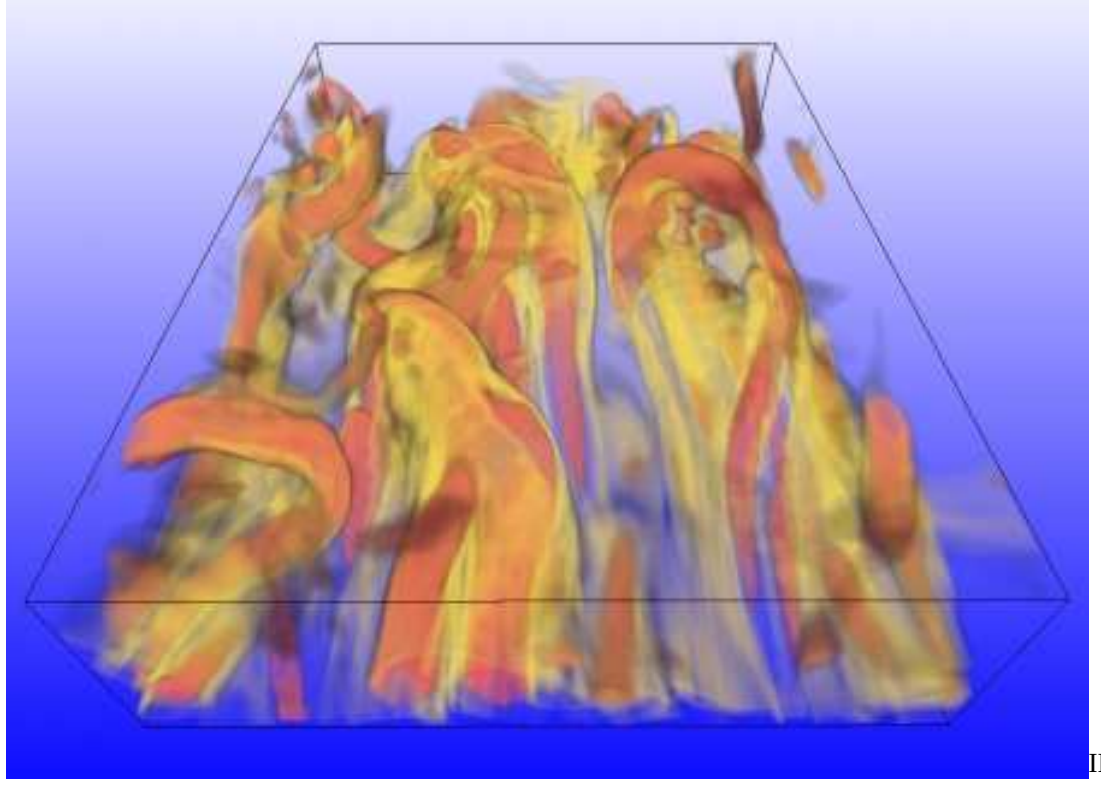
Parallel Computing

What will we learn today?

- Introduction to parallel computing
- Finding parallelism
- Parallel programming

Background (1)

- Increasingly sophisticated mathematical models
- Increasingly higher resolution Δx, Δy, Δz, Δt
- Increasingly longer computation time
- Increasingly larger memory requirement



Background (2)

Traditional serial computing (single processor) has limits

- Physical size of transistors
- Memory size and speed
- Instruction level parallelism is limited
- Power usage, heat problem

Moore’s law will not continue forever

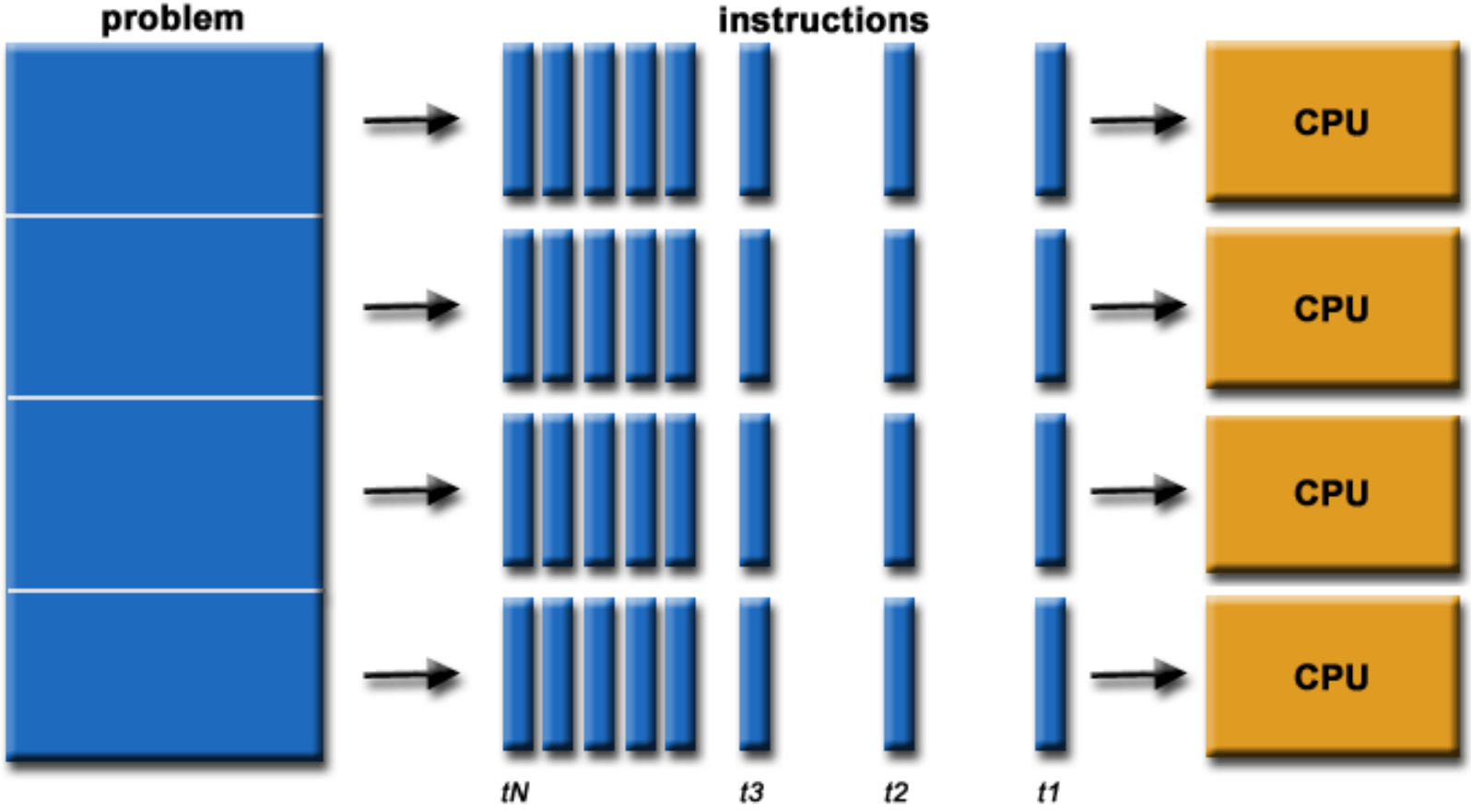
Background (3)

Parallel computing platforms are nowadays widely available

- Access to HPC centers
- Local Linux clusters
- Multiple multi-core CPUs in laptops
- GPUs (graphics processing units)

What is parallel computing?

