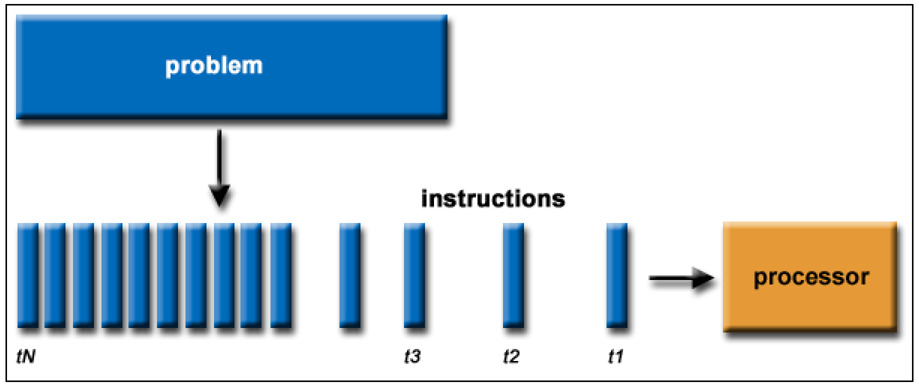
Introduction to parallel Computing

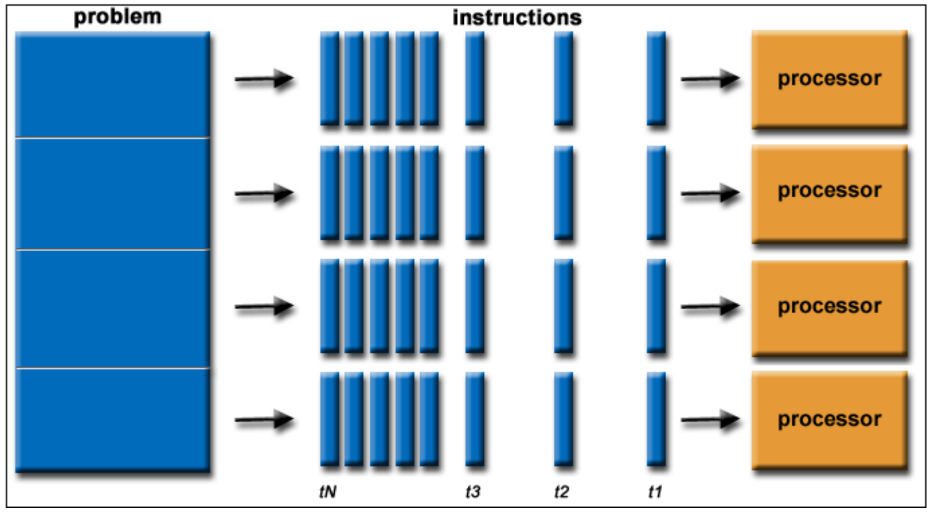
Outline

- Serial vs Parallel programming
- Hardware trends
- Why HPC matters
- HPC Concepts and Terminology
- Amdahl’s Law and Scalability
- Parallel programming models
- Communications
- Synchronization
- Load Balancing
- Pseudo-code examples

Serial vs Parallel Programming

- Traditionally software is written for serial computation
  - A problem is broken into discrete parts
  - Parts (instructions) are executed each at a time (serially)
- In parallel programming more than one processors are used
  - Problem is broken into discrete parts that can be solved **concurrently**
  - Several instructions are executed at a given time **asynchronously**
  - An overall control (syncing) mechanism is needed

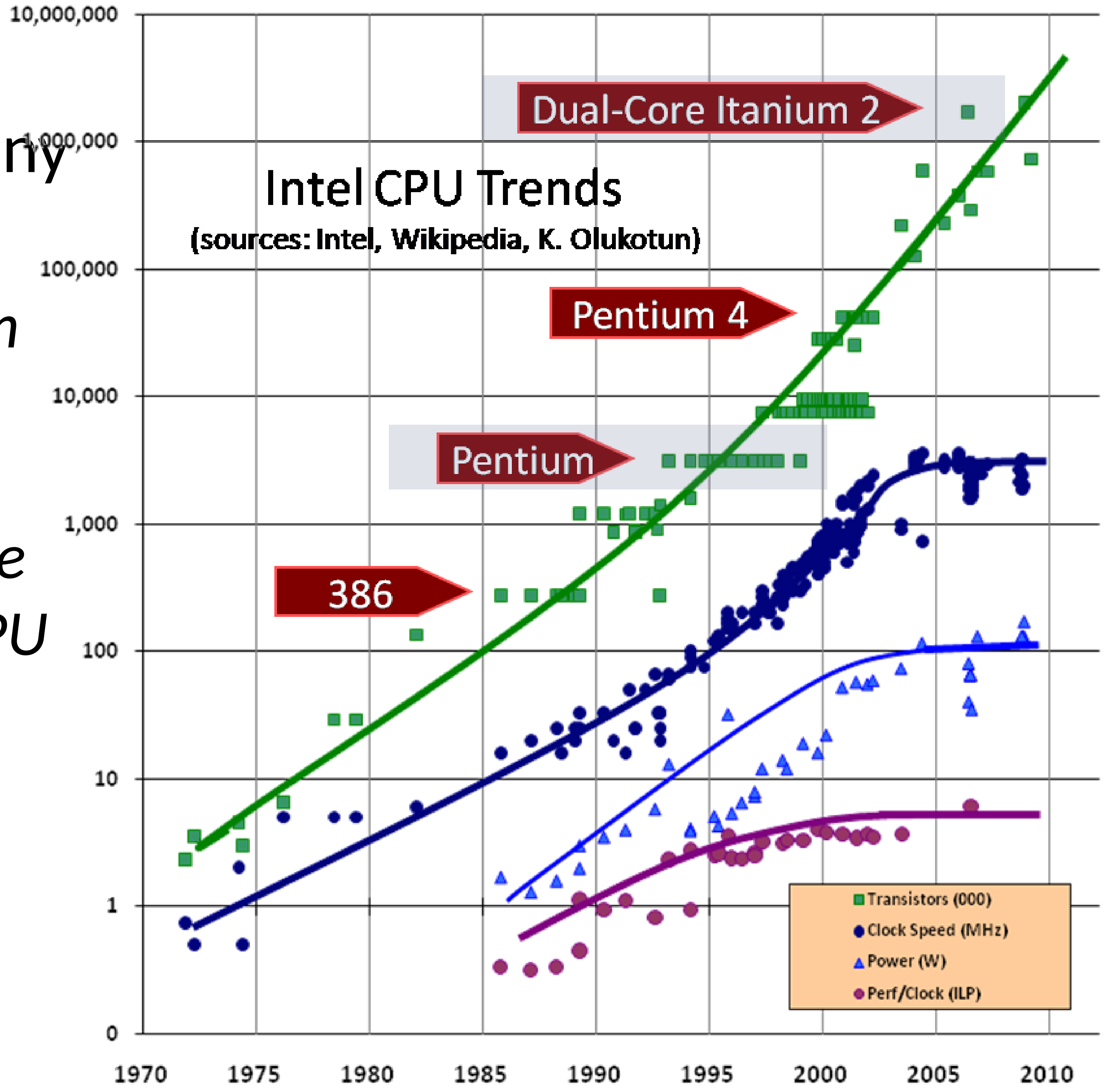




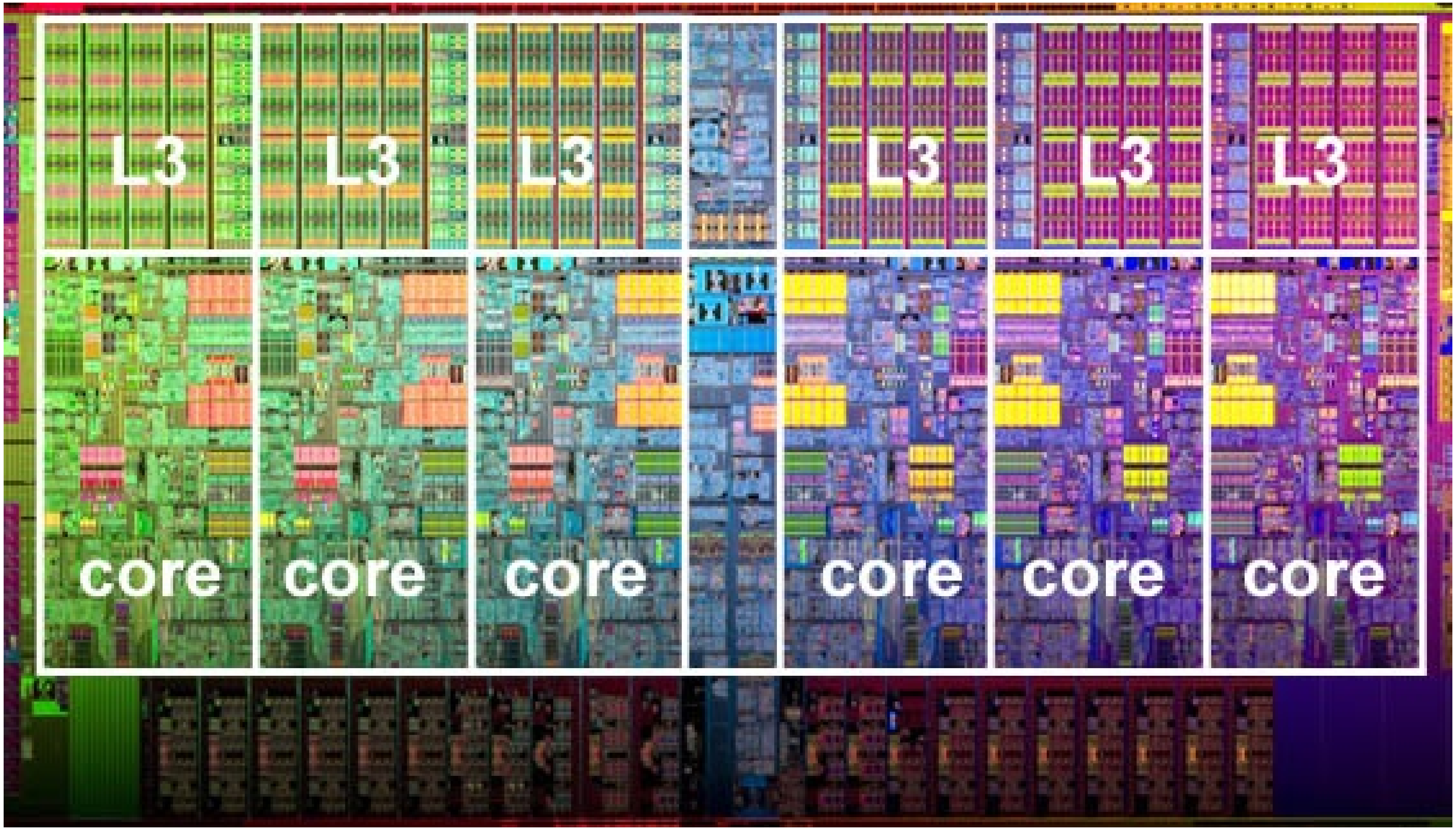
CPU trends

- We are not living in the 90s any more…

  “From 2007 to 2011, maximum CPU clock speed (with Turbo Mode enabled) rose from 2.93GHz to 3.9GHz, an increase of 33%. From 1994 to 1998, CPU clock speeds rose by 300%.”

