Distributed and Parallel Technology

Communication Libraries II
Communication in MPI

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

[0]No proprietary software has been used in producing these slides
[0]Based on earlier versions by Greg Michaelson and Patrick Maier

Communication is Bad

Transmission takes >0 time:

- Suppose program takes time T_1 on 1 processor.
- Simple parallelisation on p workers only worthwhile if $T_1 > S_p + T_1/p + R_p$
  - S_p = time to send input to p processors
  - T_1/p = (unachievable) min time to run program on p processors
  - R_p = time to receive results from p processors

Synchronisation disrupts parallelism:

- Each message synchronises sender and receiver.
  - Frequently, one (most often receiver) has to wait for the other.

Rules of thumb:

1. Transmit as little data as possible.
2. Send as few messages as possible.
   - In general, prefer few large messages to many small ones. (There are exceptions to this corollary.)

Recap: Sequential Matrix Multiplication

```
int ** matrixProd(int ** M1, int ** M2T, int m, int n)
{
  int i, j;
  int ** M3 = allocMatrix(m, m);
  for (i = 0; i < m; i++)     /* iterate over all rows */
    for (j = 0; j < m; j++)   /* iterate over all columns */
      M3[i][j] = dotProd(M1[i], M2T[j], n);
  return M3;
}
```

- input `M1`: $\mathtt{m} \times \mathtt{n}$ matrix
- input `M2T`: transposed $\mathtt{n} \times \mathtt{m}$ matrix
- output: $\mathtt{m} \times \mathtt{m}$ matrix = $\mathtt{M1} * \mathtt{M2T}^T$

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix3.c

Naive Parallel Matrix Multiplication

```
void matrixProdWorker(int m, int n)
{
  MPI_Status status;
  int dp, i;
  int * R = allocVector(n);    /* allocate row vector */
  int * C = allocVector(n);    /* allocate column vector */

  /* iterate m times */
  for (i = 0; i < m; i++) {
    /* receive row and column vectors from master */
    MPI_Recv(R, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(C, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* compute row/column product */
    dp = dotProd(R, C, n);
    /* send product back to master */
    MPI_Send(&dp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
}
```

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix4

Naive Parallel Matrix Multiplication

```
int ** matrixProdMaster(int ** M1, int ** M2T, int m, int n)
{
  int i, j;
  MPI_Status status;
  int ** M3 = allocMatrix(m, m);    /* allocate result matrix */

  /* iterate over columns of result */
  for (j = 0; j < m; j++) {
    /* send row and column vectors to workers */
    for (i = 0; i < m; i++) {
      MPI_Send(M1[i], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
      MPI_Send(M2T[j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
    }
    /* receive result column j */
    for (i = 0; i < m; i++)
      MPI_Recv(&(M3[i][j]), 1, MPI_INT, i+1, 0, MPI_COMM_WORLD, &status);
  }
  return M3;
}
```

- $2m^2$ * SEND(n ints)
  - Row `M1[i]` sent m times to processor `i+1`.
- m^2 * RECV(1 int)

Improved Par Mat Mult — Send Rows Once

```
void matrixProdWorker(int m, int n)
{
  MPI_Status status;
  int dp, i;
  int * R = allocVector(n);
  int * C = allocVector(n);

  /* receive row vector from master, once */
  MPI_Recv(R, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  /* iterate m times */
  for (i = 0; i < m; i++) {
    /* receive column vector from master */
    MPI_Recv(C, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* compute row/column product and send back to master */
    dp = dotProd(R, C, n);
    MPI_Send(&dp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
}
```

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix5.c

Improved Par Mat Mult — Send Rows Once

```
int ** matrixProdMaster(int ** M1, int ** M2T, int m, int n)
{
  int i, j;
  MPI_Status status;
  int ** M3 = allocMatrix(m, m);

  /* send each row of M1, once */
  for (i = 0; i < m; i++)
    MPI_Send(M1[i], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
  /* iterate over columns of result */
  for (j = 0; j < m; j++) {
    /* send column vectors to workers */
    for (i = 0; i < m; i++)
      MPI_Send(M2T[j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
    /* receive result column j */
    for (i = 0; i < m; i++)
      MPI_Recv(&(M3[i][j]), 1, MPI_INT, i+1, 0, MPI_COMM_WORLD, &status);
  }
  return M3;
}
```

- $m + m^2$ * SEND(n ints)   #msgs and data nearly halved!
- m^2 * RECV(1 int)
  - Processor `i+1` sends m messages of size 1 int.