Parallel computing: simultaneous use of multiple processing units to solve one computational problem

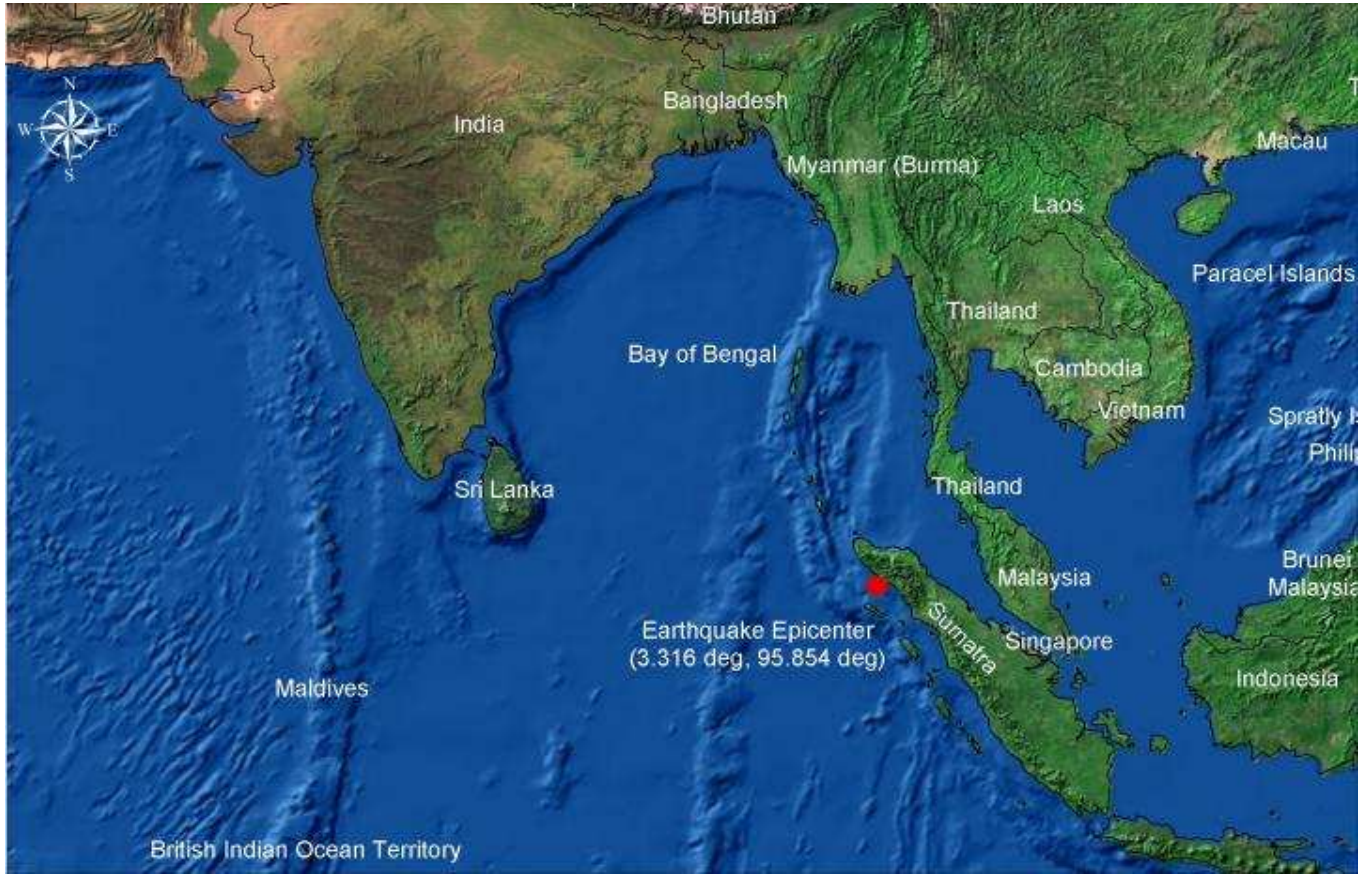


Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

Why parallel computing?

- Saving computation time
- Solving larger and more challenging problems
  - access to more memory
- Providing concurrency
- Saving cost

Example of Indian Ocean



- 1km$\times$1km resolution overall: about 40×10^6 mesh points
- 200m$\times$200m resolution overall: 10^9 mesh points

Example of Indian Ocean (cont'd)

Suppose we solve a 2D shallow-water wave equation

$$\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (gH(x,y)\nabla u)$$

over the Indian Ocean, using finite differences

- Four 2D arrays are needed: $u_{i,j}^{\ell+1}$, $u_{i,j}^{\ell}$, $u_{i,j}^{\ell-1}$, $H_{i,j}$
- Using double-precision (each value needs 8 bytes)
  - 40×10^6 mesh points $\rightarrow 4 \times 40 \times 10^6 \times 8 = 1.28$ GB memory needed
  - 10^9 mesh points $\rightarrow 32$ GB memory needed $\rightarrow$ too large for a regular computer
- Parallel computing necessary also because of the amount of floating-point operations