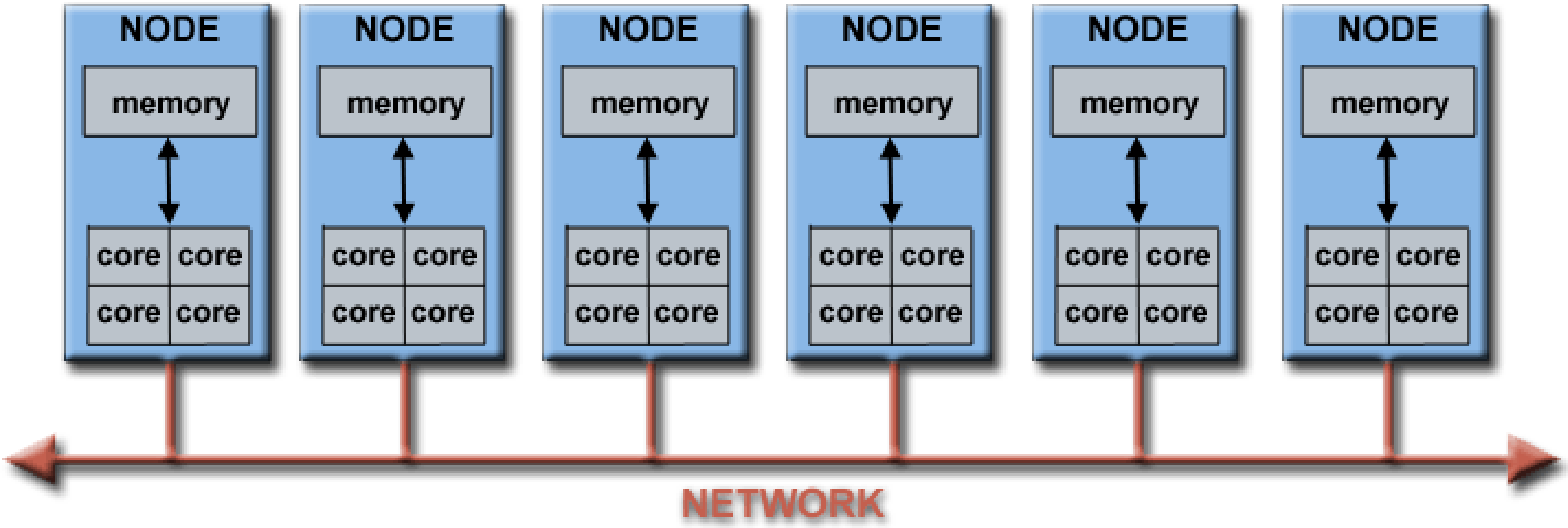


Hardware nowadays **is** designed for parallel computing



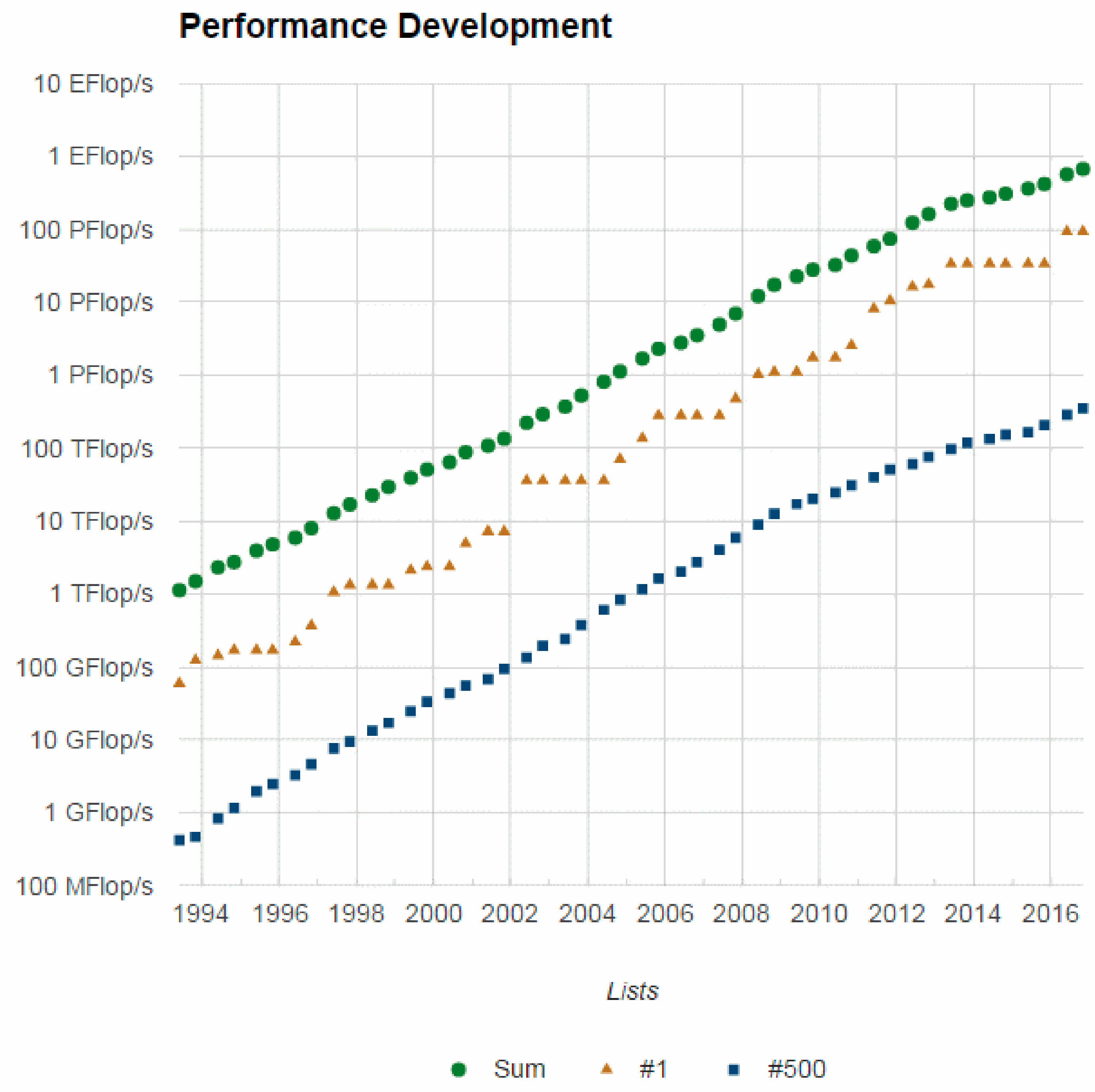
Networks *multiply* nodes into HPC clusters

- Each node is a multi-processor computer
- Multiple nodes are networked together
- Special purpose nodes, also multi-processor, are used for other purposes (i.e. GPU nodes, I/O nodes etc).



Voila!

- Development of:
  - Multi-core, multi-thread CPUs
  - Networks
  - Distributed (I/O) systems
  - GPUs
- (+ Parallel programming)



Why HPC matters

- Exact solutions are not always possible using current theoretical tools and methods
  - i.e. most problems we have to solve are non-linear
- Numerical integration and simulation technics are providing answers to difficult problems
- The more complex the problem the more demanding the solution will be. Hence, high end research requires
  - better hardware
  - improved software stacks etc

Science will most likely impose the following..

- *"Is it possible to reduce the time it takes to solve the problem?"*
- *"Is it possible to increase the problem size?"*

IT will likely respond..

- *"Yes, a new machine is on the way…"* – **Hardware specs**
- *"Try improving your code"* – **Software refactoring**
- *"Try linking with OpenBLAS or MKL"* – **Code re-use**

Basic Concepts and terminology

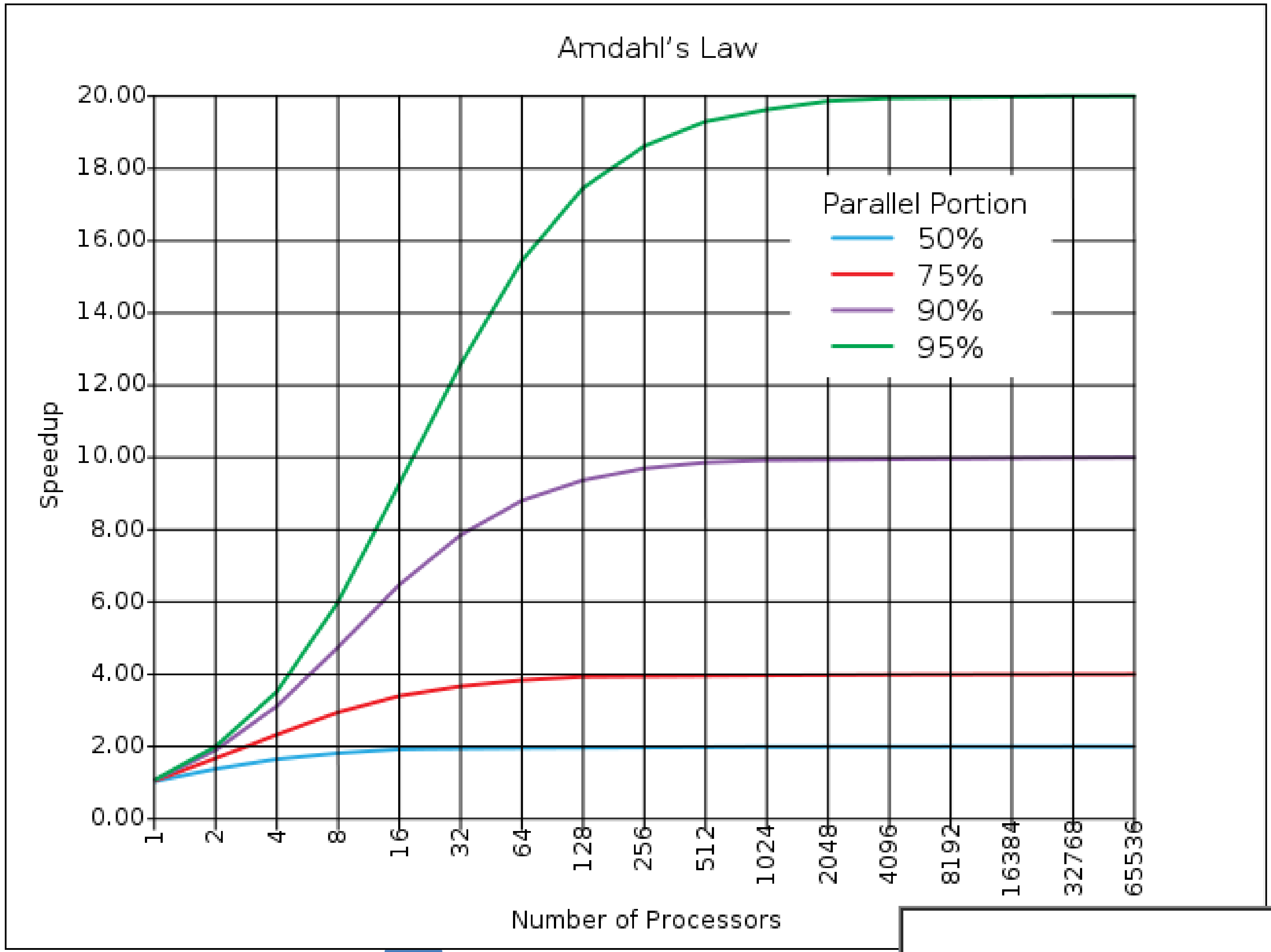
- A Compute **Node** is a standalone computer
  - It has multiple {**CPU**/Processor/Socket}s
    - Each CPU has multiple **Cores**
      - Each Core may support multiple **threads**
- A parallel program consists of multiple tasks
  - Each **task** is a “serial” set of instructions
    - Parallel **tasks exchange data** and
      - Hence may need to be **synchronized** (i.e. before or after the exchange takes place)

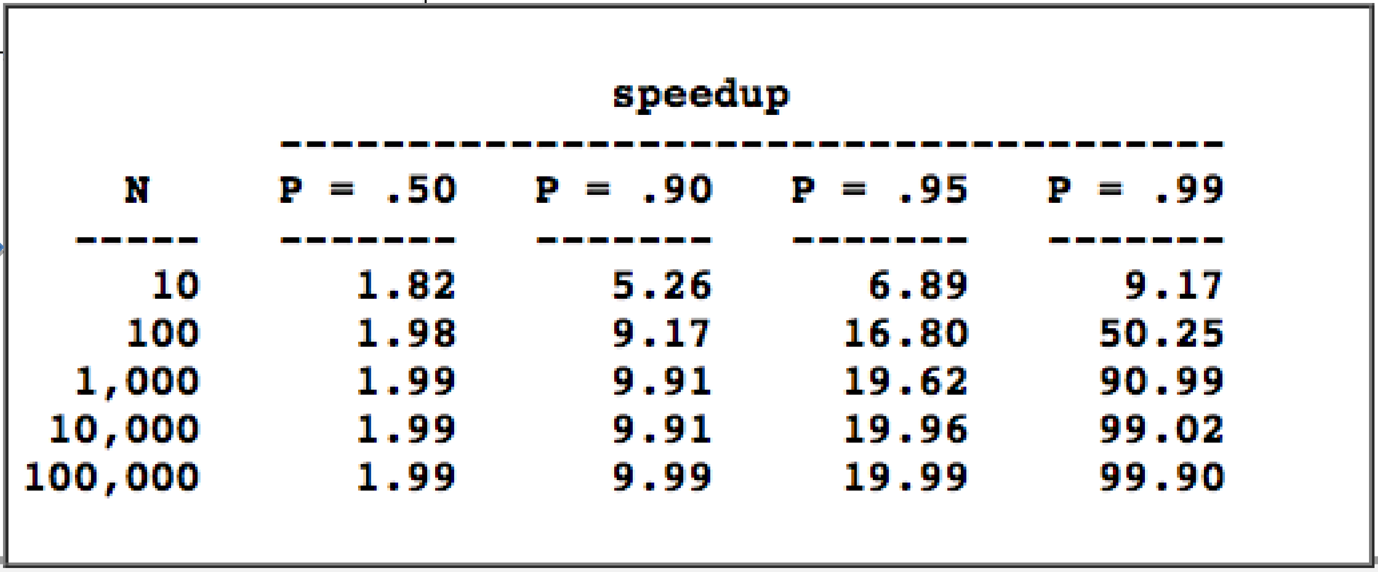
Concepts and terminology

- **Speedup** is the wallclock of the serial version over the wallclock of the parallel version
- **Parallel overhead** is the extra overhead (amount of time) needed to build up the parallel execution environment
- **Scalability** is the ability of a given code to demonstrate speedup with the addition of more resources.

