Overview

- Parallel matrix multiplication
  - Analysis of direct matrix multiply with master and workers
  - Analysis of blocked matrix multiplication with master and workers
  - Analysis of parallel matrix multiply complexity lower bound
  - Full recursive blocked matrix multiply
  - Cannon’s algorithm
  - Systolic array
  - Fox’ algorithm in MPI

- Iterative solver in MPI

- Heat distribution problem in MPI

- Further reading
Parallel Matrix Multiply: Direct Implementation

- Basic algorithm, \( n \times l \) matrix \( A \), \( l \times m \) matrix \( B \), \( n \times m \) matrix \( C \)

\[
C = A \times B \quad c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j}
\]

- Assume square matrices, thus \( n = m = l \)

- \( P = n \times n \) worker processors with \( c_{i,j} \) stored locally on \( p_{i,j} \)

- One master processor sends \( 2n \) elements \( a_{i,k} \) and \( b_{k,j} \) for \( k = 0, \ldots, n-1 \) to each worker \( p_{i,j} \)

- Workers compute and return \( c_{i,j} \) to master processor

- Computation: \( t_{\text{comp}} = 2n \)

- Communication: \( t_{\text{comm}} = n^2(t_{\text{startup}} + 2n \ t_{\text{data}}) + n^2(t_{\text{startup}} + t_{\text{data}}) \)
Parallel Matrix Multiply: Block Matrix Multiplication

- Block matrix multiplication algorithm, with $s \times s$ blocks of size $m \times m$ where $m = n/s$
  
  ```
  \begin{align*}
  &\text{for } p = 0 \text{ to } s-1 \\
  &\quad \text{for } q = 0 \text{ to } s-1 \\
  &\quad \quad C_{p,q} = 0 \\
  &\quad \text{for } r = 0 \text{ to } s-1 \\
  &\quad \quad C_{p,q} = C_{p,q} + A_{p,r} \times B_{r,q} \quad // \text{matrix + and } \times \text{operations}
  \end{align*}
  ```

- $P = s \times s$ worker processors with submatrices $C_{p,q}$ stored locally on $p_{p,q}$

- Master processor sends $2s$ blocks $A_{p,r}$ and $B_{r,q}$ for $r = 0, ..., s-1$ to each worker $p_{p,q}$

- Workers compute inner loop and return $C_{p,q}$ to master processor

- Computation: $t_{comp} = s(2m^3 + m^2) = O(sm^3) = O(nm^2)$

- Communication: $t_{comm} = s^2 \left[ 2(t_{startup} + nm \ t_{data}) + (t_{startup} + m^2 t_{data}) \right]$
Parallel Matrix Multiply: Lower Bound on Complexity

- First assume we have $P = n \times n$ processors.
- Each processor computes $c_{i,j}$ in parallel.
- Assume zero communication overhead, so $a_{i,k}$ and $b_{k,j}$ for $k = 0, ..., n-1$ are directly available to all processors.
- Now add another dimension of $n$ processors ($P = n \times n \times n$) to compute

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

using a parallel tree-reduction in $\log n$ steps.

- Computation: $t_{\text{comp}} = 1 + \log n = O(\log n)$
- Not cost optimal: $O(P \log n) = O(n^3 \log n) \neq O(n^3)$
Parallel Matrix Multiply: Recursive Implementation

- Block matrix multiplication in recursion by decomposing matrix in $2 \times 2$ submatrices and computing the submatrices recursively

```c
Mat matmul(Mat A, Mat B, int s)
{ if (s == 1)
    C = A * B;
else
{ s = s/2;
    P0 = matmul(A\_p,p, B\_p,p, s);
    P1 = matmul(A\_p,q, B\_q,p, s);
    P2 = matmul(A\_p,p, B\_p,q, s);
    P3 = matmul(A\_p,q, B\_q,q, s);
    P4 = matmul(A\_q,p, B\_p,p, s);
    P5 = matmul(A\_q,q, B\_q,p, s);
    P6 = matmul(A\_q,p, B\_p,q, s);
    P7 = matmul(A\_q,q, B\_q,q, s);
    C\_p,p = P0 + P1;
    C\_p,q = P2 + P3;
    C\_q,p = P4 + P5;
    C\_q,q = P6 + P7;
}
return C;
}
```

- Level of parallelism increases with deepening recursion
- Suitable for shared memory systems
- Excessive message passing on distributed memory systems
Parallel Matrix Multiply: Cannon's Algorithm

1. Initially each $p_{i,j}$ has $a_{i,j}$ and $b_{i,j}$.
2. Align elements $a_{i,j}$ and $b_{i,j}$ by reordering them so that $a_{i,j+i}$ and $b_{i+j,j}$ are on $p_{i,j}$.
3. Each $p_{i,j}$ computes
   \[ c_{i,j} = a_{i,j+i} \times b_{i+j,j} \]
   ($a_{i,j+i}$ and $b_{i+j,j}$ are local on $p_{i,j}$).
4. For $k = 1$ to $n-1$ repeat 5-7:
5. Rotate $A$ left by one column.
6. Rotate $B$ up by one row.
7. Each $p_{i,j}$ computes
   \[ c_{i,j} = c_{i,j} + a_{i,j+i+k} \times b_{i+j+k,j} \]
   ($a_{i,j+i+k}$ and $b_{i+j+k,j}$ are local on $p_{i,j}$ after $k$ rotations).

