Simulation and High-Performance Computing

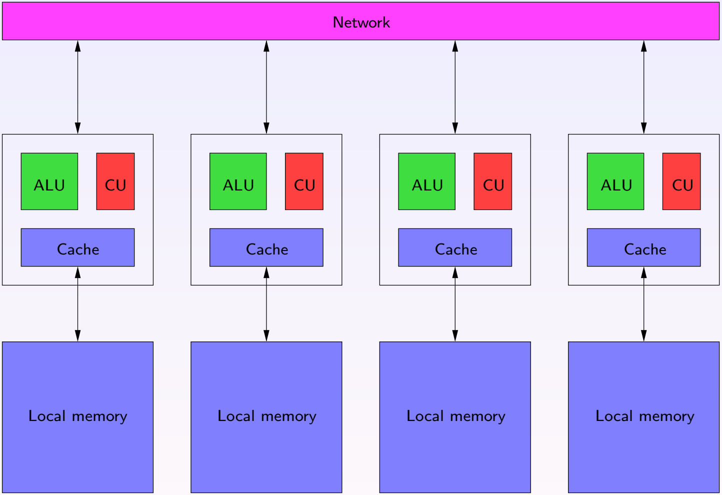
Part 17: Introduction to Distributed Computing

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 8th, 2020

Distributed computer



Distributed computing

Advantages:

- With powerful networking hardware, very large distributed computers are possible.
- Processors have direct access to local memory.
- Simpler structure compared to shared-memory systems, therefore lower costs.
- Nodes can be fairly cheap and easily replacable.

Distributed computing

Advantages:

- With powerful networking hardware, very large distributed computers are possible.
- Processors have direct access to local memory.
- Simpler structure compared to shared-memory systems, therefore lower costs.
- Nodes can be fairly cheap and easily replacable.

Disadvantages:

- Communication has to be performed explicitly.
- Communication networks are generally slower than shared memory.

Message Passing Interface

MPI is a well-established industry standard for programming distributed computers. It is managed by the MPI Forum.

While OpenMP and CUDA extend the C compiler, MPI essentially just adds a library of functions.

Compiling and running: Since in data centers the computer used for compiling a program and the computers used for runnning it may be entirely different, MPI programs are

- translated using a special compiler (frequently called `mpicc`), and
- run using special tools (on PCs `mpirun`, in data centers flexible batch processing systems).

MPI terminology

Processes are instances of a program running on a node of the distributed computer. Their address spaces are disjoint from those of other processes.

MPI terminology

Processes are instances of a program running on a node of the distributed computer. Their address spaces are disjoint from those of other processes.

Communicators provide a context for communication, e.g.,

- they assign a unique rank to each participating process, and
- they ensure that messages are received in the order they are sent.

Pre-defined communicators are `MPI_COMM_WORLD` and `MPI_COMM_SELF`.

MPI terminology

Processes are instances of a program running on a node of the distributed computer. Their address spaces are disjoint from those of other processes.

Communicators provide a context for communication, e.g.,

- they assign a unique rank to each participating process, and
- they ensure that messages are received in the order they are sent.

Pre-defined communicators are `MPI_COMM_WORLD` and `MPI_COMM_SELF`.

Messages are used to move data around.

- Every message has a source and a destination, given as ranks with respect to a communicator.
- Every message is equipped with a tag that can be used to distinguish between different messages.
- Every message contains a number of elements of a given type.

Example: Sending a message

```
int
main(int argc, char **argv)
{
  char buf[6];
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0)
    MPI_Send("Hello", 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  if(rank == 1) {
    MPI_Recv(buf, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("%s World\n", buf);
  }
  MPI_Finalize();
  return 0;
}
```

Basic MPI functions

- `MPI_Init` initializes the MPI system.
- `MPI_Finalize` exits the MPI system.
- `MPI_Comm_size` returns the number of processes in a communicator.
- `MPI_Comm_rank` returns the process's rank within a communicator.
  Ranks are contiguous, starting at zero.
- `MPI_Send` sends a message to a process.
- `MPI_Recv` receives a message from a process.

Send and receive

```
int
MPI_Send(const void *buf, int count, MPI_Datatype datatype,
         int dst, int tag, MPI_Comm comm);

int
MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int src, int tag, MPI_Comm comm,
         MPI_Status *status);
```

- `buf` points to the send or receive buffer.
- `count` gives the number of elements.
- `datatype` defines the element type, e.g., `MPI_INT`, `MPI_FLOAT`.
- `dst` and `src` are the ranks of destination and source processes.
- `tag` is an additional tag. The tags of send and receive operations have to match. `MPI_ANY_TAG` matches any tag when receiving.
- `comm` is the communicator for this data exchange.

Blocking communication

Problem: `MPI_Send` may block, i.e., the program may wait until a matching receive operation has started or even completed.
`MPI_Recv` always blocks in order to ensure that the buffer is filled with meaningful data.

```
if(rank == 0) {
  MPI_Send(&out, 1, MPI_FLOAT, 1, 0, comm);
  MPI_Recv(&in,  1, MPI_FLOAT, 1, 0, comm, MPI_STATUS_IGNORE);
}
if(rank == 1) {
  MPI_Send(&out, 1, MPI_FLOAT, 0, 0, comm);
  MPI_Recv(&in,  1, MPI_FLOAT, 0, 0, comm, MPI_STATUS_IGNORE);
}
```

Deadlock: If `MPI_Send` is blocking, this program will wait forever, since no process can start a receive operation.