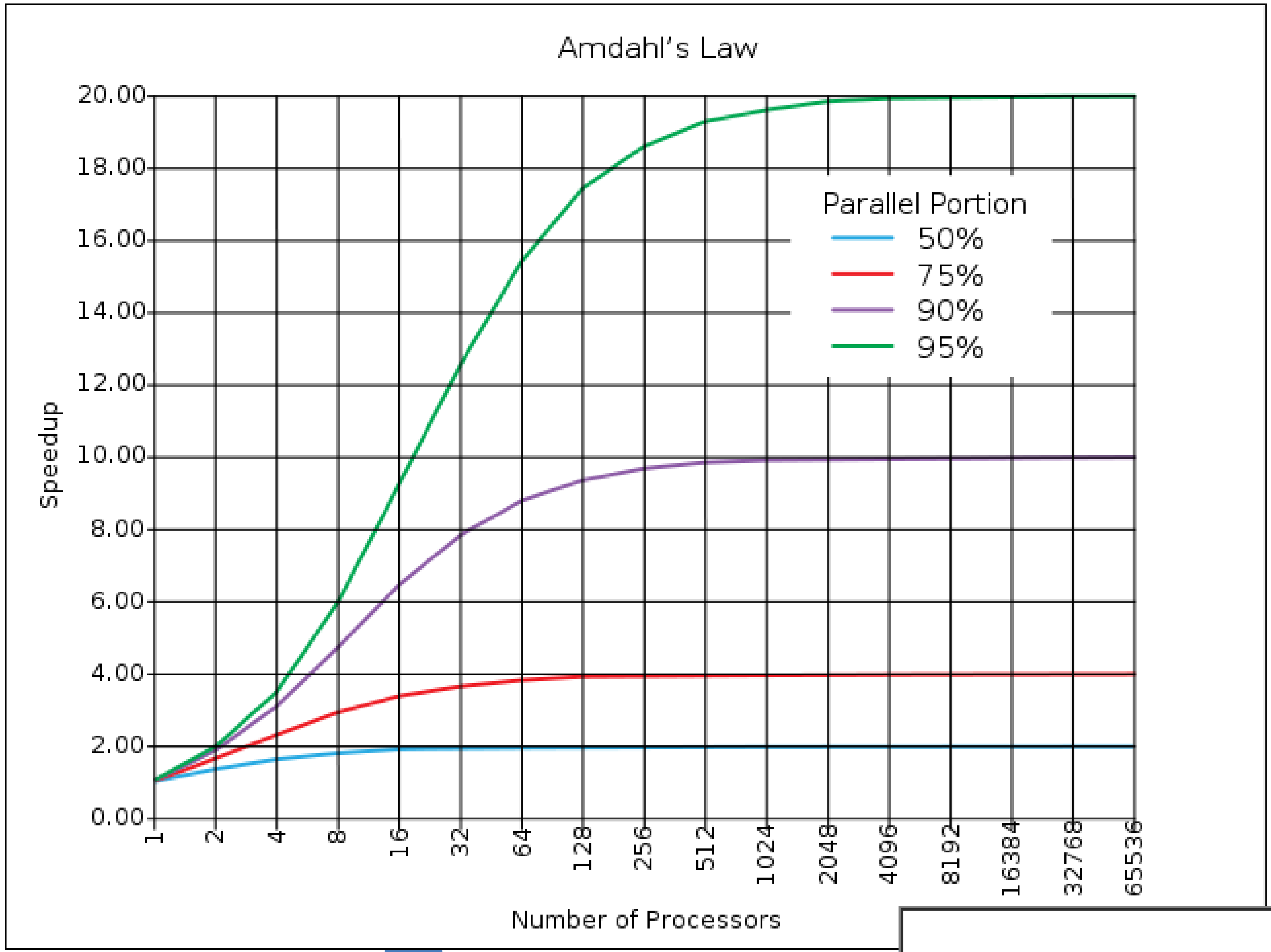
Amdahl’s Law

Amdahl’s law predicts the theoretical maximum speedup when using multiple processors



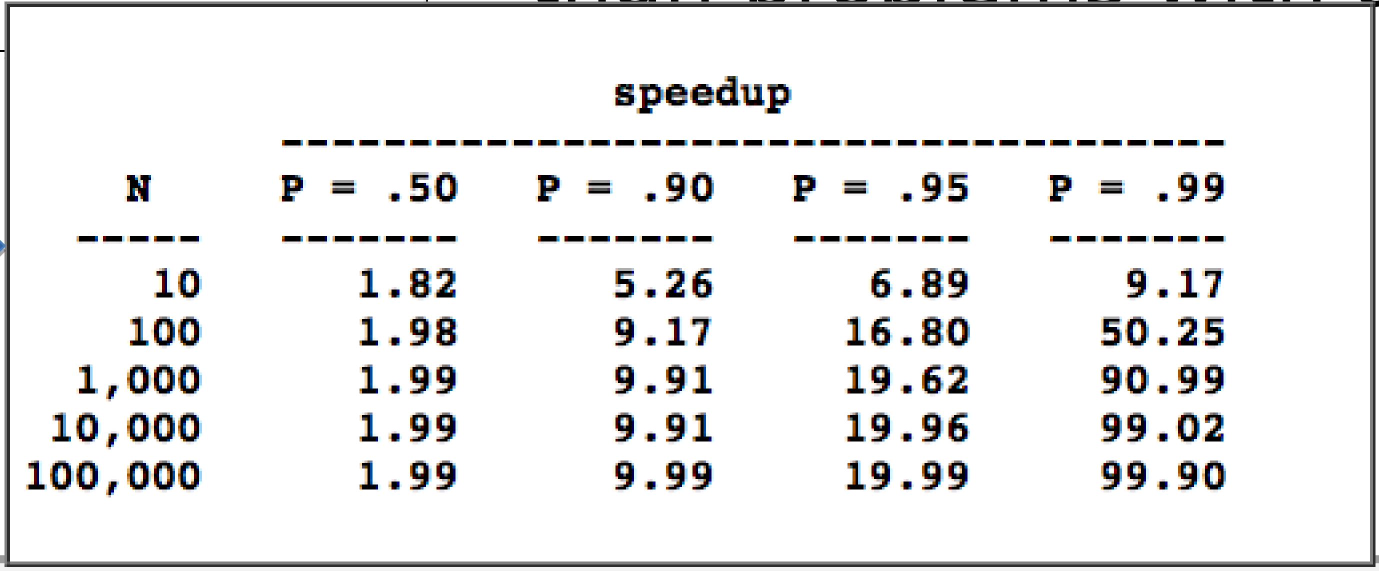


	speedup			
N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90



However, by increasing the problem size, the non-parallel segment can be reduced

- Problems that increase the percentage of parallel time with their size are more scalable than problems with a



	speedup			
N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

Scalability

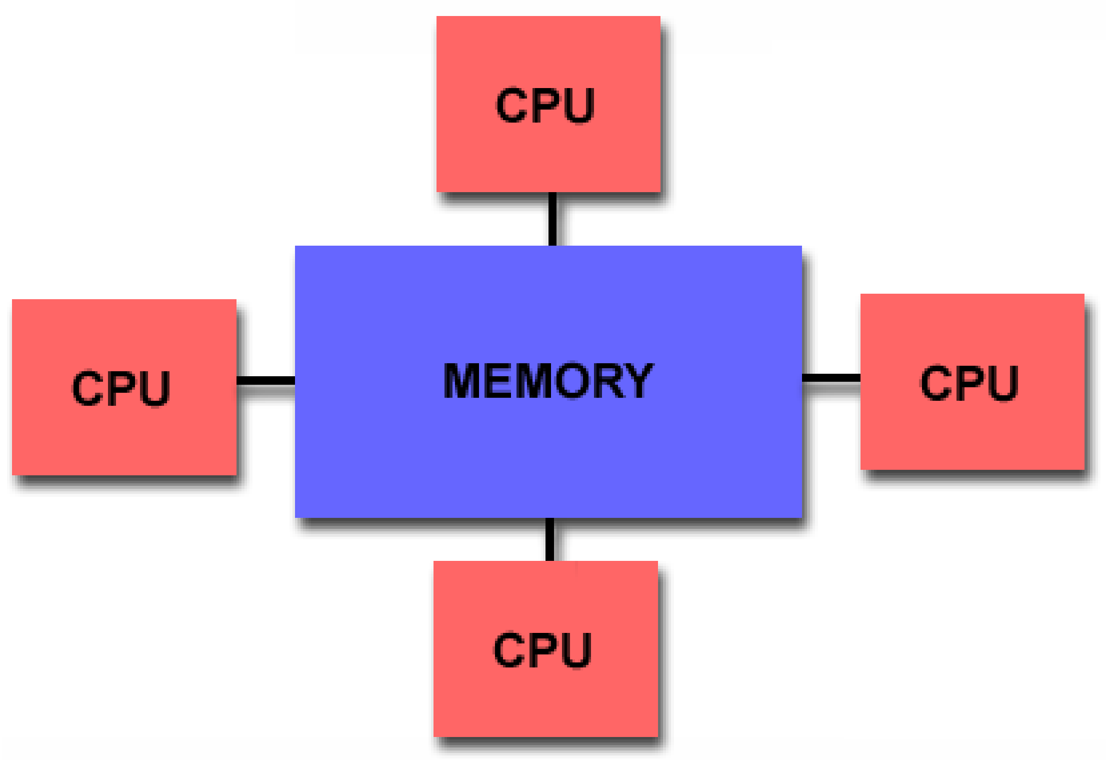
- Strong scaling
  - Problem size is fixed
  - Goal is to solve faster
  - Perfect scaling means speedup is ~P
- Weak scaling
  - Problem size <u>per processor</u> is fixed
  - Goal is to run a larger problem in the same unit of time
  - Perfect scaling means the larger problem is solved in the same unit of time