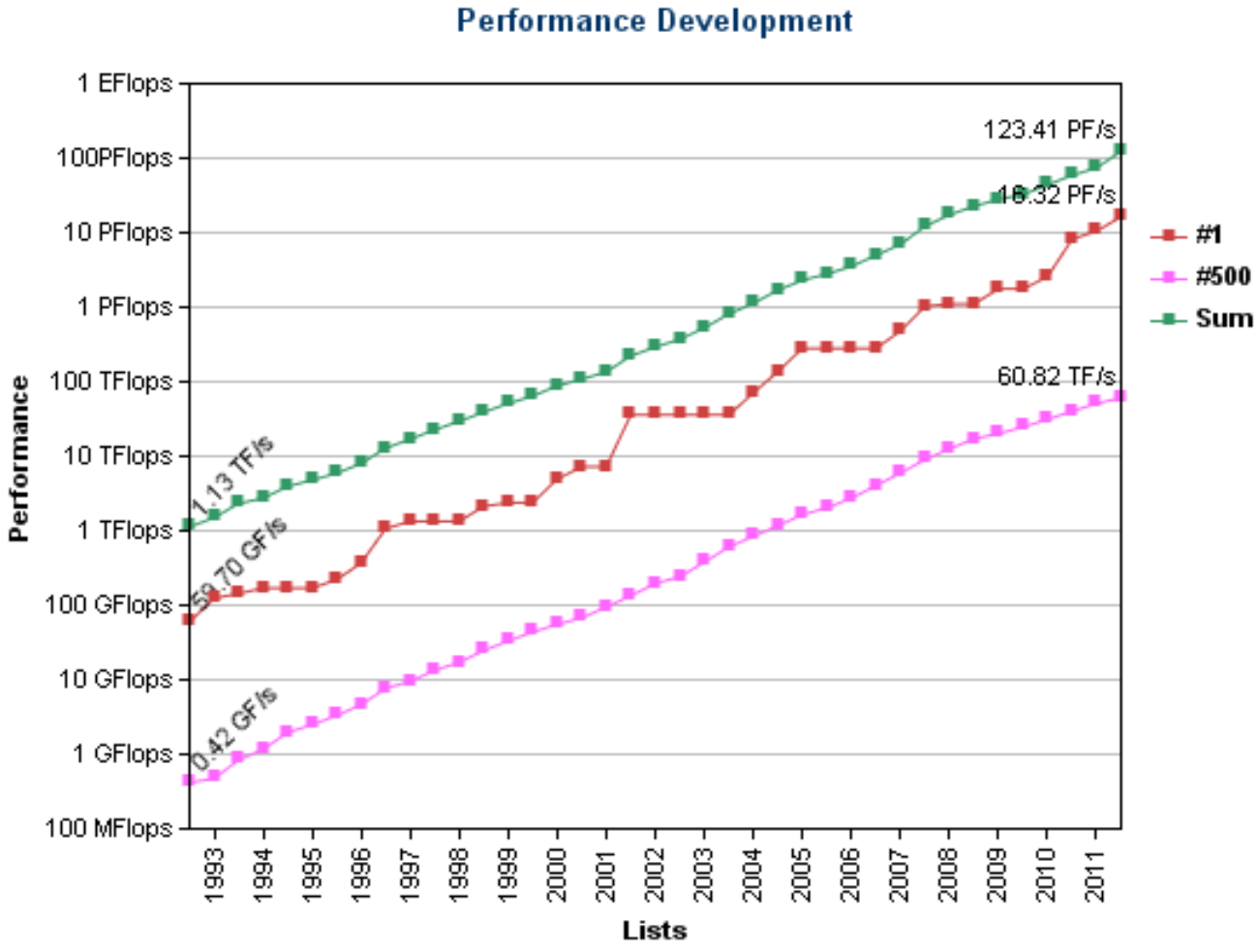
Today's most powerful computer

- IBM BlueGene/Q system at Lawrence Livermore Lab
- 1,572,864 CPU cores
- Theoretical peak performance: 20.13 petaFLOPS (20.13×10^{15} floating-point operations per second)
- Linpack benchmark: 16.32 petaFLOPS

Top 5 supercomputers (June 2012)

Rank	Name	Location	Peak	Linpack
1	Sequoia	Lawrence Livermore	20.132	16.325
2	K computer	RIKEN, Japan	11.280	10.510
3	Mira	Argonne	10.066	8.162
4	SuperMUC	Leibniz, Germany	3.185	2.897
5	Tianhe-1A	Tianjin, China	4.701	2.566

Top500 list (June 2012)



http://www.top500.org

Flynn's taxonomy

Classification of computer architectures:

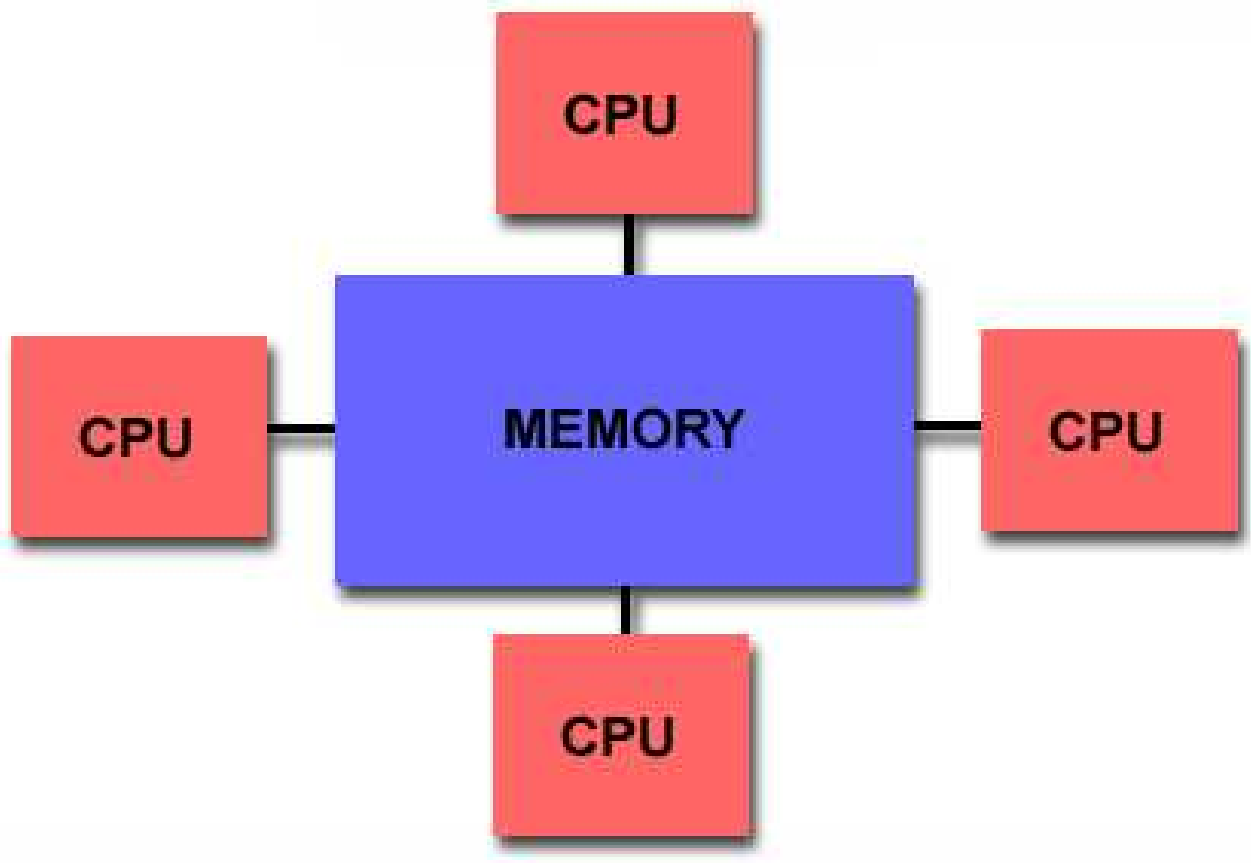
- SISD (single instruction, single data) – serial computers
- SIMD (single instruction, multiple data) – array computers, vector computers, GPUs
- MISD (mulitple instruction, single data) – systolic array (very rare)
- MIMD (mulitple instruction, multiple data) – mainstream parallel computers

