Lecture 7: More about MPI programming

Some recaps (1)

- One way of categorizing parallel computers is by looking at the memory configuration:
  - In *shared-memory systems* the CPUs share the same address space. Any CPU can access any data in the global memory.
  - In *distributed-memory systems* each CPU has its own memory. The CPUs are connected by some network and can exchange messages.
- Example 1: The processor cores within a multicore PC have shared memory
- Example 2: A cluster of multicore PCs is a distributed shared-memory system

Some recaps (2)

- Distributed memory gives rise to the programming paradigm of *message passing*
- Message-passing – all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the sub-problems to the processors.
- Collaboration between processors is achieved by explicit message passing, which has the purpose of data transfer and synchronization.

Some recaps (3)

- The message passing paradigm is very general
  - Shared-memory systems can also use message passing for programming
- By using message passing, the programmer has full control, but also has full responsibility
- Appropriate use of message passing commands is essential for achieving good performance on distributed-memory systems, sometimes also on shared-memory systems
- Message-passing programming is often non-trivial, due to the many exposed details
- MPI is the de-facto standard of message-passing programming

Some recaps (4)

- Shared-memory programming assumes a global memory address space
- Creation of child processes (also called threads)
  - – static (at beginning of program execution)
  - – dynamic (*fork* and *join*)
- Coordination among threads by three types of primitives:
  - specifying variables that can be accessed by all threads
  - preventing threads from improperly accessing shared resources
  - providing a means for synchronizing the threads
- More on share-memory programming (OpenMP) later in this course

Some recaps (5)

- SPMD – *single program multiple data*
- It suffices with a SPMD model for the message passing paradigm
- Same executable for all the processors
- There are typically conditional branches based on the processor id
- Each processor works primarily with its assigned local data
- Progression of code on different processors is relatively independent between synchronization points

Overhead present in parallel computing

- *Overhead of communication*
  - Latency and bandwidth — the cost model of send a message of length L between two processors:

$$t_C(L) = \tau + \beta L$$

- *Uneven load balance* $\rightarrow$ not all the processors can perform useful work at any time
- *Overhead of synchronization*
- *Extra computation due to parallelization*

Two very important concepts

- Speed-up

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

  - The larger the value of $S(P)$ the better

- Efficiency

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

  - The closer to 100% of $\eta(P)$ the better

Rules for point-to-point communication

- *Message order preservation* – If Process A sends two messages to Process B, which posts two matching receive calls, then the two messages are guaranteed to be received in the order they were sent.
- *Progress* – It is not possible for a matching send and receive pair to remain permanently outstanding. That is, if one process posts a send and a second process posts a matching receive, then either the send or the receive will eventually complete.

Probing in MPI

- It is possible in MPI to only read the envelope of a message before choosing whether or not to read the actual message.

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
              MPI_Status *status)
```

- The `MPI_Probe` function blocks until a message matching the given `source` and/or `tag` is available
- The result of probing is returned in an `MPI_Status` data structure

Example: sum of random numbers

```
int main (int nargs, char** args)
{
  int size, my_rank, i, a, sum;
  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  srand (7654321*(my_rank+1));
  a = rand()%100;

  if (my_rank==0) {
    MPI_Status status;
    sum = a;
    for (i=1; i<size; i++) {
      MPI_Probe (MPI_ANY_SOURCE,500,MPI_COMM_WORLD,&status);
      MPI_Recv (&a, 1, MPI_INT,
                status.MPI_SOURCE,500,MPI_COMM_WORLD,&status);
      sum += a;
    }
    printf("<%02d> sum=%d\n",my_rank,sum);
  }
  else
    MPI_Send (&a, 1, MPI_INT, 0, 500, MPI_COMM_WORLD);