One final note…

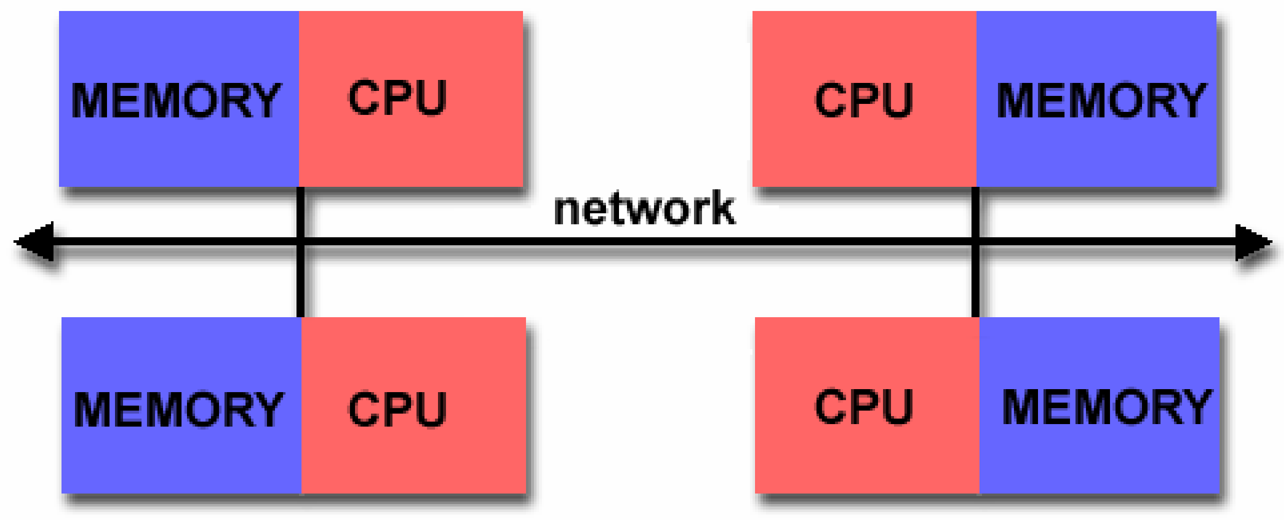
- Parallel programming may overcome the issues but before doing anything parallel make sure that:
  - Your serial code is already optimized! Questions to ask yourself:
    - *Are you using other people's computational and I/O libraries?*
    - *Have you tested with other compilers and, if yes, have you tried various optimization flags?*
    - *What does profiling tell you?*

Parallel programming models

Shared memory

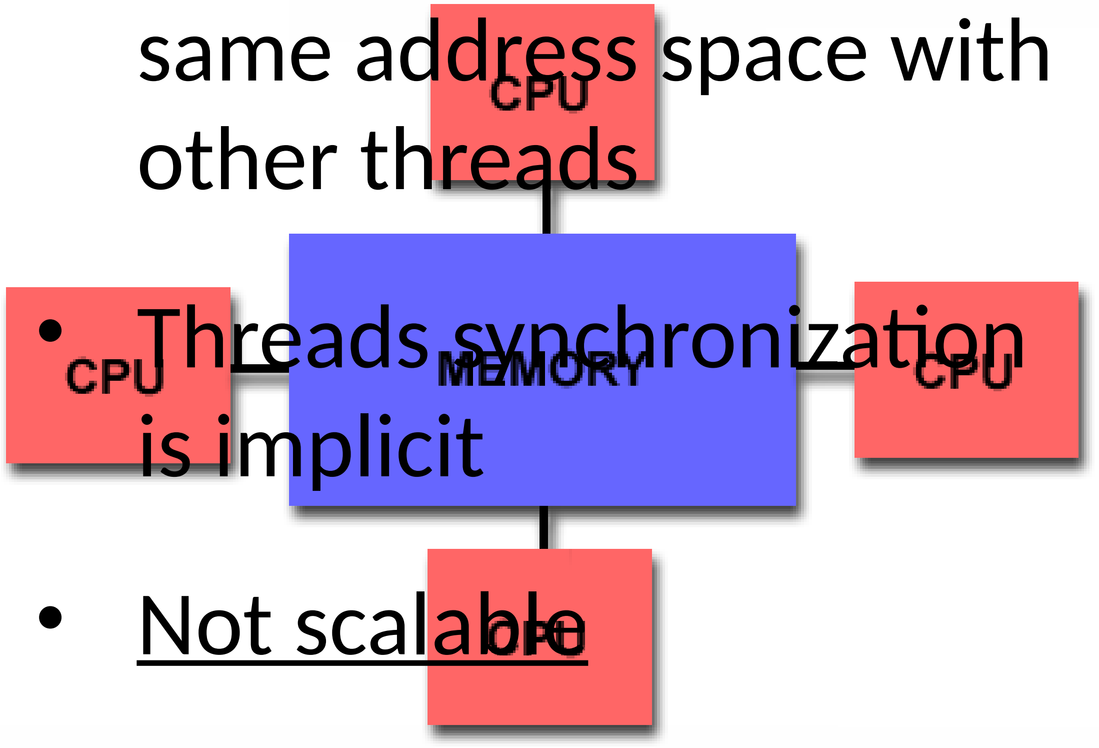


Distributed memory



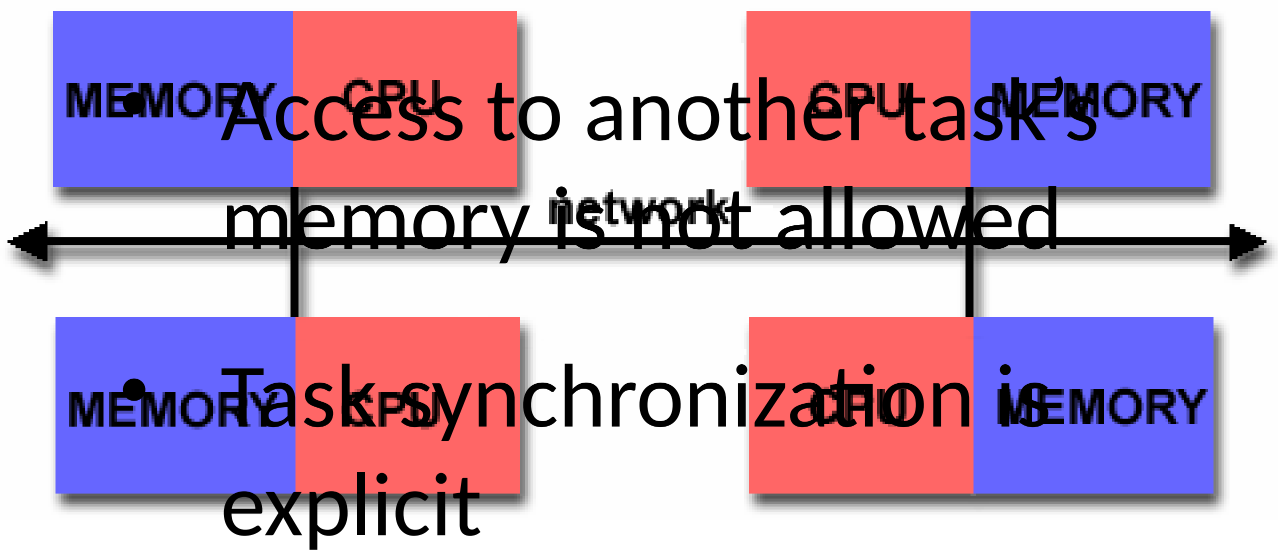
Parallel programming models

Shared memory



- Each **thread** shares the same address space with other threads
- Threads synchronization is implicit
- <u>Not scalable</u>

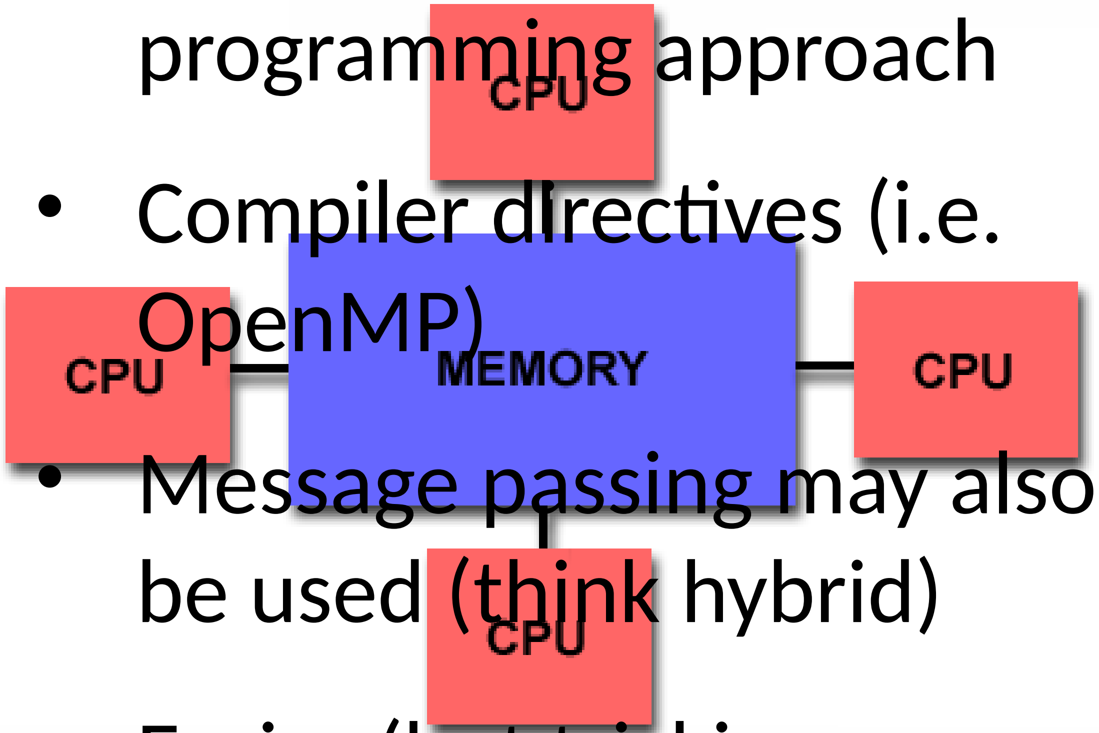
Distributed memory



- Each **task** has access to a unique address space
- Access to another task's memory is not allowed
- Task synchronization is explicit