Classification of parallel computers

From the memory perspective:

- Shared-memory systems
  - A single global address space
  - SMP – (symmetric multiprocessing)
  - NUMA – (non-uniform memory access)
  - Multi-core processor – CMP (chip multi-processing)
- Distributed-memory systems
  - Each node has its own physical memory
  - Massively parallel systems
  - Different types of clusters
- Hybrid distributed-shared memory systems

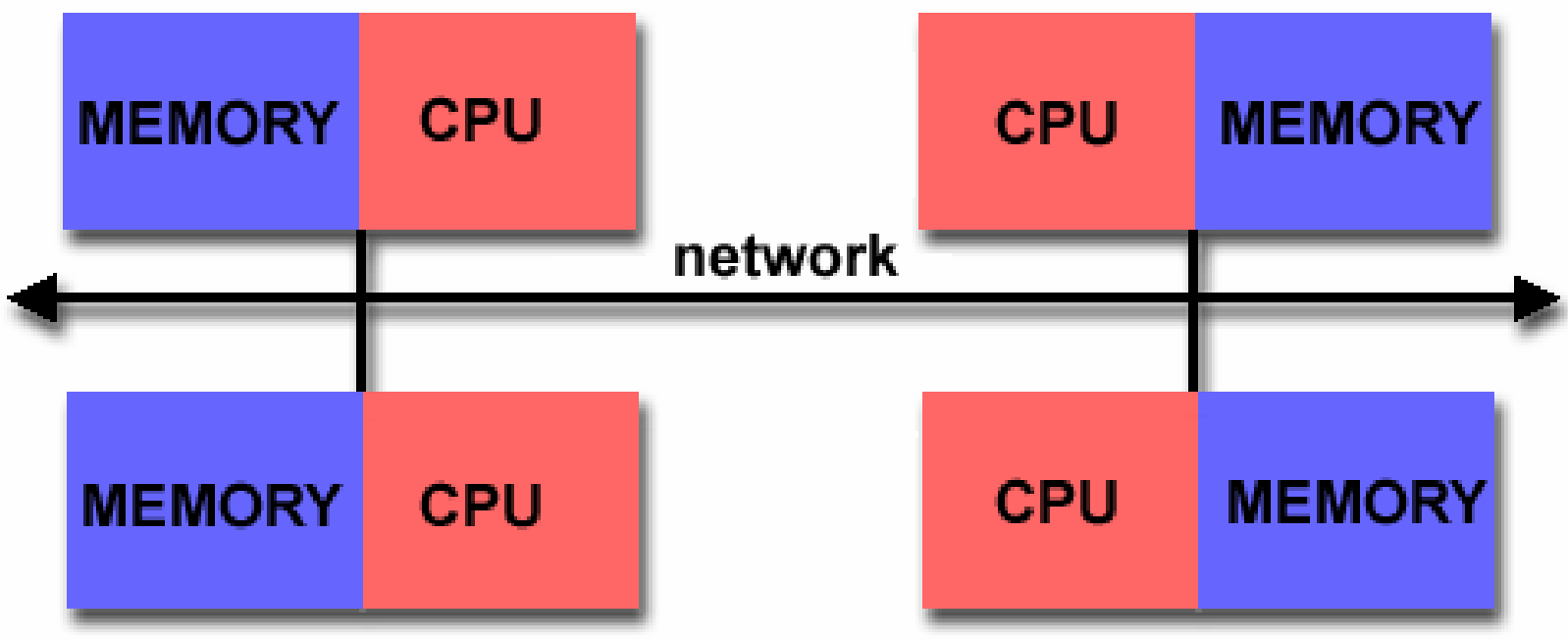
Shared memory



- Advantage: user-friendly
- Disadvantage: poor scalability

Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

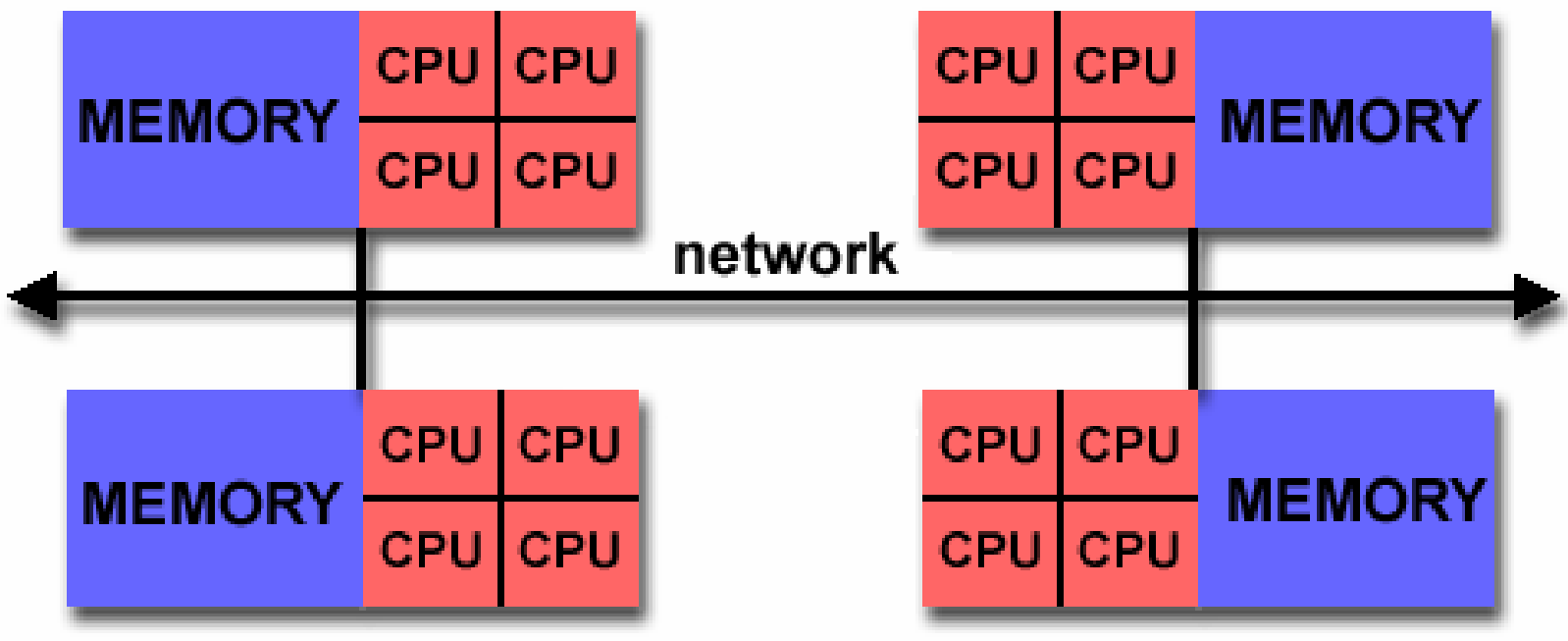
Distributed memory



- Advantages: data locality (no interference), cost-effective
- Disadvantages: explicit communication, explicit decomposition of data or tasks

Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

Hybrid distributed-shared memory



Plot obtained from `https://computing.llnl.gov/tutorials/parallel_comp/`

Finding parallelism

- Parallelism: Some work of a computational problem can be divided into a number of simultaneously computable pieces
- Applicability of parallel computing depends on the existence of parallelism
  - No parallelism $\rightarrow$ no use of parallel computers
- Parallelism can exist in different forms

Example 1

The *axpy* operation involves two vectors:

$$\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$$

- Computing y_i can be done totally independently of y_j
- The entries of $\mathbf{y}$ can be computed simultaneously
- Suppose the length of $\mathbf{y}$ is n, we can employ n workers, each computing a single entry
- Embarrassingly parallel

Example 2

Dot-product between two vectors:

$$d = \mathbf{x} \cdot \mathbf{y} := x_1 y_1 + x_2 y_2 + \ldots + x_n y_n.$$

Can we also employ n workers to do the computational work?

- At a first glance, parallelism is not obvious
- However, if we temporally introduce an assistant vector $\mathbf{d}$, such that $d_i = x_i y_i$, then each worker can independently compute one entry of $\mathbf{d}$

Example 2 (cont'd)

But what about the remaining computational work?

$$d = 0, \quad d \leftarrow d + d_i \quad \text{for } i = 1, 2, \ldots, n$$

