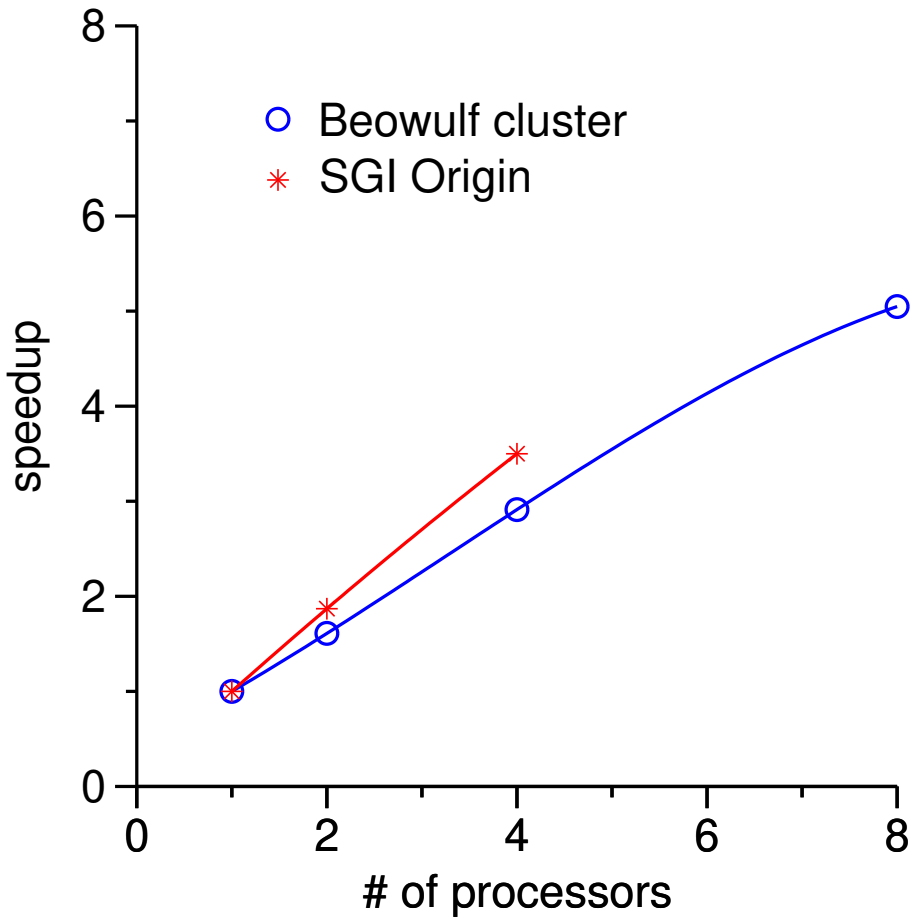
Fraction transfer-time/cpu-time:

$$t_{\text{com}}/t_{\text{cpu}} \sim n_p/L \overset{L\to\infty}{\longrightarrow} 0$$

- Domain decomposition becomes more efficient for larger system sizes!

Speedup for PDE with Domain Decomposition

$2d$ Allan-Cahn equation, mesh size 800×800



Domain Decomposition for PDE

- distributed memory allows you to do larger system sizes: MPI allows to access the memory on all nodes
- speedup quite good even with fast ethernet
- whether a numerical algorithm is good or bad now depends on the amount of communication required:

  Euler – just two sendrecv operations per timestep
  $\rightarrow$ fairly little communication

  Crank-Nicholson – solution of linear set of equations required in each time step $\rightarrow$ very inefficient for distributed memory, but possibly ok for SMP
  $t_{\text{com}}/t_{\text{cpu}}$ no longer goes to zero with L.

  Spectral Methods – two fast Fourier transforms (FFT) per time step

Fast Fourier Transforms

for $2d$ system of size $L \times L$:

$$\begin{aligned}\widehat{f}(k_x,k_y) &= \sum_{x,y} f(x,y)\, e^{-i(k_x x + k_y y)} \\ &= \sum_{x} \left[f(x,y) e^{-ik_y y} \right] e^{-ik_x x} \\ &= \sum_{x} \widehat{f}_y(x,k_y) e^{-ik_x x}\end{aligned}$$

with

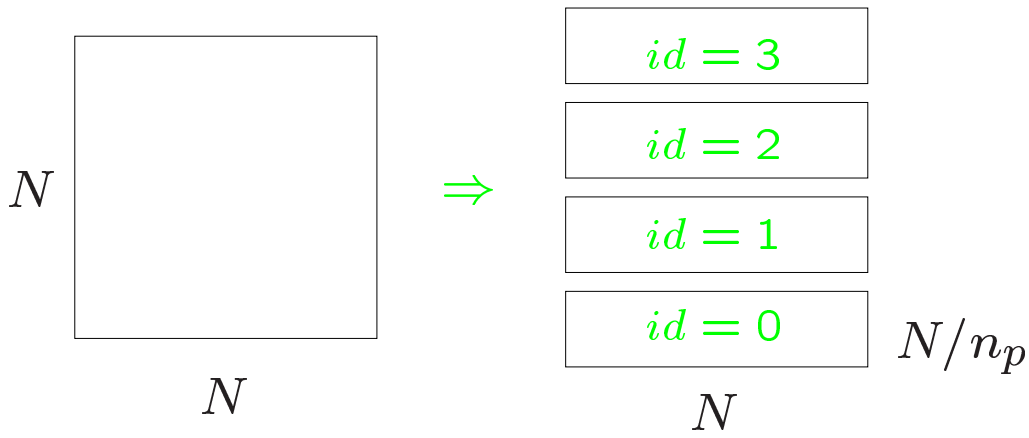
$$\begin{aligned}N &= L\Delta x\,, \\ x &= \mu\Delta x\,, \qquad \mu = 0,1,\ldots,N-1\,, \\ y &= \nu\Delta x\,, \qquad \nu = 0,1,\ldots,N-1\,, \\ k_x &= \frac{2\pi}{L} m\,, \qquad m = 0,1,\ldots,N-1\,, \\ k_y &= \frac{2\pi}{L} n\,, \qquad n = 0,1,\ldots,N-1\,.\end{aligned}$$