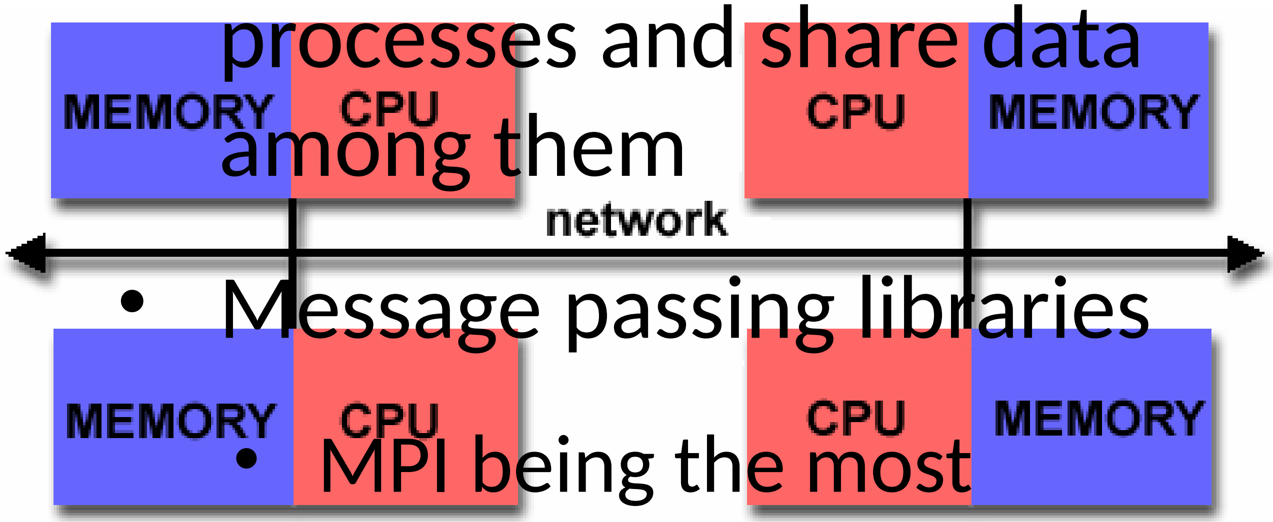
Parallel programming models

Shared memory



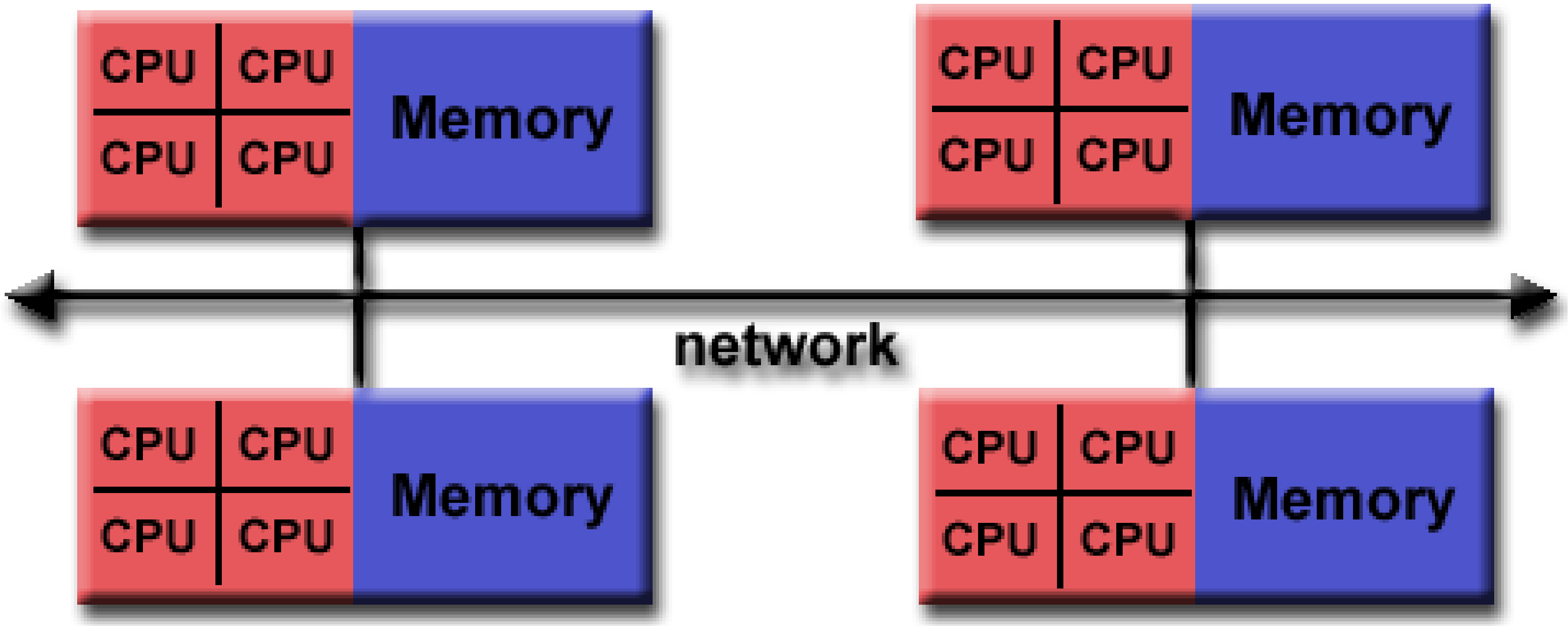
- Thread based programming approach
- Compiler directives (i.e. OpenMP)
- Message passing may also be used (think hybrid)
- Easier (but trickier sometimes) to implement

Distributed memory



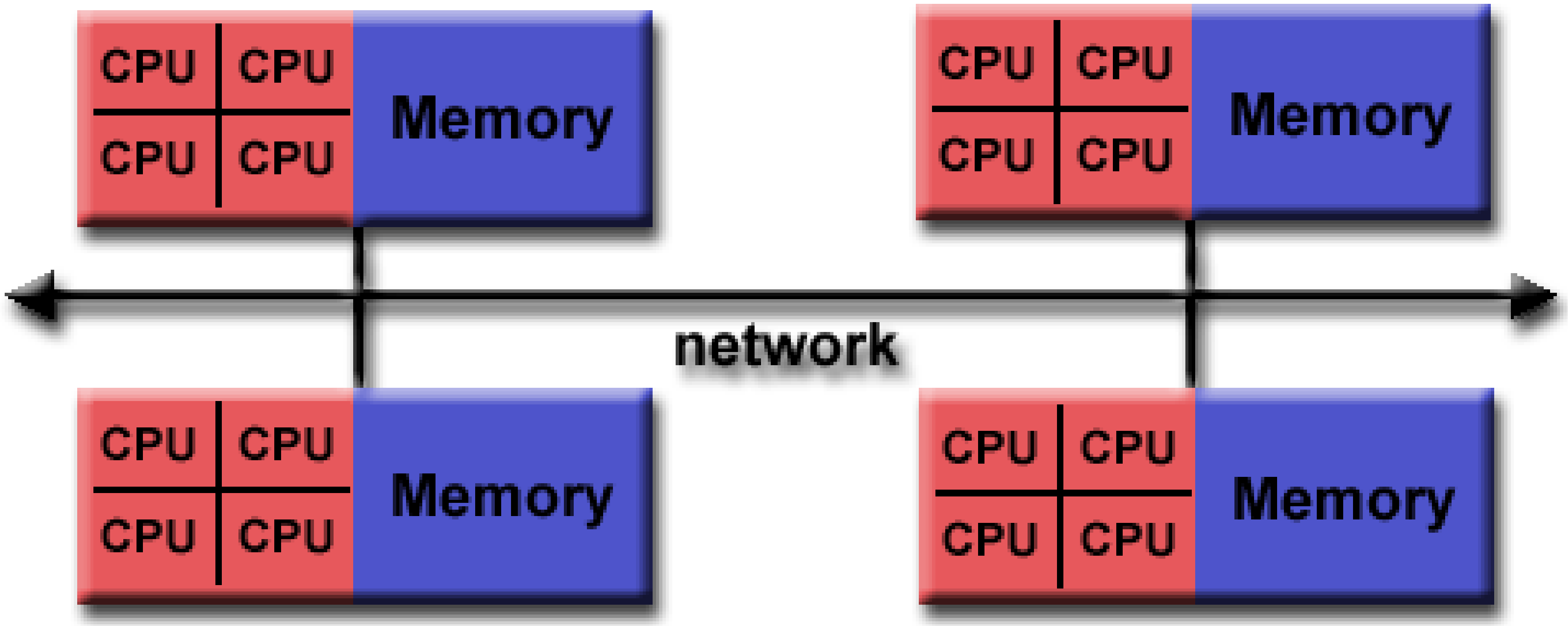
- Developer uses “Message Passing” in order to sync processes and share data among them
- Message passing libraries
  - MPI being the most common
- Difficult to implement

Hybrid Distributed-Shared memory



Hybrid Distributed-Shared memory

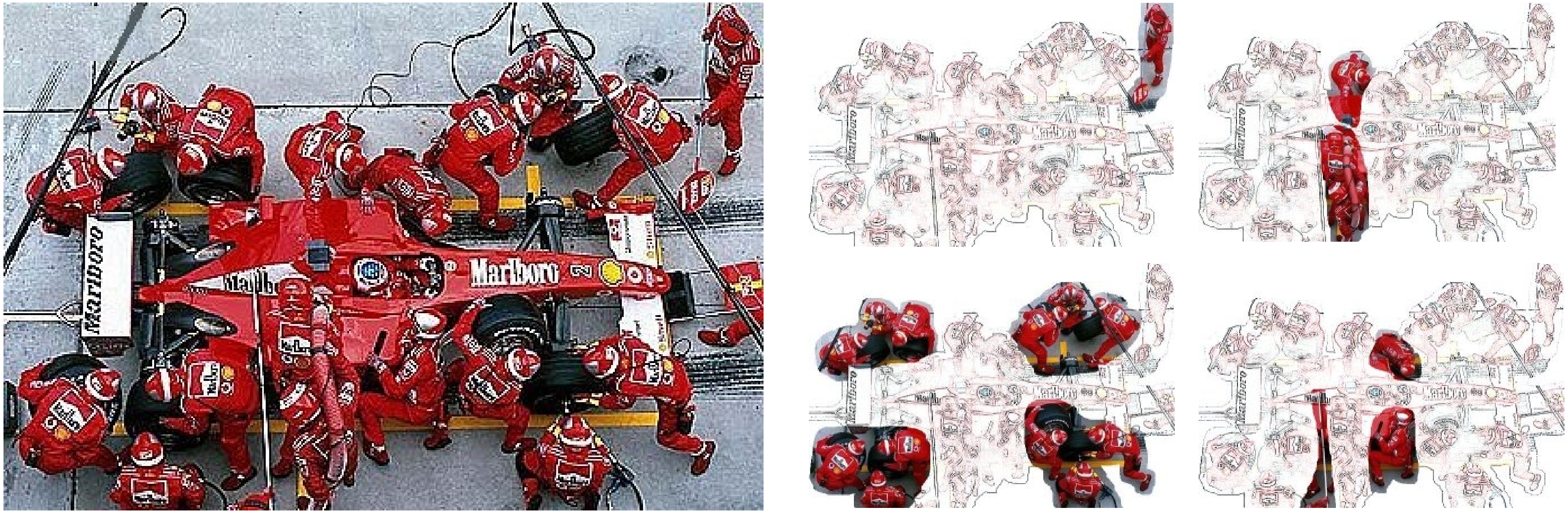
- Most (all?) HPC systems currently employ both shared and distributed memory architectures
- Th
  (ir
- Th
- A (
  - –
    el
  - –



Designing parallel programs

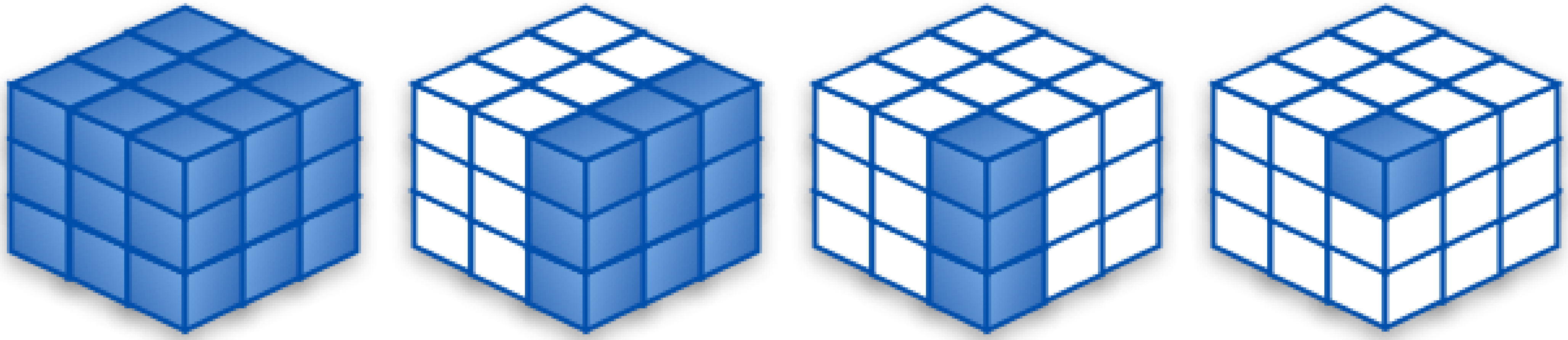
- First step: Understand the problem
  - Before beginning make sure the serial only version is already at its best
    - For example take advantage of optimized libraries such as OpenBLAS or MKL
  - Investigate if (and if yes, how) the program can be parallelized
- Profile the serial program’s runtime
  - Where is time mostly being spent
    - Focus on those parts
  - Identify potential bottlenecks (i.e. I/O steps)