- Now, the n workers need to collaborate!
- Let each even-ID worker k give its computed d_k value to worker $k-1$, who does $d_{k-1} + d_k$
- Then, all the even-ID workers retire and let the remaining workers repeat the above step, until there is only one worker left
- The solely surviving worker has the correctly computed value of d

Parallel reduction

- Parallel reduction: Using n workers to carry out similar computations such as

$$d = 0, \quad d \leftarrow d + d_i \quad \text{for } i = 1, 2, \ldots, n$$

- $\lceil \log_2 n \rceil$ stages are needed
  - During each stage, two and two workers collaborate
- It is seemingly much faster than the original serial operation, which has n stages
  - However, collaboration means additional time usage—overhead

Example 2 revisited

What if we employ m workers, where $m < n$?

- Each worker is responsible for several entries of $\mathbf{d}$
- First, each worker independently does a local summation over its assigned entries of $\mathbf{d}$
- Then, the m workers carry out a parallel reduction
- Very important that the workers are assigned with (roughly) the same number of entries of $\mathbf{d}$—load balance
  - Even if n is not a multiple of m, a fair work division makes the heaviest and lightest loaded workers only differ by one entry

Example 3

Matrix-vector multiply

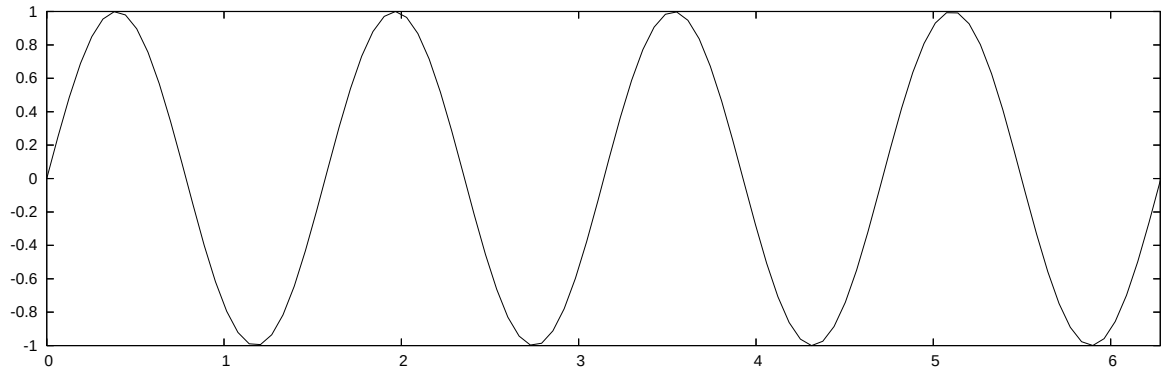
$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

where $\mathbf{A}$ is a $n \times n$ matrix, and $y_i = \sum_{j=1}^{n} A_{ij} x_j$

- Suppose n workers are employed
- Division of work with respect to the rows of $\mathbf{A}$
  - Each worker computes one entry of $\mathbf{y}$
  - Each worker makes use of the entire $\mathbf{x}$ vector
- Division of work with respect to the columns of $\mathbf{A}$
  - Each worker uses only one entry of $\mathbf{x}$
  - However, parallel reduction is needed to compute each entry of $\mathbf{y}$
- Actually, we can employ as many as n^2 workers

Example 4

1D standard wave equation



$$\frac{\partial^2 u}{\partial t^2} = \gamma^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0,1),\ t > 0,$$

$$u(0,t) = U_L,$$

$$u(1,t) = U_R,$$

$$u(x,0) = f(x),$$

$$\frac{\partial}{\partial t} u(x,0) = 0\,.$$

Example 4 (cont'd)