Improved Par Mat Mult — Collect Result Once

```
void matrixProdWorker(int m, int n)
{
  MPI_Status status;
  int i;
  int * R = allocVector(n);
  int * C = allocVector(n);
  int * RR = allocVector(m);    /* allocate result row vector */

  /* receive row vector from master, once */
  MPI_Recv(R, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  /* iterate over entries of result row vector */
  for (i = 0; i < m; i++) {
    /* receive column vector from master */
    MPI_Recv(C, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* compute i-th row/column product */
    RR[i] = dotProd(R, C, n);
  }
  /* send result row vector back to master */
  MPI_Send(RR, m, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix6.c

Improved Par Mat Mult — Collect Result Once

```
int ** matrixProdMaster(int ** M1, int ** M2T, int m, int n)
{
  int i, j;
  MPI_Status status;
  int ** M3 = allocMatrix(m, m);

  /* send each row of M1, once */
  for (i = 0; i < m; i++)
    MPI_Send(M1[i], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
  /* send all columns of M2 to all workers */
  for (j = 0; j < m; j++)
    for (i = 0; i < m; i++)
      MPI_Send(M2T[j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
  /* receive result rows, each row takes 1 message */
  for (i = 0; i < m; i++)
    MPI_Recv(M3[i], m, MPI_INT, i+1, 0, MPI_COMM_WORLD, &status);
  return M3;
}
```

- $m + m^2$ * SEND(n ints)
- m * RECV(m int)

same data but much lower #msgs!

Improved Par Mat Mult — Arbitrary #Processors

```
void matrixProdWorker(int m, int n, int workers, int id)
{
  MPI_Status status;
  int i, j;
  int s = m/workers; if (id <= m%workers) s++; /* #rows to receive */
  int ** R = allocMatrix(s, n);      /* allocate s rows of input */
  int * C = allocVector(n);          /* allocate column vector */
  int ** RR = allocMatrix(s, m);     /* allocate s result rows */
  /* receive s rows from master */
  for (i = 0; i < s; i++)
    MPI_Recv(R[i], n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
  /* iterate over columns of result matrix chunk */
  for (j = 0; j < m; j++) {
    /* receive column j from master */
    MPI_Recv(C, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    /* compute column j of result matrix chunk */
    for (i = 0; i < s; i++)
      RR[i][j] = dotProd(R[i], C, n);
  }
  /* send s rows back to master */
  for (i = 0; i < s; i++)
    MPI_Send(RR[i], m, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix7.c

Improved Par Mat Mult — Arbitrary #Processors

```
/* Precondition: workers >= 1 */
int ** matrixProdMaster(int ** M1, int ** M2T, int m, int n, int workers)
{
  int i, j;
  MPI_Status status;
  int ** M3 = allocMatrix(m, m);
  int s = m/workers;                /* lower bound on #rows per worker */
  int r = m%workers;                /* remaining rows */

  /* First r workers get sent s+1 rows of M1 each */
  for (i = 0; i < r; i++)
    for (j = 0; j < s+1; j++)
      MPI_Send(M1[i * (s+1) + j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
  /* Remaining workers get sent s rows of M1 each */
  for (i = r; i < workers; i++)
    for (j = 0; j < s; j++)
      MPI_Send(M1[r + i * s + j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
  /* All columns of M2 sent to all workers */
  for (j = 0; j < m; j++)
    for (i = 0; i < workers; i++)
      MPI_Send(M2T[j], n, MPI_INT, i+1, 0, MPI_COMM_WORLD);
```

(continues on next slide)


```
  /* First r workers deliver s+1 rows of M3 each */
  for (i = 0; i < r; i++)
    for (j = 0; j < s+1; j++)
      MPI_Recv(M3[i*(s+1) + j],m,MPI_INT,i+1,0,MPI_COMM_WORLD,&status);
  /* Remaining workers deliver s rows of M3 each */
  for (i = r; i < workers; i++)
    for (j = 0; j < s; j++)
      MPI_Recv(M3[r + i*s + j],m,MPI_INT,i+1,0,MPI_COMM_WORLD,&status);

  return M3;
}
```

Communication complexity:

- $(m + workers \cdot m)$ * SEND(n ints)
- m * RECV(m ints)

Collectives

- `MPI_Send` may not be optimal for distributing data to a whole communicator (ie. group of processors).
  - MPI might be able to exploit system architecture optimally if it knew that *n* `MPI_Send` calls were related.
- Main collective communication functions to replace *related calls* to `MPI_Send` and `MPI_Recv`:
  - `MPI_Bcast` broadcasts data to a communicator.
  - `MPI_Scatter` scatters data in chunks across communicator.
  - `MPI_Gather` gathers chunks of data from communicator.
  - `MPI_Reduce` applies an operator across communicator.
  - There are many more ...
- Collectives are elegant constructs but they increase the risk of unintended global synchronisation.