Nature of the problem

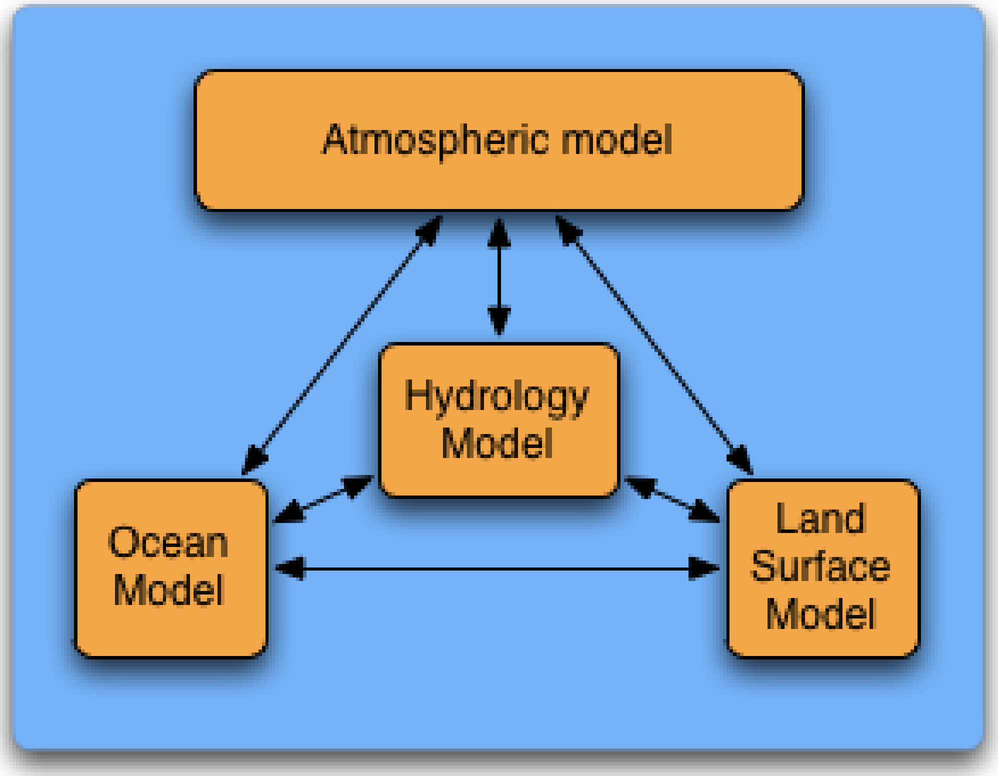


Nature of the problem (examples)

- Domain decomposition



- Functional partitioning


Atmospheric model

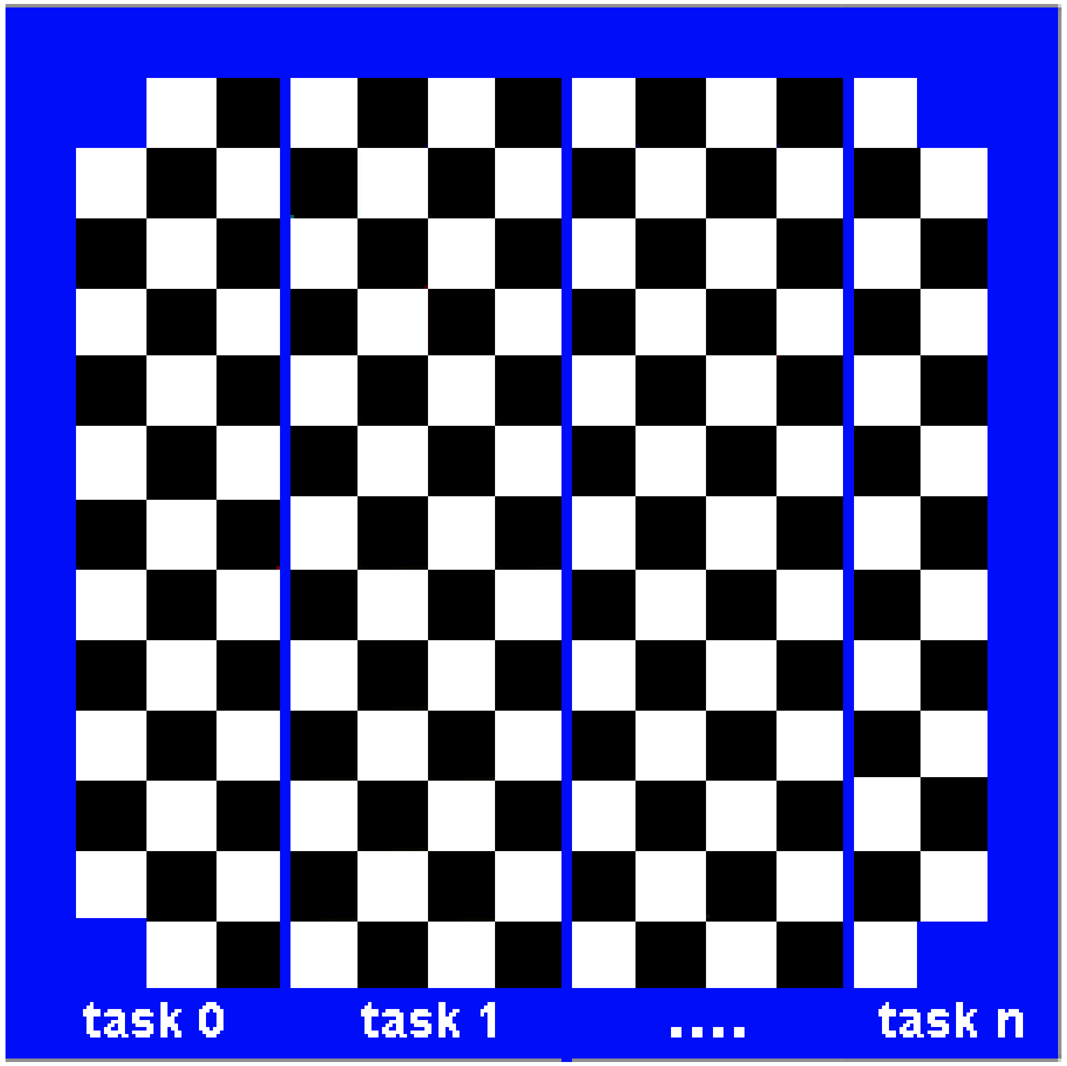
Hydrology Model

Ocean Model

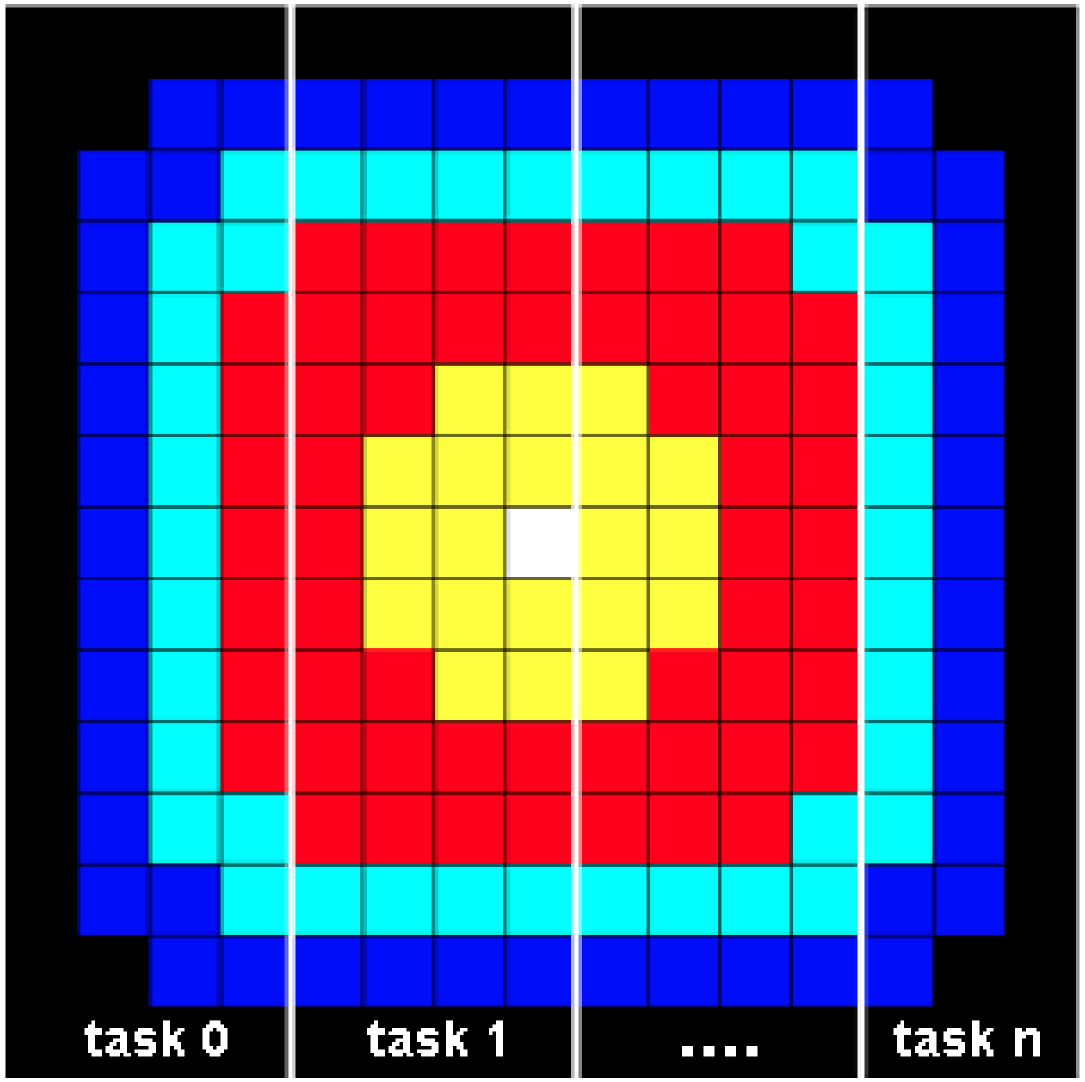
Land Surface Model

Communications (domain decomposition)

- Rendering a 2D image



- Explicit 2D scheme



Communications Overhead

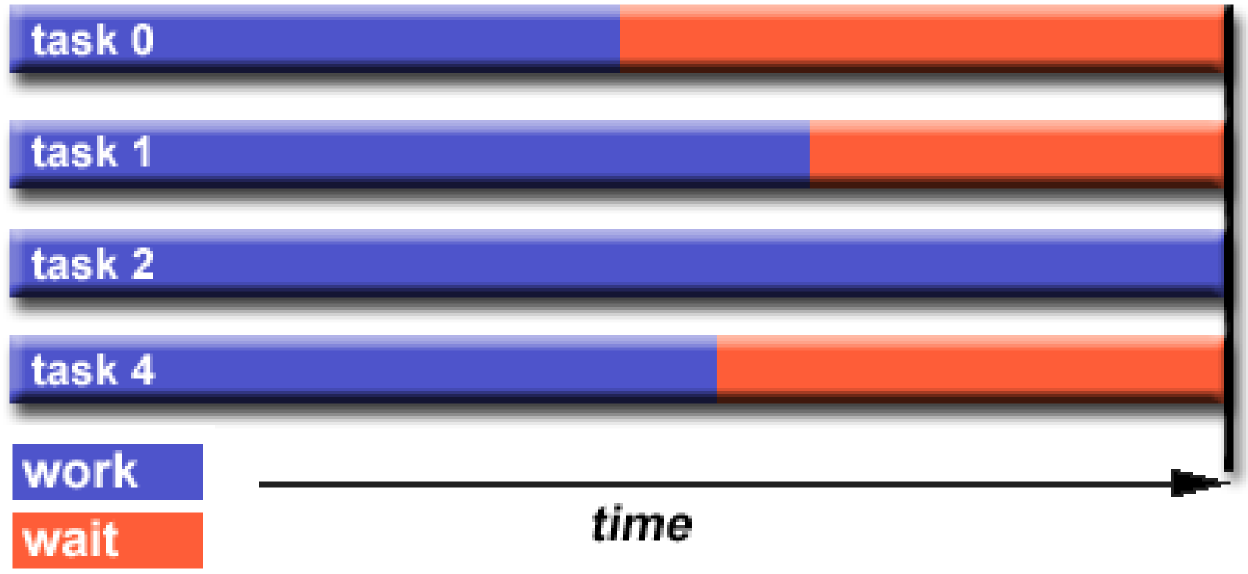
- Communications induce (inevitably) parallel overheads
  - CPU Cycles are used to dispatch/receive data
  - Points of synchronization need to be introduced
  - Traffic may saturate the available network bandwidth
- Modes of communication
  - Synchronous (or blocking)
  - Asynchronous (or non-blocking)
    - Used to <u>overlap computation with communication</u> in order not to waste CPU Cycles
- Efficiency of communications depends mostly on
  - MPI (or other protocol) Implementation being used
  - Network fabric