  MPI_Finalize ();
  return 0;
}
```

Example of deadlock

- When a large message is sent from one process to another
  - and if there is insufficient OS storage at the destination, the send command must wait for the user to provide the memory space (through a receive command)
  - the following code is unsafe because it depends on the availability of system buffers

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

Solutions to deadlocks

- Order the send/receive calls more carefully
- Use `MPI_Sendrecv`
- Use `MPI_Bsend`
- Use non-blocking operations

Overlap communication with computation

- Performance may be improved on many systems by overlapping communication with computation. This is especially true on systems where communication can be executed autonomously by an intelligent communication controller.
- Use of non-blocking and completion routines allow computation and communication to be overlapped. (Not guaranteed, though.)

Non-blocking send

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

- The command returns “immediately”
- The message buffer should not be rewritten when the command returns
- Must check for local completion

Non-blocking receive

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

- The command returns “immediately”
- The message buffer should not be read yet
- Must check for local completion
- The use of nonblocking receives may also avoid system buffering and memory-to-memory copying, as information is provided early on the location of the receive buffer.

MPI_Request

- A request object identifies various properties of a communication operation
- A request object also stores information about the status of the pending communication operation

Local completion

- Two ways of checking on non-blocking sends and receives
  - `MPI_Wait` blocks until the communication is complete

    ```
    MPI_Wait(MPI_Request *request, MPI_Status *status)
    ```
  - `MPI_Test` returns “immediately”, and sets flag to true is the communication is complete

    ```
    MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
    ```

More about point-to-point communication

- When a standard mode blocking send call returns, the message data and envelope have been “safely stored away”. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
- MPI decides whether outgoing messages will be buffered. If MPI buffers outgoing messages, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. Then the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver.

Four modes of MPI’s send

- standard mode – a send may be initiated even if a matching receive has not been initiated
- buffered mode – similar to standard mode, but completion is always independent of matching receive, and message may be buffered to ensure this
- synchronous mode – a send will not complete until message delivery is guaranteed
- ready mode – a send may be initiated only if a matching receive has been initiated

Persistent communication requests

- Often a communication with the same argument list is repeatedly executed. MPI can bind the list of communication arguments to a persistent communication request once and, then, repeatedly use the request to initiate and complete messages
- Overhead reduction for communication between the process and communication controller
- Creation of a persistent communication request (before communication)

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

Persistent communication requests (cont´d)

- Initiation of a communication that uses a persistent request:

```
int MPI_Start(MPI_Request *request)
```

- A communication started with a call to `MPI_Start` can be completed by a call to `MPI_Wait` or `MPI_Test`
- The request becomes inactive after successful completion. The request is not deallocated and it can be activated anew by a `MPI_Start` call
- A persistent request is deallocated by a call to `MPI_Request_free`

Send-receive operations

- MPI send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process
- A send-receive operation is very useful for executing a shift operation across a chain of processes

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status)
```

Null process

- The special value `MPI_PROC_NULL` can be used to specify a “dummy” source or destination for communication
- This may simplify the code that is needed for dealing with boundaries, e.g., a non-circular shift done with calls to send-receive
- A communication with process `MPI_PROC_NULL` has no effect

Example:

```
int left_rank = my_rank-1, right_rank = my_rank+1;
if (my_rank==0)
  left_rank = MPI_PROC_NULL;
if (my_rank==size-1)
  right_rank = MPI_PROC_NULL;
```

MPI timer

```
double MPI_Wtime(void)
```

This function returns a number representing the number of wall-clock seconds elapsed since some time in the past.

Example usage:

```
double starttime, endtime;
starttime = MPI_Wtime();
/* ....  work to be timed  ... */
endtime   = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

Solving 1D wave equation (1)

Mathematical model

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}$$

- Spatial domain: $x \in (0, 1)$
- Temporal doamin: $0 < t \leq T$
- Boundary conditions: u is known at $x = 0$ and $x = 1$
- Initial conditions: $u(x, 0)$ is known, and “wave initially at rest”

Solving 1D wave equation (2)

Numerical method

- Uniform mesh in x-direction: $M+2$ points, $\Delta x = \frac{1}{M+1}$
  - x_0 is left boundary point, x_{M+1} is rightboundary point
  - $x_1, x_2, \ldots, x_M$ are interior points
- Time step size: Δt
- 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:

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

Solving 1D wave equation (3)

Enforcement of the initial conditions

- $u(x,0)$ is given, for example $u(x,0) = I(x)$