Implementation:

```
for mu=0 to mu=N-1 do
   <1d FFT with respect to y>
end do
for n=0 to n=N-1 do
   <1d FFT with respect to x>
end do
```

Parallel Version, Domain Decomposition

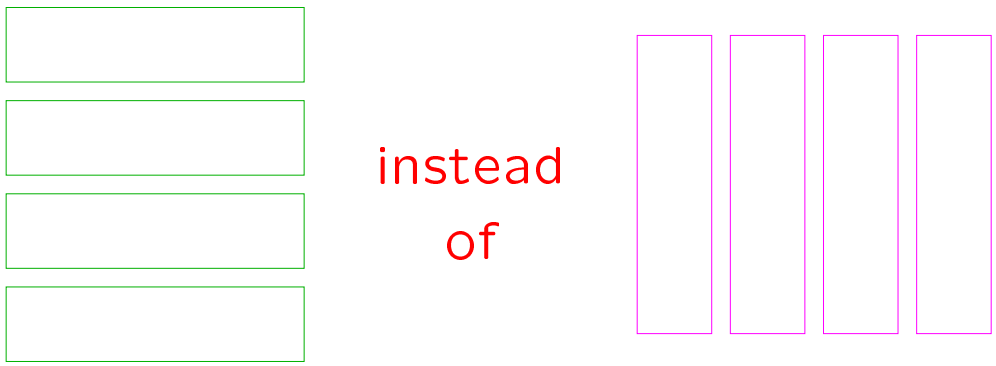
Domain decomposition as before



Works fine for the first part:

```
for mu=id*N/np to mu=(id+1)*N/np-1 do
   <1d FFT with respect to y>
end do
```

but not for the second part:



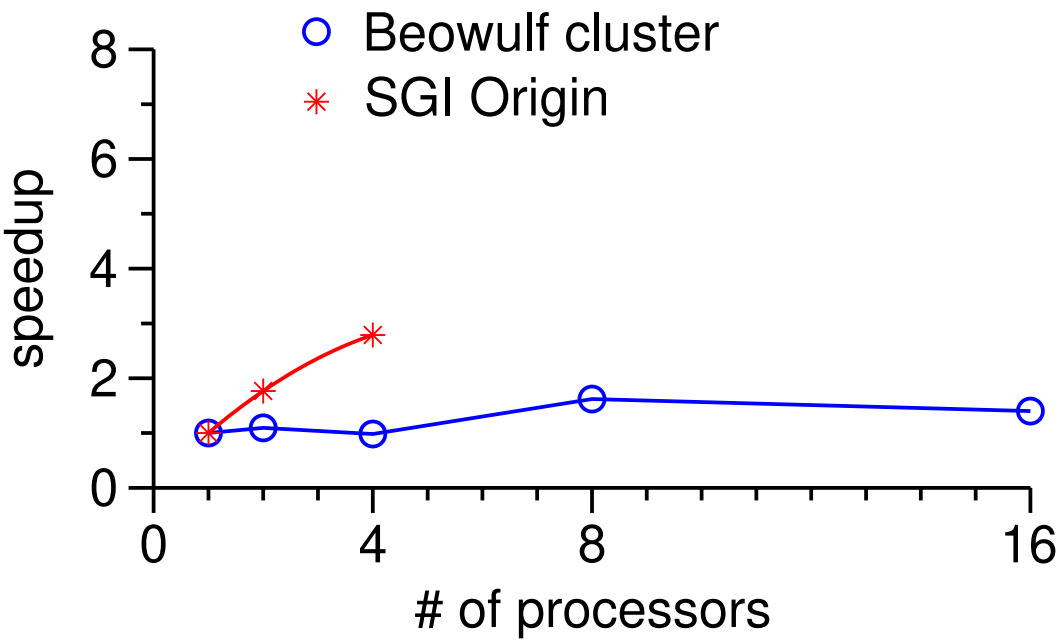

```
for n=id*N/np to n=(id+1)*N/np-1 do
⇒     <1d FFT with respect to x>
end do
```

does not work!

Parallel FFT in $2d$

```
for mu=id*N/np to mu=(id+1)*N/np-1 do
   <1d FFT with respect to y>
end do
<transpose matrix>
for n=id*N/np to n=(id+1)*N/np-1 do
   <1d FFT with respect to x>
end do
(<transpose matrix>)
```

- requires massive communication: all-to-all



Conclusion

- Monte-Carlo simulations ideal for Beowulf
- Integration of differential equations can be done on a Beowulf, if algorithm is chosen appropriately: $t_{\text{com}}/t_{\text{cpu}} \overset{L\to\infty}{\longrightarrow} 0$
  - PDE: Euler, Predictor-Corrector, etc.
  - Computational Fluid Dynamics (CFD)
  - Molecular Dynamics
- Algorithms that require large scale communication (system size) don't work very well on a Beowulf. On SMP this is nevertheless possible.

Future Developments

- Gigabit Ethernet
- ccNuma for Linux, Linux SMP
- IA 64