Synchronization types

- Barriers
  - All tasks are involved
  - All wait until the last (slowest) task reaches the barrier
- All synchronous communications
  - Collective ones are hence implicitly barriers
- Locks
  - May involve any number of parallel tasks
  - Used to protect a code segment from being executed in parallel (i.e. increments)
  - Can be blocking or non-blocking

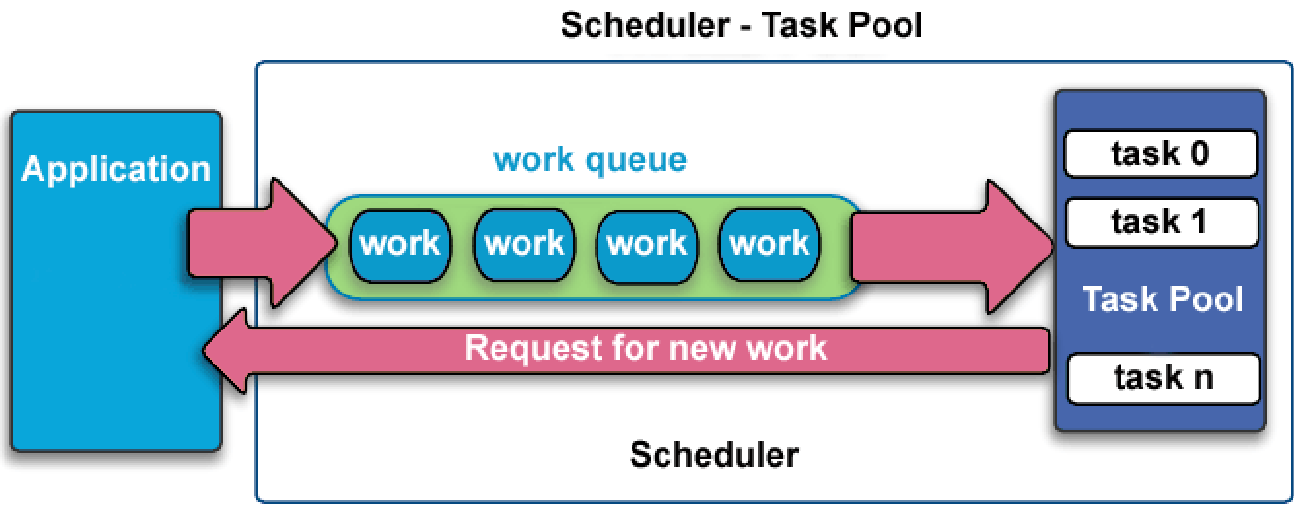
Load balancing

- The practice of
  - distributing approximately equal amount of work to all tasks
  - Minimizing idle time
- The slowest (most loaded) task determines the overall performance



Common approaches

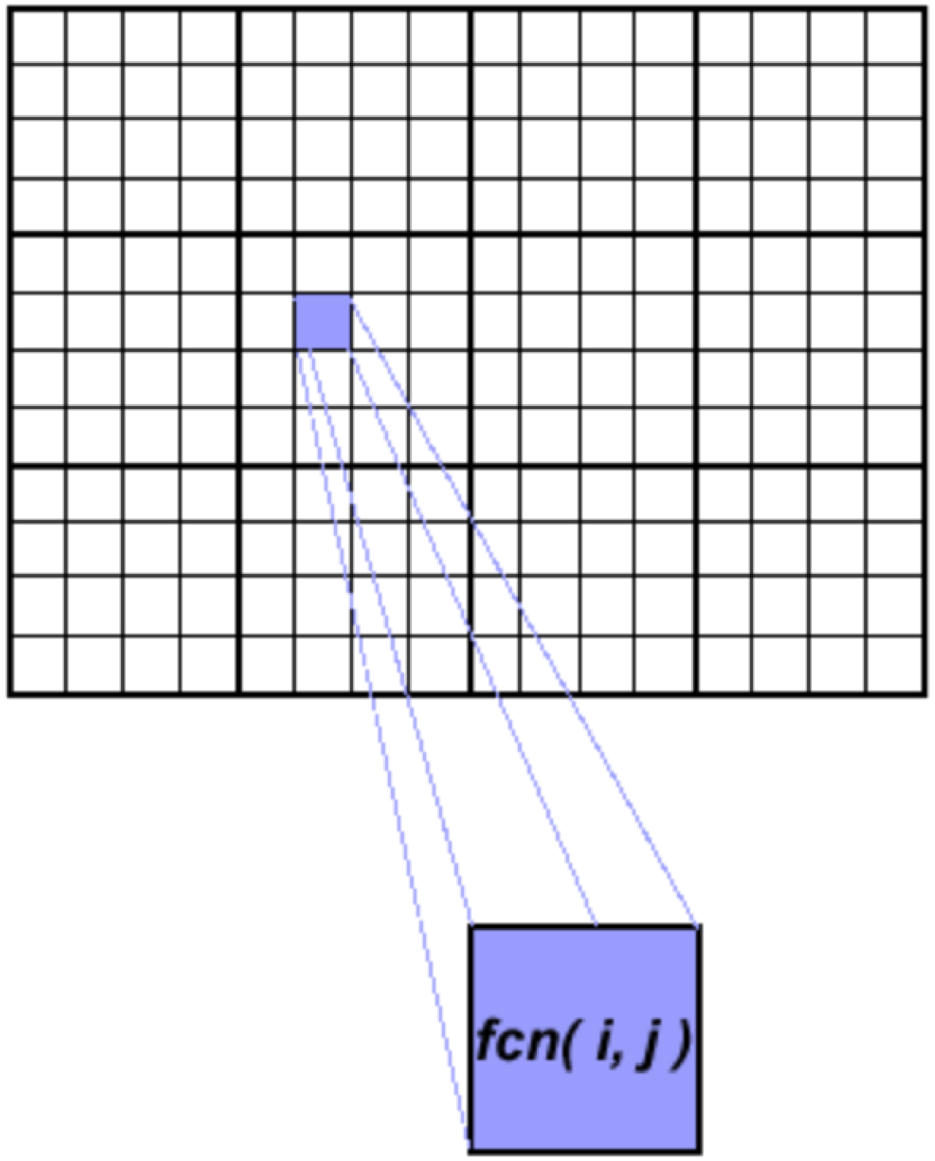
- Equally partition the work (whenever possible)
- But under circumstances this cannot be done beforehand
  - For example: Sparse linear algebra, adaptive mesh refinement and other use cases
  - In such cases we use dynamic scheduling of small chunks of workloads
    - Each task picks up a workload and requests a new one upon completion



Example – 2D Array

- Calculation of array elements

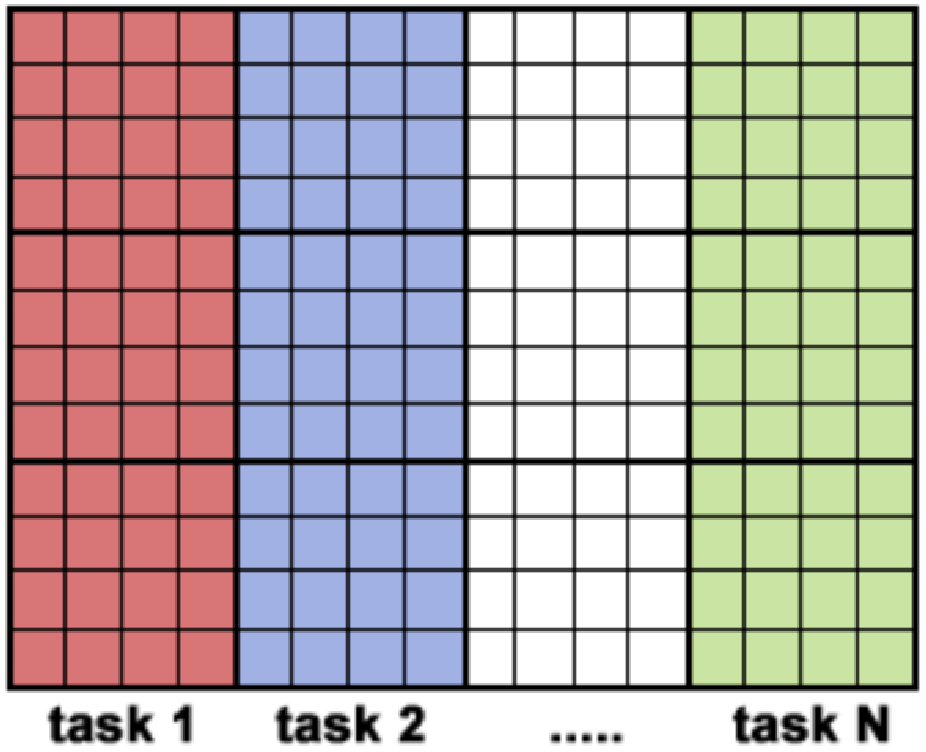
```
do j = 1,n
   do i = 1,n
      a(i,j) = fcn(i,j)
   end do
end do
```



Example – 2D Array

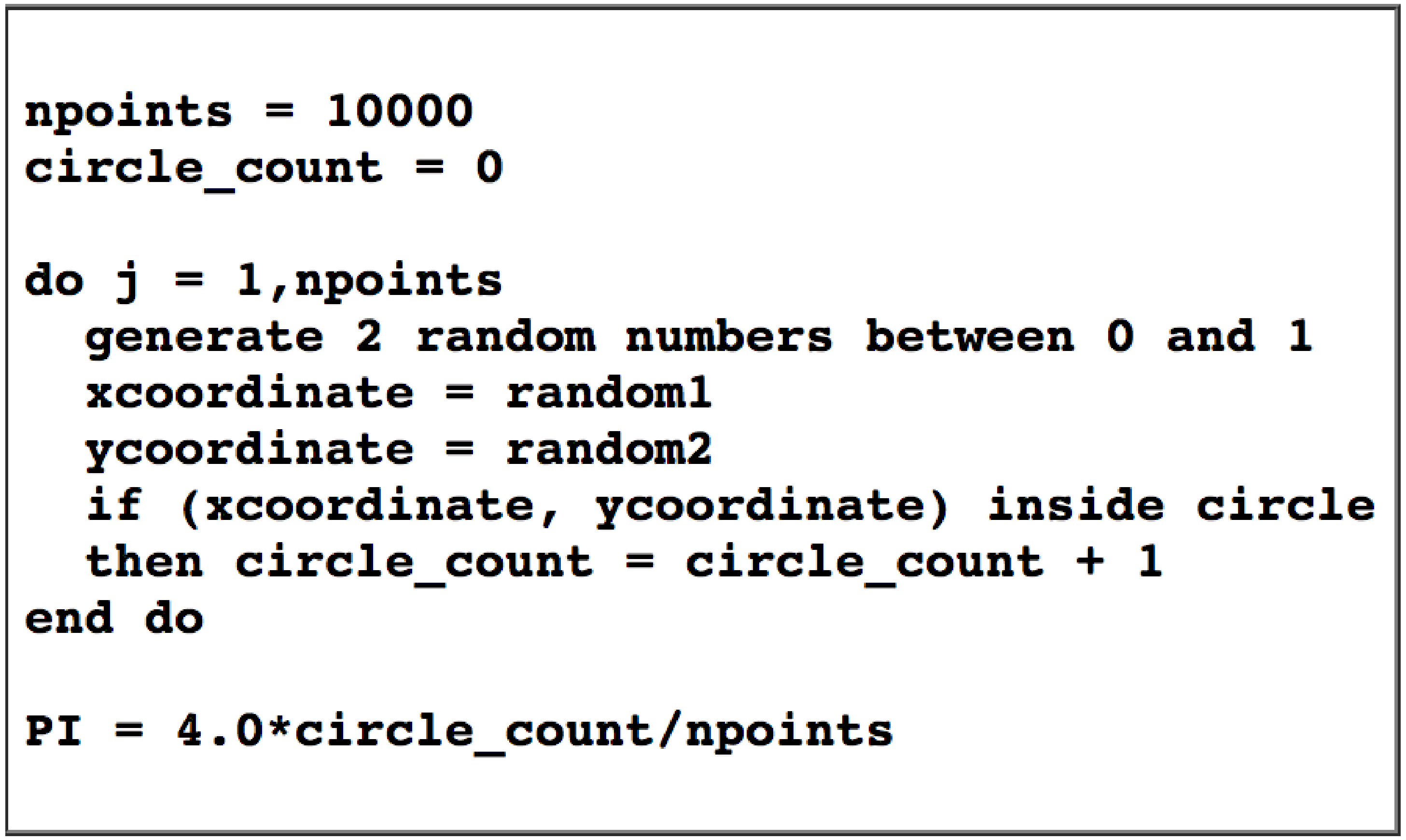
- Calculation of array elements

```
do j = mystart, myend
   do i = 1, n
      a(i,j) = fcn(i,j)
   end do
end do
```