Note: send-recv wrap around the processor grid.

Note: subscripts are modulo $n$. 

send($a$, $p_{i,j-1}$)  
recv($a$, $p_{i,j+1}$)  
send($b$, $p_{i-1,j}$)  
recv($b$, $p_{i+1,j}$)  
c = c + a*b
Parallel Matrix Multiply: Analysis of Cannon’s Algorithm

- Consider block matrix multiplication with Cannon’s algorithm, with \( s \times s \) blocks of size \( m \times m \) where \( m = n/s \)
- Initial alignment requires \( s - 1 \) rotations of \( A \) and \( B \) each moving \( m \times m \) blocks in parallel, and one \( m \times m \) matrix multiply per processor in \( 2m^3 \) time
- Algorithm takes \( s - 1 \) steps
  - Each processor performs a local matrix multiply on its \( m \times m \) block in \( 2m^3 \) time
  - Rotation of \( A \) and \( B \) on \( m \times m \) blocks, where each processor sends and receives two \( m \times m \) blocks (one per row and one per column)
- Computation: \( t_{comp} = 2sm^3 = 2m^2n = O(m^2n) \)
- Communication: \( t_{comm} = 4(s-1)(t_{startup} + m^2t_{data}) \)
Parallel Matrix Multiply: Systolic Array

“Two-dimensional pipeline”

Each processor repeats a “recv-compute-send” stage n times:

recv(a, p_{i,j-1})
recv(b, p_{i-1,j})
c = c + a*b
send(a, p_{i,j+1})
send(b, p_{i+1,j})
Parallel Matrix Multiply: Fox’ Algorithm

- Similar to Cannon’s algorithm
- No initial alignment
- Combines broadcast of $A$ with rotation of $B$

1. For $k=0$ to $n-1$ repeat 2-4:
2. For each row $i$, broadcast element $a_{i,i+k}$ along that row
3. Compute $c_{i,j} = c_{i,j} + a_{i,i+k} \cdot b_{i+k,j}$ ($a_{i,i+k}$ and $b_{i+k,j}$ are local on $p_{i,j}$)
4. Rotate $B$ up by one row

For this algorithm, what is $t_{\text{comp}}$ and $t_{\text{comm}}$?
Parallel Matrix Multiply: Fox’ Blocked Algorithm

- Fox’ block matrix multiply with $s \times s$ blocks of size $m \times m$ where $m = n/s$

1. For $k=0$ to $s-1$ repeat 2-4:
2. For each processor row $p$, broadcast submatrix $A_{p,q+k}$ along processor row $p$
3. Compute
   $C_{p,q} = C_{p,q} + A_{p,q+k} \cdot B_{p+k,q}$
   ($A_{p,q+k}$ & $B_{p+k,q}$ are local on $p_{p,q}$)
4. Rotate $B$ up by one processor row
Parallel Matrix Multiply:
Fox’ Algorithm in MPI

```c
void Fox(GridInfo *grid, Matrix *Apq, Matrix *Bpq, Matrix *Cpq, int M) {
  int k, r;
  int dn = (grid->p + 1) % grid->s; // The “below” and “above” processes
  int up = (grid->p + grid->s - 1) % grid->s;
  MPI_Status stat;
  Matrix Atmp[M*M];

  setzero(C);
  for (k = 0; k < grid->s; k++)
  {
    r = (grid->p + k) % grid->s;
    if (r == grid->q)
    { MPI_Bcast(Apq, M*M, MPI_DOUBLE, r, grid->row);
      matmul(Apq, Bpq, Cpq, M);
    }
    else
    { MPI_Bcast(Atmp, M*M, MPI_DOUBLE, r, grid->row);
      matmul(Atmp, Bpq, Cpq, M);
    }
    MPI_Sendrecv_replace(Bpq, M*M, MPI_DOUBLE, up, 0, dn, 0, grid->col, &stat);
  }
}
```

