Parallel Algorithms & Implementations: Data-Parallelism, Asynchronous Communication and Master/Worker Paradigm

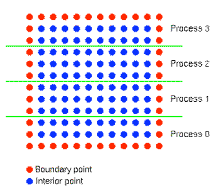
FDI 2007 Track Q

Day 2 – Morning Session

Example: Jacobi Iteration

For all $1 \le i,j \le n$, do until converged …

$u_{ij}^{(new)} \leftarrow 0.25 * (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1})$



(1D Decomp)

Jacobi: 1D Decomposition

- Assign responsibility for *n/p* rows of the grid to each process.
- Each process holds copies (“ghost points”) of one row of old data from each neighboring process.
- Potential for deadlock?
  - Yes, if order of send’s and recv’s is wrong
  - Maybe, with periodic boundary conditions and insufficient buffering, i.e., if recv has to be posted before send returns.

Jacobi: 1D Decomposition

- There is a potential for serialized communication under 2nd scenario above, with Dirichlet boundary conditions:
  - When passing data north, only process 0 can finish send immediately, then process 1 can go, then process 2, etc.
- MPI_Sendrecv function exists to handle this “exchange of data” dance without all the potential buffering problems.

Jacobi: 1D vs. 2D Decomposition

- 2D decomposition: each process holds $n/\sqrt{p} \times n/\sqrt{p}$ subgrid.
- Per-process memory requirements:
  - 1D case: each holds an $n \times n/p$ subgrid
  - 2D case: each holds an $n/\sqrt{p} \times n/\sqrt{p}$ subgrid.
  - If n^2/p is a constant, then in the 1D case the number of rows per process shrinks as n and p grow.

Jacobi: 1D vs. 2D Decomposition

- The ratio of computation to communication is key to scalable performance.
- 1D decomposition:

$$\frac{\text{Computation}}{\text{Communication}} = \frac{n^2/p}{n} = \frac{1}{\sqrt{p}} * \frac{n}{\sqrt{p}}$$

- 2D decomposition:

$$\frac{\text{Computation}}{\text{Communication}} = \frac{n^2/p}{n/\sqrt{p}} = \frac{n}{\sqrt{p}}$$

MPI Non-Blocking Message Passing

- MPI_Isend initiates send, returning immediately with a request handle.
- MPI_Irecv posts a receive and returns immediately with a request handle.
- MPI_Wait blocks until a given message passing event, specified by handle, is complete.
- MPI_Test can be used to check a handle for completion without blocking.

MPI Non-Blocking Send

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	count	number of entries to send (integer)
IN	datatype	datatype of each entry (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	request handle (handle)

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

MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERR
```

MPI Non-Blocking Recv

MPI_IRECV(buf, count, datatype, source, tag, comm, request)

OUT	buf	initial address of receive buffer (choice)
IN	count	max number of entries to receive (integer)
IN	datatype	datatype of each entry (handle)
IN	dest	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	status	request handle (handle)

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

MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERR)
    <type> BUF(*)
    INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERR
```

Function MPI_Wait

MPI_WAIT(request, status)
INOUT request request handle (handle)
OUT status status object (Status)

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

MPI_WAIT(REQUEST, STATUS, IERR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERR
```

Jacobi with Asynchronous Communication

- With non-blocking sends/recvs, can avoid any deadlocks or slowdowns due to buffer management.
- With some code modification, can improve performance by overlapping communication and computation:

Old Algorithm:	New Algorithm:
exchange data	initiate exchange
do updates	update strictly interior grid points
	complete exchange
	update boundary points

Master/Worker Paradigm

- A common pattern for non-uniform, heterogenous sets of tasks.
- Get dynamic load balancing for free (at least that's the goal)
- Master is a potential bottleneck.
- See example.