Finite difference discretization (with n interior mesh points):

$$u_i^0 = f(x_i), \quad i = 0, \ldots, n+1,$$

$$u_i^{-1} = u_i^0 + \frac{1}{2}C^2(u_{i+1}^0 - 2u_i^0 + u_{i-1}^0), \quad i = 1, \ldots, n$$

$$u_i^{k+1} = 2u_i^k - u_i^{k-1} + C^2(u_{i+1}^k - 2u_i^k + u_{i-1}^k),$$

$$i = 1, \ldots, n, \; k \geq 0,$$

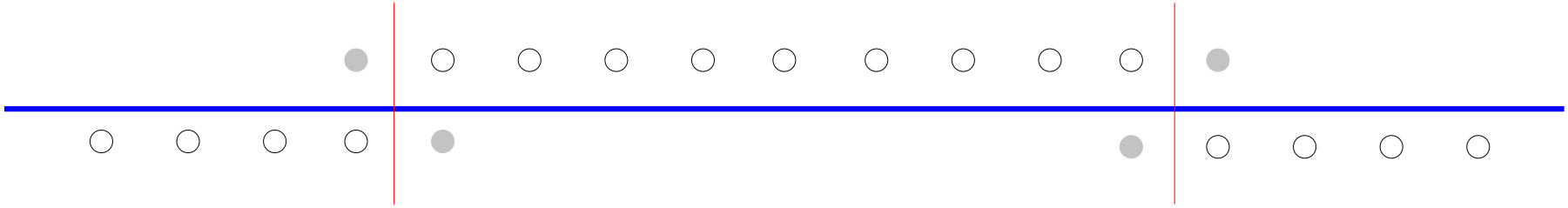
$$u_0^{k+1} = U_L, \quad k \geq 0,$$

$$u_{n+1}^{k+1} = U_R, \quad k \geq 0.$$

$C = \gamma \Delta t / \Delta x$

Example 4 (cont'd)

Each worker responsible for a sub-interval of the domain



- The spatial domain is divided
- Each worker only updates the values of u^{k+1} on its assigned mesh points
- Coordination is needed: A worker cannot go to the next time level, unless both its left and right neighbors have finished the current time level

Example 5

Finite differences for 2D wave equation

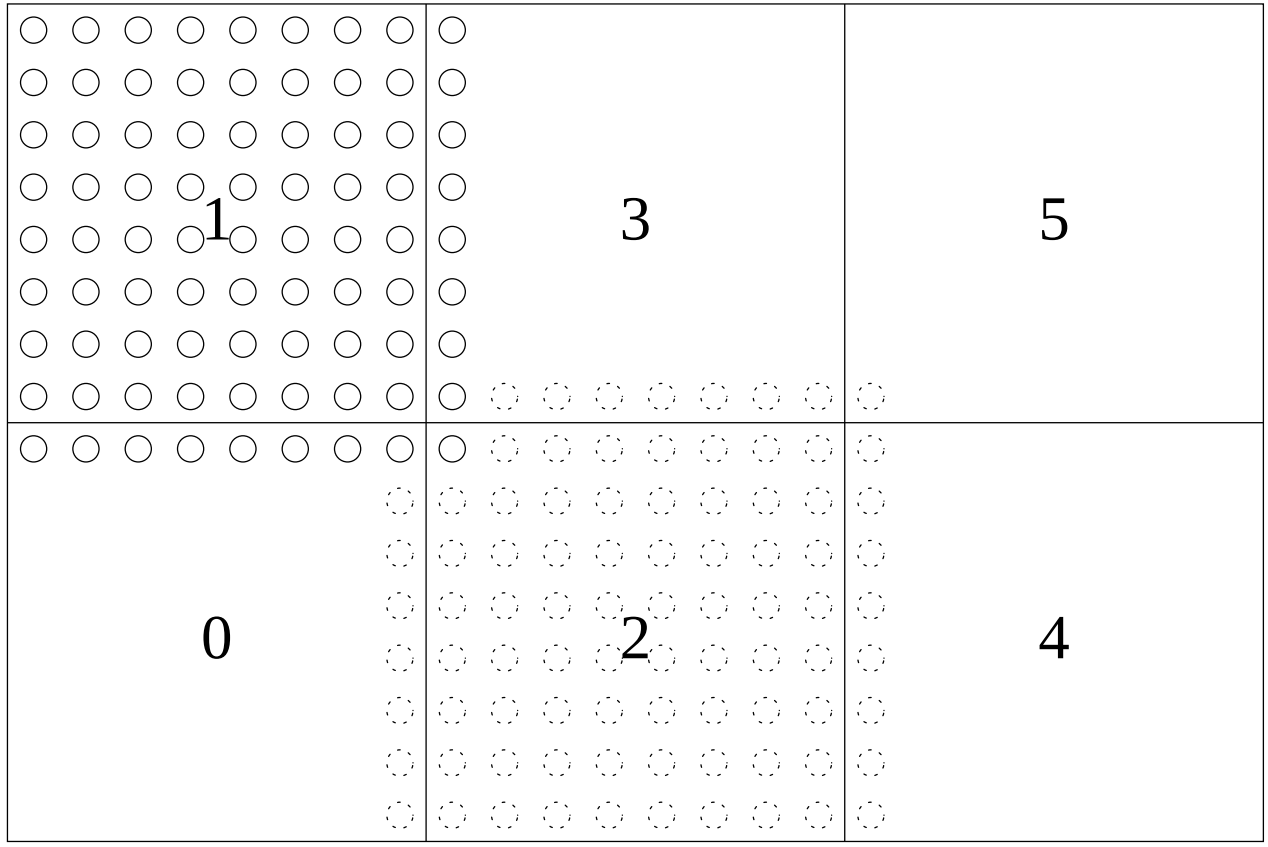
- An explicit numerical scheme (point-wise update):

$$u_{i,j}^{k+1} = S(u_{i,j\pm1}^{k}, u_{i\pm1,j}^{k}, u_{i,j}^{k}, u_{i,j}^{k-1}, x_{i,j}, t_k)$$

- Can compute all new $u_{i,j}^{k+1}$ values simultaneously
- Each worker is responsible for a rectangular region
- Before moving onto a new time level, workers need coordination

Example 5 (cont'd)

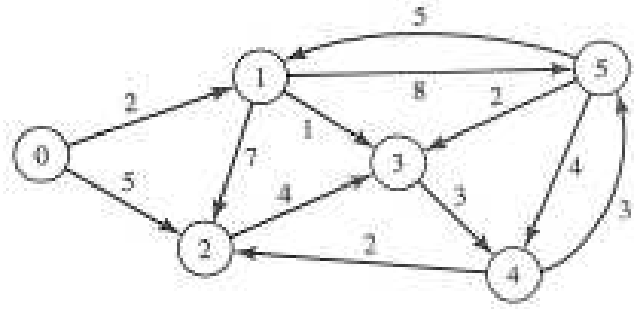
Example of work division



Example 6

Floyd's algorithm: finding the shortest paths

- Starting point: a graph of vertices and weighted edges



- Each edge is of a direction and has a length
  - if there's path from vertex i to j, there may not be path from vertex j to i
  - path length from vertex i to j may be different than path length from vertex j to i
- Objective: finding the shortest path between every pair of vertices $(i \rightarrow j)$