Non-blocking communication

```
int
MPI_Isend(const void *buf, int count, MPI_Datatype datatype,
          int dest, int tag, MPI_Comm comm,
          MPI_Request *request);

int
MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
          int source, int tag, MPI_Comm, comm,
          MPI_Request *request);

int
MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Idea: Non-blocking functions return an `MPI_Request` that can be used to wait for their completion.

Example: Avoiding deadlocks

```
MPI_Request r_out, r_in;

if(rank == 0) {
  MPI_Isend(&out, 1, MPI_FLOAT, 1, 0, comm, &r_out);
  MPI_Irecv(&in,  1, MPI_FLOAT, 1, 0, comm, &r_in);
  MPI_Wait(&r_out, MPI_STATUS_IGNORE);
  MPI_Wait(&r_in,  MPI_STATUS_IGNORE);
}
if(rank == 1) {
  MPI_Isend(&out, 1, MPI_FLOAT, 0, 0, comm, &r_out);
  MPI_Irecv(&in,  1, MPI_FLOAT, 0, 0, comm, &r_in);
  MPI_Wait(&r_out, MPI_STATUS_IGNORE);
  MPI_Wait(&r_in,  MPI_STATUS_IGNORE);
}
```

Buffers may only be used once the operation has completed, i.e., after returning from `MPI_Wait`.

Example: One-dimensional wave equation

Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative. Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Example: One-dimensional wave equation

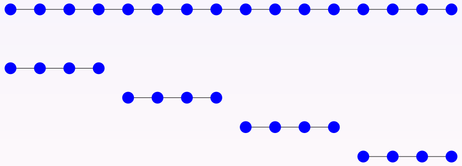
Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative.
Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Domain decomposition: Split mesh among several processes.



Example: One-dimensional wave equation

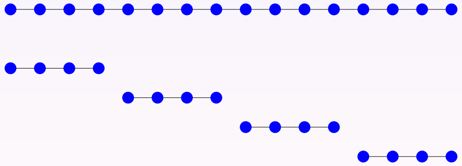
Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative. Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Domain decomposition: Split mesh among several processes.



Example: One-dimensional wave equation

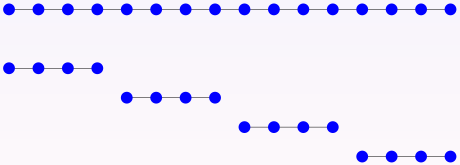
Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative.
Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Domain decomposition: Split mesh among several processes.



Example: One-dimensional wave equation

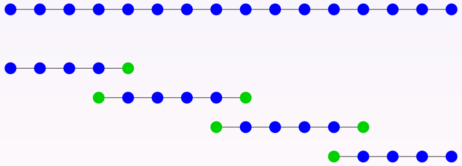
Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative.
Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Domain decomposition: Split mesh among several processes.



Example: One-dimensional wave equation

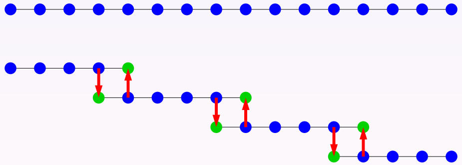
Goal: Solve the partial differential equation

$$\frac{\partial u}{\partial t}(t,x) = v(t,x), \qquad \frac{\partial v}{\partial t}(t,x) = c\frac{\partial^2 u}{\partial^2 x}(t,x)$$

Approach: Difference quotient with meshwidth h for spatial derivative. Leapfrog with stepsize δ for timestepping.

$$\tilde{u}_i \leftarrow \tilde{u}_i + \delta\,\tilde{v}_i, \qquad \tilde{v}_i \leftarrow \tilde{v}_i + \delta\,\tfrac{c}{h^2}(\tilde{u}_{i-1} - 2\tilde{u}_i + \tilde{u}_{i+1})$$

Domain decomposition: Split mesh among several processes.



Implementation: Wave equation

```
for(i=1; i<=n; i++)
  x[i] += delta * v[i];

if(rank > 0) {
  MPI_Isend(x+1, 1, MPI_DOUBLE, rank-1, 0, comm, &sl);
  MPI_Irecv(x,   1, MPI_DOUBLE, rank-1, 0, comm, &rl);
}
else
  sl = rl = MPI_REQUEST_NULL;

if(rank+1 < size) ...

MPI_Wait(&sl, MPI_STATUS_IGNORE);
...

for(i=1; i<=n; i++)
  v[i] += delta * ch2 * (x[i-1] - 2.0 * x[i] + x[i+1]);
```

Hiding communication latencies

Idea: While data is transmitted, we can carry out computations.

Wave equation: While we are waiting for `x[0]` and `x[n+1]`, we can already compute `v[2]` to `v[n-1]`.

```
if(rank > 0)
  /* start send and receive left */
if(rank+1 < size)
  /* start send and receive right */

for(i=2; i<n; i++)
  v[i] += delta * ch2 * (x[i-1] - 2.0 * x[i] + x[i+1]);

MPI_Wait(&sl, MPI_STATUS_IGNORE);
...

v[1] += delta * ch2 * (x[0] - 2.0 * x[1] + x[2]);
v[n] += delta * ch2 * (x[n-1] - 2.0 * x[n] + x[n+1]);
```

Summary

MPI allows us to implement programs that run across multiple connected computer nodes.

Messages containing arrays of data are used to exchange information between processes running on the nodes.

Communicators provide context for communication between nodes.

Non-blocking communication allows us to avoid deadlocks and hide communication latencies.