```
for (i=0; i<=M+1; i++) {
  x = i*dx;
  um[i] = I(x);
}
```

- We also need to compute u^1, because u^2 relies on both u^0 and u^1:

$$u_i^1 = u_i^0 + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i-1}^0 - 2u_i^0 + u_{i+1}^0\right) \quad i = 1, 2, \ldots, M$$

```
for (i=1; i<=M; i++)
  u[i] = um[i] +((dt*dt)/(2*dx*dx))*(um[i-1]-2*um[i]+um[i+1]);
u[0] = value_of_left_BC(dt);
u[M+1] = value_of_right_BC(dt);
```

Solving 1D wave equation (4)

Serial implementation

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

Solving 1D wave equation (5)

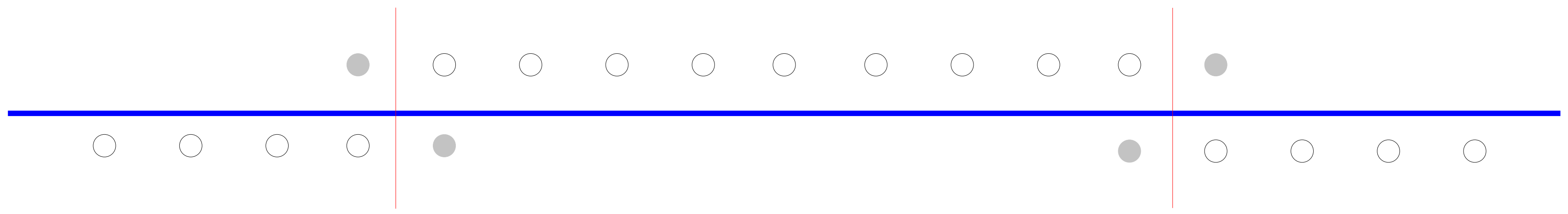
```
t = dt;
while (t<T){
  t += dt;
  for (i=1; i<=M; i++)
    up[i] = 2*u[i]-um[i]+((dt*dt)/(dx*dx))*(u[i-1]-2*u[i]+u[i+1]);
  up[0] = value_of_left_BC(t);   // enforcing left BC
  up[M+1] = value_of_rigt_BC(t); // enforcing right BC

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

Solving 1D wave equation (6)

Parallelization starts with dividing the work

- The global domain is decomposed into P segments (subdomains)
  - actually, the M interiors points are divided



Solving 1D wave equation (7)

Parallel implementation using MPI

- Each subdomain has M/P interior points, plus two “ghost points”
  - if there is a neighbor domain over the boundary, the value of the ghost point is to be provided
  - if there is no neighbor domain over the boundary, the ghost point is actually a physical boundary point
- First, `up_local[i]` is computed on each interior point `i=1,2,...,M_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[M_local]` to the right neighbor
  - receive `up_local[M_local+1]` from the right neighbor

Solving 1D wave equation (8)

Local data structure

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

Solving 1D wave equation (9)

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`
- Afterwards, main local computation over indices `i=2,3,...,M-1`
- Finally, finish communication with left neithbor using `MPI_Wait`
- Finally, finish communication with right neithbor using `MPI_Wait`

Solving 2D wave equtation (1)

Mathematical model

$$\frac{\partial^2 u}{\partial t^2} = \frac{1}{2}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

- Spatial domain: unit square $(x, y) \in (0, 1) \times (0, 1)$
- Temporal domain: $0 < t \leq T$
- Boundary conditions: u is known on the entire boundary
- Initial conditions same as for the 1D case

Solving 2D wave equtation (2)