Example 6 (cont'd)

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for $k \leftarrow 0$ to $n-1$
  for $i \leftarrow 0$ to $n-1$
    for $j \leftarrow 0$ to $n-1$
      $a[i,j] \leftarrow \min(a[i,j],\ a[i,k]+a[k,j])$
    endfor
  endfor
endfor

Example 6 (cont'd)

- Inside the *k*'th iteration

```
for (i=0; i<n; i++)
  for (j=0; j<n, j++)
    a[i][j] = MIN(a[i][j], a[i][k]+a[k][j]);
```

- Can all the entries in a be updated concurrently?
- Yes, because the *k*'th column and the *k*'th row will not change during the *k*'th iteration!
  - Note that `a[i][k]=MIN(a[i][k],a[i][k]+a[k][k])` will be the same as `a[i][k]`
  - Note that `a[k][j]=MIN(a[k][j],a[k][k]+a[k][j])` will be the same as `a[k][j]`

Remarks so far

- For different computational problems, parallelism may exist in different forms
- For a same computational problem, parallelism may exist on different levels
- Finding parallelism (as much as possible) may not be straightforward
- However, once parallelism is identified, parallel computing becomes possible
  - Also need to understand the required collaboration between workers
- Parallel programming is the next big step

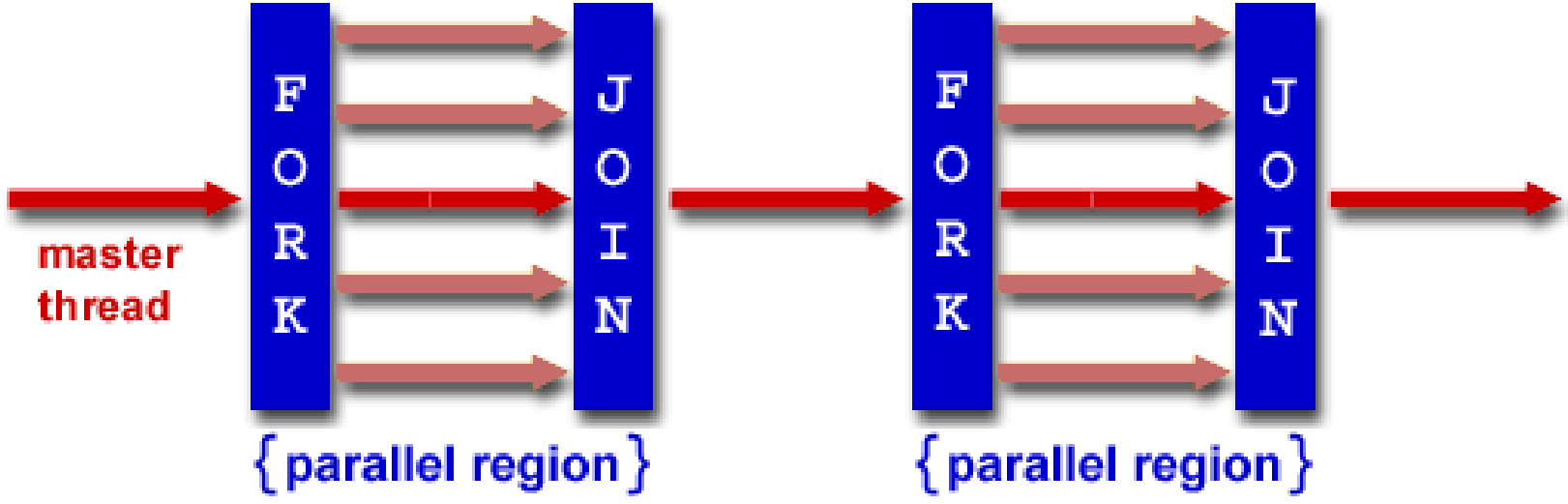
Parallel programming models

- Threads model
  - Easy to program (such as OpenMP)
  - Difficult to scale to many CPUs (NUMA, cache coherence)
- Message-passing model
  - Many programming details (MPI or PVM)
  - Better user control (data & work decomposition)
  - Larger systems and better performance
- Stream-based programming (for using GPUs)
- Hybrid parallel programming

OpenMP programming

OpenMP is a portable API for programming shared-memory computers

- Existence of multiple threads
- Use of compiler directives
- Fork-join model



Plot obtained from `https://computing.llnl.gov/tutorials/openMP/`

OpenMP example

Dot-product between two vectors $\mathbf{x}$ and $\mathbf{y}$:

$$d = \mathbf{x} \cdot \mathbf{y} := x_1 y_1 + x_2 y_2 + \ldots + x_n y_n.$$

```
d = 0.0;

#pragma omp parallel for      \
  default(shared) private(i) schedule(static,chunk) reduction(+:d)

  for (i=0; i < n; i++)
    d = d + (x[i] * y[i]);
```

MPI programming

MPI (message passing interface) is a library standard

- Implementation(s) of MPI available on almost every major parallel platform
- Portability, good performance & functionality
- Each process has its local memory
- Explicit message passing enables information exchange and collaboration between processes

More info: `http://www-unix.mcs.anl.gov/mpi/`

MPI example

Dot-product between two vectors: $d = \sum_{i=1}^{n} x_i y_i$

```
MPI_Comm_size (MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

my_start = n*my_rank/num_procs;
my_stop = n*(my_rank+1)/num_procs;

my_d = 0.;
for (i=my_start; i<my_stop; i++)
  my_d = my_d + (x[i] * y[i]);

MPI_Allreduce (&my_d, &d, 1, MPI_DOUBLE,
               MPI_SUM, MPI_COMM_WORLD);
```

In this example, we've assumed that both $\mathbf{x}$ and $\mathbf{y}$ are duplicated on all MPI processes

Data decomposition

- If an MPI process only uses a subset of the entire data structure, data decomposition should be done
  - Otherwise, data duplication will be a killing factor
- Very often, neighboring MPI processes have some overlap in their “data footprints”
  - Need to distinguish the computational responsibility from data footprint
  - Ghost points (halo points) are usually part of the local data structure of an MPI process

Solving 1D wave equation; revisited

$$\frac{\partial^2 u}{\partial t^2} = \gamma^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < 1$$

- Uniform mesh in x-direction: $n+2$ points, $\Delta x = \frac{1}{n+1}$
  - x_0 is left boundary point, x_{n+1} is right boundary point
  - $x_1, x_2, \ldots, x_n$ are interior points
- Notation: $u_i^\ell \approx u(i\Delta x, \ell \Delta t)$
- $\frac{\partial^2 u}{\partial t^2} \approx \frac{1}{\Delta t^2}\left(u_i^{\ell+1} - 2u_i^\ell + u_i^{\ell-1}\right)$
- $\frac{\partial^2 u}{\partial x^2} \approx \frac{1}{\Delta x^2}\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right)$
- Overall numerical scheme:

$$u_i^{\ell+1} = 2u_i^\ell - u_i^{\ell-1} + \gamma^2 \frac{\Delta t^2}{\Delta x^2}\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right) \quad i = 1, 2, \ldots, n$$