Example – Pi calculation

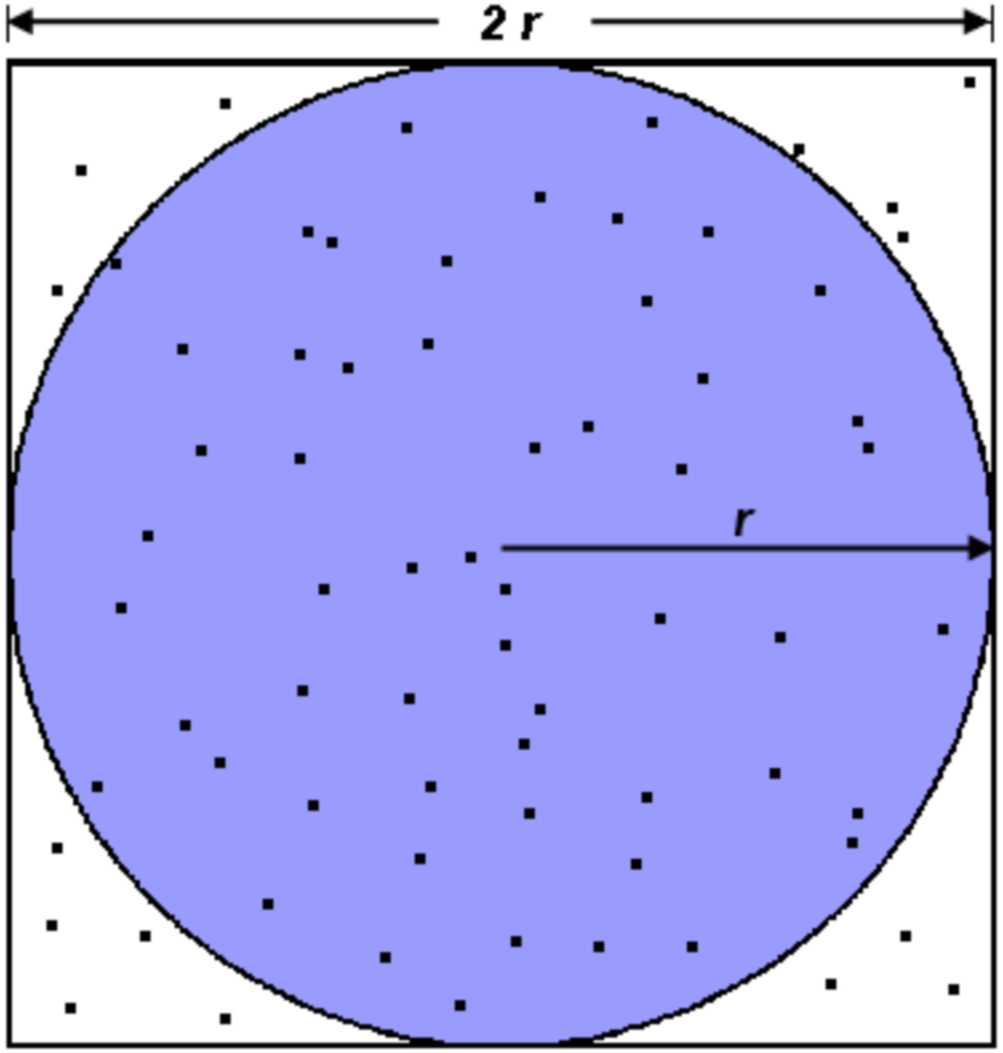
- Serial pseudo-code:



```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$

Example – Pi calculation

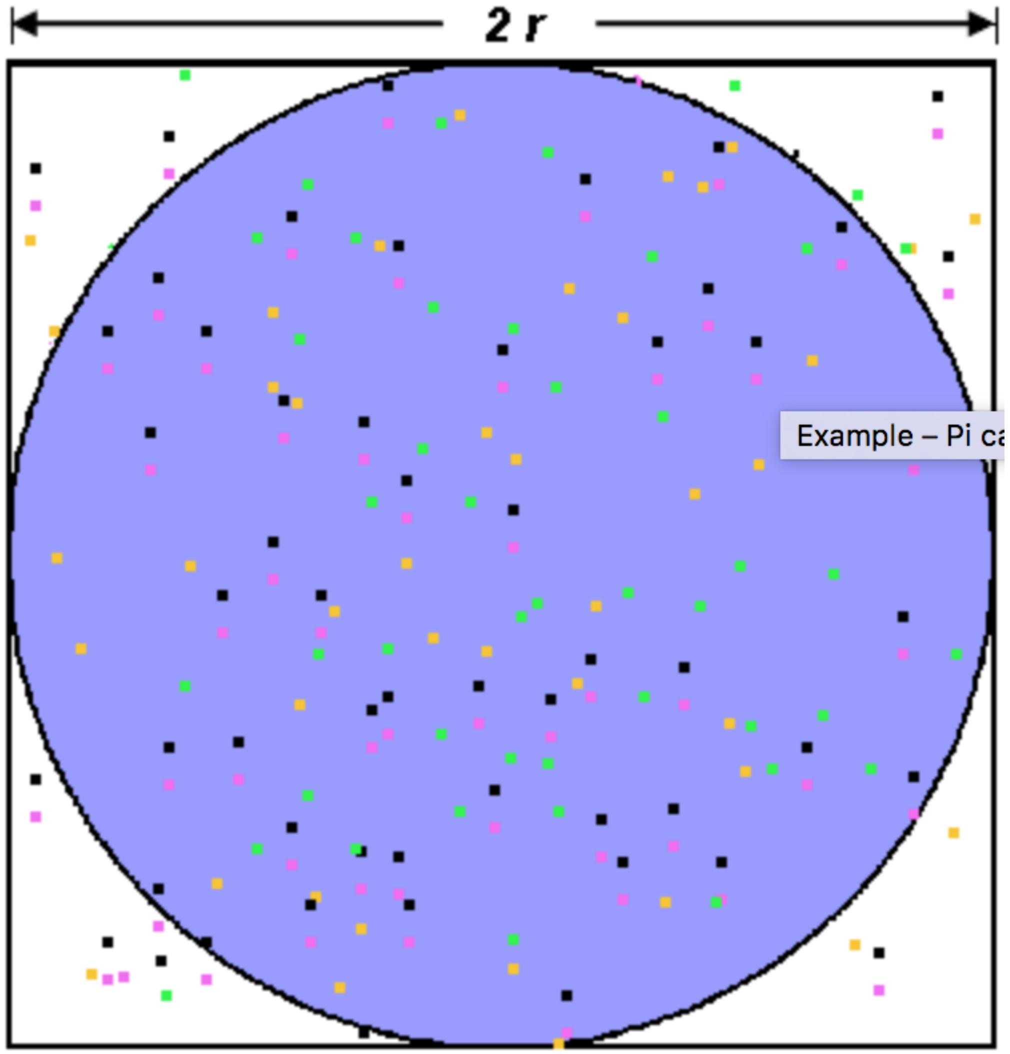
- Modified pseudo-code:

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

do j = 1,num
   generate 2 random numbers between 0 and 1
   xcoordinate = random1
   ycoordinate = random2
   if (xcoordinate, ycoordinate) inside circle
   then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```



$$A_S = (2r)^2 = 4r^2$$

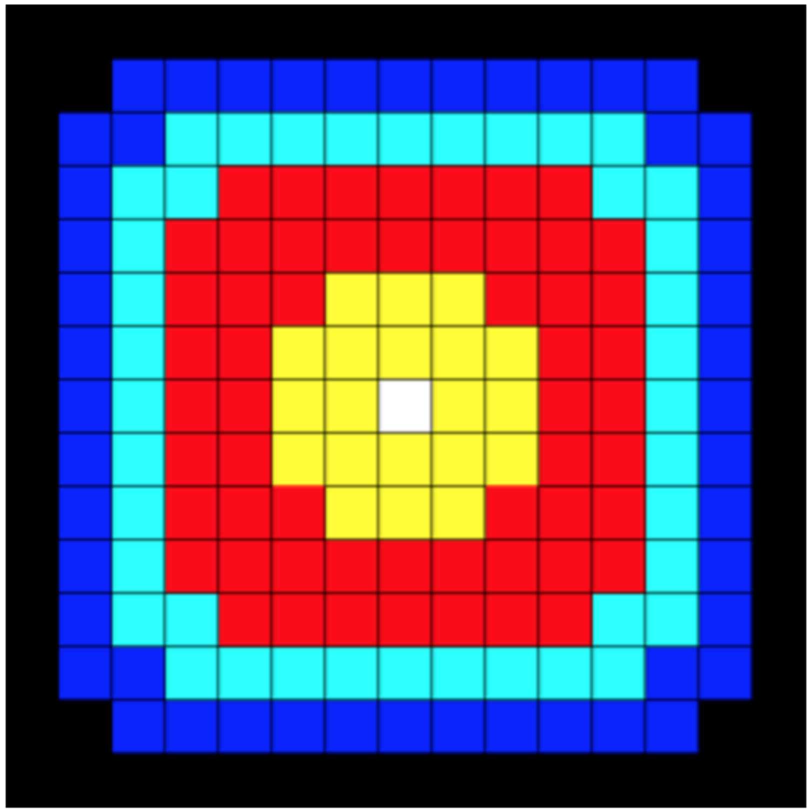
$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Example – 2D heat equation

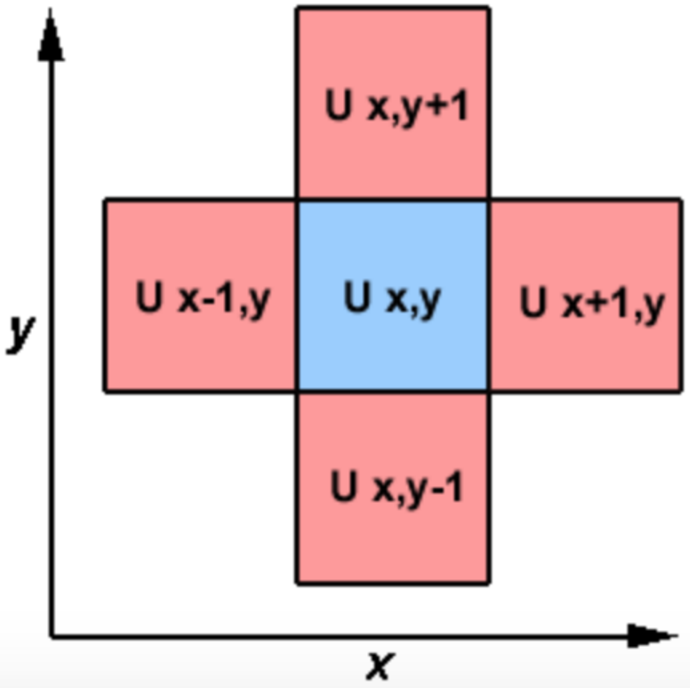
- Calculation dependes upon neighboring grid points

$$
\begin{aligned}
U_{x,y} = U_{x,y} \\
&+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) \\
&+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})
\end{aligned}
$$



- Serial pseudo-code:

```
do iy = 2, ny - 1
  do ix = 2, nx - 1
    u2(ix, iy) =  u1(ix, iy)  +
        cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
        cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
  end do
end do
```



Example – 2D heat equation

- Modified pseudo-code:

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray
  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  # Perform time steps
  do t = 1, nsteps
    update time
    send neighbors my border info
    receive from neighbors their border info
    update my portion of solution array

  end do

  send MASTER results

endif
```