Row and column communicators
(these are relative to each process)
typedef struct GridInfo
{
    int s;                    // The s×s processor grid
    int p, q;                 // Position (p,q) of the process on the grid
    MPI_Comm row, col;        // Row and column communicators
} GridInfo;

void setup(GridInfo *grid)
{
    MPI_Comm comm;
    int numproc, rank, dim[2], wrap[2], coord[2], freecoord[2];
    MPI_Comm_size(MPI_COMM_WORLD, &numproc);
    grid->s = (int)sqrt(numproc);
    dim[0] = dim[1] = grid->s;
    wrap[0] = wrap[1] = 1;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dim, wrap, 1, &comm);
    MPI_Comm_rank(comm, &rank);
    MPI_Cart_coords(comm, rank, 2, coord);
    grid->p = coord[0];
    grid->q = coord[1];
    ...
}
Parallel Matrix Multiply:
Fox’ Algorithm in MPI (cont’d)

typedef struct GridInfo
{
    int s; /* The s\times s processor grid */
    int p, q; /* Position (p,q) of the process on the grid */
    MPI_Comm row, col; /* Row and column communicators */
} GridInfo;

void setup(GridInfo *grid)
{
    ...
    freecoord[0] = 0;
    freecoord[1] = 1;
    MPI_Cart_sub(comm, freecoord, &grid->row);
}

Setup row communicator
feecoord[0] = 0;
feecoord[1] = 1;
MPI_Cart_sub(comm, freecoord, &grid->row);

Setup column communicator
feecoord[0] = 1;
feecoord[1] = 0;
MPI_Cart_sub(comm, freecoord, &grid->col);
Parallel Iterative Solver: Jacobi Method

\[ x_i^k = \frac{1}{a_{i,i}} \left[ b_i - \sum_{j \neq i} a_{i,j} x_j^{k-1} \right] \]

for \( i = 0; i < m; i++ \)
\[ x_p[i] = b_p[i]; \]
do
\{ \n  allGather x_p[0...m-1] into xold[0...n-1];
  for \( i = 0; i < m; i++ \)
  \{  
    x_p[i] = b_p[i];
    for \( j = 0; j < n; j++ \)
    \{  
      if \( j != p*m+i \)
        x_p[i] = x_p[i] - A_p[i,j]*xold[j];
    \}
  \}
} while (...);

1. Distribute \( n \times n \) matrix \( A \) by rows and vector \( b \) in blocks of size \( m = n/P \) over \( P \) processors into local \( A_p \) and \( b_p \)
2. Assign \( x_p = b_p \)
3. Repeat 4-6 until convergence or max iterations reached:
4. Gather \( x_p \) into \( xold \) \{ allGather \}
5. Broadcast \( xold \)
6. Compute new \( x_p \) using \( A_p \), \( b_p \), \( xold \)
Parallel Iterative Solver: Jacobi Method in MPI

```c
void Jacobi(Matrix *Ap, Vector *bp, Vector *xp, int N, int M)
{
    Vector xold[N];
    int i, j, p;

    MPI_Comm_rank(MPI_COMM_WORLD, &p);
    for (i = 0; i < M; i++)
        xp[i] = bp[i];
    do
    {
        MPI_Allgather(xp, M, MPI_DOUBLE, xold, M, MPI_DOUBLE, MPI_COMM_WORLD);
        for (i = 0; i < M; i++)
        {
            xp[i] = bp[i];
            for (j = 0; j < p*M+i; j++)
                xp[i] = xp[i] - Ap[i][j]*xold[j];
            for (j = p*M+i+1; j < N; j++)
                xp[i] = xp[i] - Ap[i][j]*xold[j];
            xp[i] = xp[i]/Ap[i][i];
        }
    } while (...);
}
```

The global row index $I = pm+i$
Heat Distribution Problem: Parallel Jacobi Iteration

1. Distribute $n \times n$ matrix $h$ block-wise by rows into local $h_p$
2. Extend local $h_p$ with additional top and bottom rows to form “halos” (ghost cells), each block has size $m \times n$, where $m = (n-2)/P+2$
3. Repeat 4-5 until convergence:
4. Exchange rows with neighbor processors to update halo rows
5. Compute $h_{new_p}$
6. Assign $h_{new_p}$ to $h_p$
Heat Distribution Problem in MPI (version 1)

```c
void HDStep(Matrix *hp, int N, int M)
{
    int i, j;
    Matrix hnew[M*N];
    MPI_Status s;
    if (p % 2 == 0 && p < P-1)
        MPI_Sendrecv(hp[M-2], N, MPI_FLOAT, p+1, 0,
                      hp[M-1], N, MPI_FLOAT, p+1, 1, MPI_COMM_WORLD, &s);
    else
        MPI_Sendrecv(hp[1], N, MPI_FLOAT, p-1, 1,
                      hp[0], N, MPI_FLOAT, p-1, 0, MPI_COMM_WORLD, &s);
    if (p % 2 == 1 && p < P-1)
        MPI_Sendrecv(hp[M-2], N, MPI_FLOAT, p+1, 2,
                      hp[M-1], N, MPI_FLOAT, p+1, 3, MPI_COMM_WORLD, &s);
    else if (p > 0)
        MPI_Sendrecv(hp[1], N, MPI_FLOAT, p-1, 3,
                      hp[0], N, MPI_FLOAT, p-1, 2, MPI_COMM_WORLD, &s);
    for (i = 1; i < M-1; i++)
        for (j = 1; j < N-1; j++)
            hnew[i][j] = 0.25*(hp[i-1][j]+hp[i+1][j]+hp[i][j-1]+hp[i][j+1]);
    for (i = 1; i < M-1; i++)
        for (j = 1; j < N-1; j++)
            hp[i][j] = hnew[i][j];
}
```