Revisit continues (1)

Serial implementation

- Three 1D arrays are needed:
  - $u^{\ell+1}$: `double *up=(double*)malloc((n+2)*sizeof(double));`
  - u^{ℓ}: `double *u=(double*)malloc((n+2)*sizeof(double));`
  - $u^{\ell-1}$: `double *um=(double*)malloc((n+2)*sizeof(double));`
- A `while`-loop for doing the time steps
- At each time step, a `for`-loop for updating the interior points

Revisit continues (2)

Main time loop:

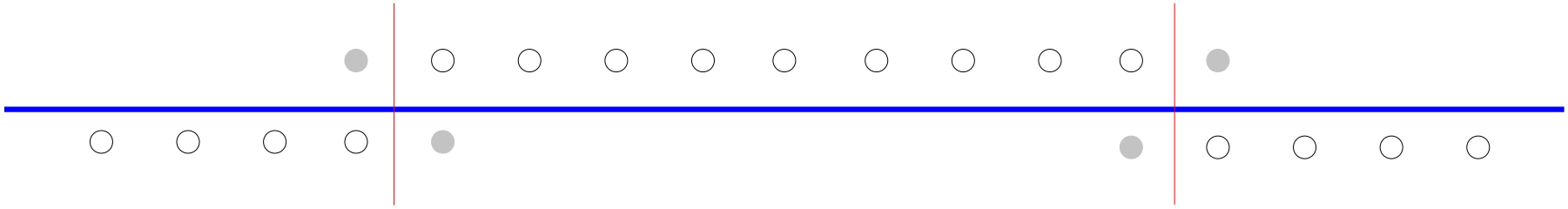
```
while (t<T){
  t += dt;
  for (i=1; i<=n; i++)
    up[i] = 2*u[i]-um[i]+C*(u[i-1]-2*u[i]+u[i+1]);
  up[0] = value_of_left_BC(t);   // enforcing left BC
  up[n+1] = value_of_rigt_BC(t); // enforcing right BC

  /* preparation for next time step: shuffle the three arrays */
  tmp = um;
  um = u;
  u = up;
  up = tmp;
}
```

MPI for 1D wave equation

MPI parallelization starts with work division

- The global domain is decomposed into P subdomains
  - Actually, the n interior points are divided, due to the chosen Dirichlet boundary conditions
  - In case of Neumann boundary conditions, the $n+2$ points are to be divided



MPI for 1D wave equation (cont'd)

- Each subdomain has n/P interior points, plus two "ghost points"

```
int n_local = n/P;   // assume that n is divisible by P
double *up_local=(double*)malloc((n_local+2)*sizeof(double));
double *u_local=(double*)malloc((n_local+2)*sizeof(double));
double *um_local=(double*)malloc((n_local+2)*sizeof(double));
```

  - If there is a neighbor subdomain to the side, the value of the ghost point is to be provided
  - Otherwise, the ghost point is actually a physical boundary point

MPI for 1D wave equation (cont'd)

Parallel implementation using MPI

- First, `up_local[i]` is computed on each interior point `i=1,2,...,n_local`
- If there's neighbor on the left,
  - send `up_local[1]` to the left neighbor
  - receive `up_local[0]` from the left neighbor
- If there's neighbor on the left,
  - send `up_local[n_local]` to the right neighbor
  - receive `up_local[n_local+1]` from the right neighbor

MPI for 1D wave equation (cont'd)

Overlapping communication with computation

- `up_local[1]` is computed first
- Initiate communication with the left neighbor using `MPI_Isend` and `MPI_Irecv`
- `up_local[M_local]` is then computed
- Initiate communication with the right neighbor using `MPI_Isend` and `MPI_Irecv`
- Afterward, main local computation over indices `i=2,3,...,n_local-1`
- Finally, finish communication with left neighbor using `MPI_Wait`
- Finally, finish communication with right neighbor using `MPI_Wait`

What about 2D wave equation?

- In 2D, each subdomain is a rectangle
- One layer of ghost points around
- Each MPI process has (at most) four neighbors
  - Four outgoing messages
  - Four incoming messages
- Each pair of neighbors exchange a 1D array in between

Recap of parallelization

- Identify the parts of a serial code that have concurrency
- Be aware of inhibitors to parallelism (e.g. data dependency)
- When using OpenMP
  - insert directives to create parallel regions
- When using MPI
  - decide an explicit decomposition of tasks and/or data
  - insert MPI calls

Parallel programming requires a new way of thinking

Some useful concepts

- Cost model of sending a message $t_C(L) = \tau + \beta L$
- Speed-up

$$S(P) = \frac{T(1)}{T(P)}$$

- Parallel efficiency

$$\eta(P) = \frac{S(P)}{P}$$

- Factors of parallel inefficiency
  - communication, synchronization
  - load imbalance
  - additional calculations due to parallelization
  - non-parallelizable sections

Amdahl's law

The upper limit of speedup

$$\frac{T(1)}{T(P)} \leq \frac{T(1)}{(f_s + \frac{f_p}{P})T(1)} = \frac{1}{f_s + \frac{1-f_s}{P}} < \frac{1}{f_s}$$

- f_s – fraction of code that is serial (not parallelizable)
- f_p – fraction of code parallelizable: $f_p = 1 - f_s$

Gustafson–Barsis's law

Things are normally not so bad as Amdahl's law says

- Normalize the parallel execution time to be 1
- Scaled speed-up

$$S_s(P) = \frac{f_s + P f_p}{f_s + f_p} = f_s + P(1 - f_s) = P + (1 - P) f_s$$

- f_s has a different meaning than Amdahl's law
- f_s normally decreases as the problem size grows
- Encouraging to solve larger problems with larger P

Granularity

Granularity is a qualitative measure of the ratio of computation over communication

- Fine-grain parallelism
  - small amounts of computation between communication
  - load imbalance may be a less important issue
- Coarse-grain parallelism
  - large amounts of computation between communication
  - high ratio of computation over communication

Objective: Design coarse-grain parallel algorithms, if possible

Summary

- We’re already at the age of parallel computing
- Parallel computing relies on parallel hardware
- Parallel computing needs parallel software
- So parallel programming is very important
  - new way of thinking
  - identification of parallelism
  - design of parallel algorithm
  - implementation can be a challenge