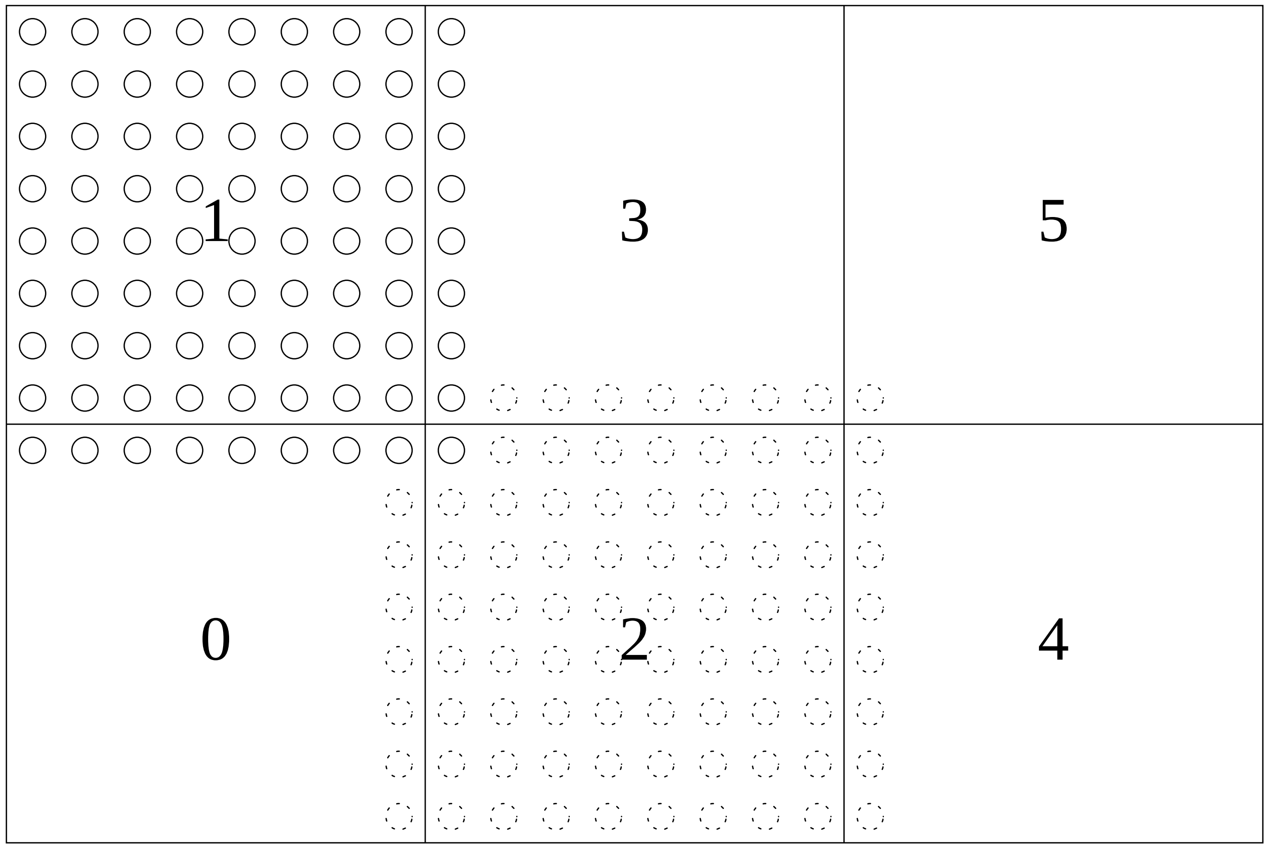
Numerical method

- Uniform 2D mesh $(\Delta x, \Delta y)$
- Time step size Δt
- Central finite differences (same as in 1D)

$$\begin{aligned}
u_{i,j}^{\ell+1} = & \quad 2u_{i,j}^{\ell} - u_{i,j}^{\ell-1} \\
& + \frac{\Delta t^2}{2\Delta x^2}\left(u_{i-1,j}^{\ell} - 2u_{i,j}^{\ell} + u_{i+1,j}^{\ell}\right) \\
& + \frac{\Delta t^2}{2\Delta y^2}\left(u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}\right) \\
& \quad i = 1, 2, \ldots, M, \; j = 1, 2, \ldots, N
\end{aligned}$$

Solving 2D wave equtation (3)

Domain decomposition



Solving 2D wave equtation (4)

Parallelization

- Each subdomain is responsible for a rectangular region of the $M \times N$ interior points
- One layer of ghost points is needed in the local data structure
- Serial local computation + exchange of values for the ghost points