“even” processor exchanges bottom rows with “odd” processor top rows

“odd” processor exchanges bottom rows with “even” processor top rows
Heat Distribution Problem in MPI (version 2)

```c
void HDStep(Matrix *hp, int N, int M)
{
    int i, j;
    Matrix hnew[M*N];
    MPI_Status s;
    if (p > 0 && p < P-1)
    {
        MPI_Sendrecv(hp[M-2], N, MPI_FLOAT, p+1, 0,
                      hp[0],   N, MPI_FLOAT, p-1, 0, MPI_COMM_WORLD, &s);
        MPI_Sendrecv(hp[1],   N, MPI_FLOAT, p-1, 1,
                      hp[M-1], N, MPI_FLOAT, p+1, 1, MPI_COMM_WORLD, &s);
    }
    else if (p == 0)
    {
        MPI_Sendrecv(hp[M-2], N, MPI_FLOAT, p+1, 0,
                      hp[M-1], N, MPI_FLOAT, p+1, 1, MPI_COMM_WORLD, &s);
        for (i = 1; i < M-1; i++)
            for (j = 1; j < N-1; j++)
                hnew[i][j] = 0.25*(hp[i-1][j]+hp[i+1][j]+hp[i][j-1]+hp[i][j+1]);
    }
    else
    {
        MPI_Sendrecv(hp[1],   N, MPI_FLOAT, p-1, 1,
                      hp[0],   N, MPI_FLOAT, p-1, 0, MPI_COMM_WORLD, &s);
        for (i = 1; i < M-1; i++)
            for (j = 1; j < N-1; j++)
                hp[i][j] = hnew[i][j];
    }
}
```
void HDStep(Matrix *hp, int N, int M, int step)
{
    int i, j;
    int dntag = 2*step; \textit{Tag for “down” sends}
    int uptag = 2*step + 1; \textit{Tag for “up” sends}
    Matrix hnew[M*N];
    MPI_Request sndreq[2], rcvreq[2];
    MPI_Status stat[2];

    if (p < P-1)
    {
        MPI_Isend(hp[M-2], N, MPI_FLOAT, p+1, dntag, MPI_COMM_WORLD, &sndreq[0]);
        MPI_Irecv(hp[M-1], N, MPI_FLOAT, p+1, uptag, MPI_COMM_WORLD, &rcvreq[0]);
    }
    if (p > 0)
    {
        MPI_Isend(hp[1],   N, MPI_FLOAT, p-1, uptag, MPI_COMM_WORLD, &sndreq[1]);
        MPI_Irecv(hp[0],   N, MPI_FLOAT, p-1, dntag, MPI_COMM_WORLD, &rcvreq[1]);
    }

    for (i = 2; i < M-2; i++)
        for (j = 1; j < N-1; j++)
            hnew[i][j] = 0.25*(hp[i-1][j]+hp[i+1][j]+hp[i][j-1]+hp[i][j+1]);
...
Heat Distribution Problem in MPI (version 3, cont’d)

void HDStep(Matrix *hp, int N, int M, int step)
{
    ...
    if (p == 0)
        MPI_Wait(&rcvreq[0], stat);
    else if (p == P-1)
        MPI_Wait(&rcvreq[1], stat);
    else
        MPI_Waitall(2, rcvreq, stat);
    for (j = 1; j < N-1; j++)
    {
        hnew[1][j] = 0.25*(hp[0][j] + hp[2][j] + hp[1][j-1] + hp[1][j+1]);
    }
    if (p == 0)
        MPI_Wait(&sndreq[0], stat);
    else if (p == P-1)
        MPI_Wait(&sndreq[1], stat);
    else
        MPI_Waitall(2, sndreq, stat);
    for (i = 1; i < M-1; i++)
        for (j = 1; j < N-1; j++)
            hp[i][j] = hnew[i][j];
}
Further Reading

- [PP2] pages 340-365