Collectives — Broadcast

```
int MPI_Bcast(
    void * buf,             /* address of 1st data item */
    int count,              /* number of data items */
    MPI_Datatype datatype,  /* type of data items */
    int root,               /* broadcaster */
    MPI_Comm comm)          /* scope of broadcast */
```

- Processor `root` sends `count` items of `datatype`, starting at address `buf`, to all processors of communicator `comm`.
- On completion, every processor finds the data at the address `buf`.
- Global synchronisation: Every processor must wait for `root`.

Collectives — Scatter

```
int MPI_Scatter(
    void * sendbuf,         /* address of 1st item to send */
    int sendcount,          /* #items to send to each proc */
    MPI_Datatype sendtype,/* type of data items to send */
    void * recvbuf,         /* address of 1st item to recv' */
    int recvcount,          /* #items to recv' by each proc */
    MPI_Datatype recvtype,/* type of data items to recv' */
    int root,               /* scatterer */
    MPI_Comm comm)          /* scope of scatter */
```

- Processor `root` splits data items starting at address `sendbuf` into chunks of size `sendcount` and sends them in rank order to all processors in communicator `comm`, including itself.
  - Send buffer must contain $\geq$ *size*(`comm`) $*$ `sendcount` data items.
- Each processor in `comm` receives `recvcount` data items and stores them at address `recvbuf`.
- Usually `sendcount` = `recvcount` and `sendtype` = `recvtype`
  - Not required — can use scatter/gather to coerce types.
- Global synchronisation: Every processor must wait for `root`.

Collectives — Gather

```
int MPI_Gather(
    void * sendbuf,         /* address of 1st item to send */
    int sendcount,          /* #items to send to each proc */
    MPI_Datatype sendtype,/* type of data items to send */
    void * recvbuf,         /* address of 1st item to recv' */
    int recvcount,          /* #items to recv' by each proc */
    MPI_Datatype recvtype,/* type of data items to recv' */
    int root,               /* gatherer */
    MPI_Comm comm)          /* scope of gather */
```

- `MPI_Gather` is the inverse of `MPI_Scatter`
- Each processor in `comm` sends a chunk of `sendcount` data items to `root`.
- `root` receives chunks of `recvcount` data items from each processor in `comm`, including itself, splices them together in rank order and stores them at address `recvbuf`.
  - Receive buffer must have space for $\geq$ *size*(`comm`) $*$ `recvcount` data items.
- Global synchronisation: `root` must wait for every processor.

Collectives — Reduce

```
int MPI_Reduce(
    void * opbuf,           /* addr of 1st item to reduce */
    void * resbuf,          /* addr of 1st reduced item */
    int count,              /* #items to be reduced */
    MPI_Datatype datatype,/* type of items to be reduced */
    MPI_Op op,              /* reduction operator */
    int root,               /* processor collecting result */
    MPI_Comm comm)          /* scope of reduction */
```

- Apply reduction `op` across communicator `comm` to `count` data items stored at address `opbuf`.
  - Reduction operators should be associative-commutative functions
  - Predefined operators: `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`
- On completion, result of reduction is available at address `resbuf` on `root` processor.
- Global synchronisation: `root` must wait for every processor.

Scatter/Reduce Example: Find Max

```
int main(int argc, char ** argv)
{
  ... the usual red tape ...

  /* read #ints from file */
  if (id == 0) {
    fin = fopen(argv[1], "r");
    fscanf(fin, "%d", &n);
  }
  /* broadcast #ints */
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  /* compute chunk size */
  if (n%p) s = n/p + 1; else s = n/p;
  /* Read and pad input on processor 0 */
  if (id == 0) {
    nums = (int *)malloc(p*s * sizeof(int)); /* space for input */
    for (i = 0; i < n; i++)                  /* read ints from file */
      fscanf(fin, "%d", &(nums[i]));
    fclose(fin);
    for (i = n; i < p*s; i++)                /* pad to multiple of p */
      nums[i] = nums[0];
  }
```

(continues on next slide)


```
  /* scatter input on p processors */
  data = (int *)malloc(s * sizeof(int));
  MPI_Scatter(nums, s, MPI_INT, data, s, MPI_INT, 0, MPI_COMM_WORLD);
  /* find local maximum */
  max = findMax(data, s);
  /* reduce local maxima to global maximum */
  MPI_Reduce(&max, &result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
  /* print result on processor 0 */
  if (id == 0) printf("%d\n", result);

  ... the usual red tape at the end ...
}
```

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/max.c

Note: This is just an example for scatter/reduce, not a reasonable parallelisation. In fact, `findMax` is too cheap (linear complexity) to be parallelised in a message passing model — communication always overwhelmes computation.

Some Questions

1. Devise a parallel matrix multiplication algorithm using MPI collective communication.
   - Simple minded solution (with #processors = #rows): http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix6.c
2. Analyse the processing and communication factors in parallelising a binary search algorithm.
3. Discuss the deployment of multiple processors to speed up the sorting of a very large file of numbers using C with MPI.