COSC 4397
Parallel Computation

Solving the Laplace Equation with MPI

Edgar Gabriel
Spring 2010

Numerical differentiation - forward difference formula

- From the definition of derivatives
  \[ f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \]
  one can derive an approximation for the 1st derivative
  \[ f'(x) \approx \frac{f(x + h) - f(x)}{h} \]

- The same formula can be obtained from the Taylor series, e.g.
  \[ f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\xi) \]
  \[ f'(x) = \frac{f(x + h) - hf(x)}{h} - \frac{h}{2} f''(\xi) \]
Central Difference Formula

- A better formula is derived if looking at the following two terms
  \[ f(x + h) = f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{3!} f'''(x) + \frac{h^4}{4!} f^{(4)}(\xi_1) \]
  \[ f(x - h) = f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{3!} f'''(x) - \frac{h^4}{4!} f^{(4)}(\xi_2) \]

- Subtracting equation (5:2) from (5:1) leads to
  \[ f'(x) = \frac{1}{2h} [f(x + h) - f(x - h) - \frac{h^2}{12} [...]] \]
  - \( h \) is quadratic in the error term

Central Difference Formula for 2nd Derivatives

- Extend (5:1) and (5:2) by an additional term
  \[ f(x + h) = f(x) + hf''(x) + \frac{h^2}{2} f'''(x) + \frac{h^3}{3!} f^{(4)}(x) + \frac{h^4}{4!} f^{(5)}(\xi_1) \]
  \[ f(x - h) = f(x) - hf''(x) + \frac{h^2}{2} f'''(x) - \frac{h^3}{3!} f^{(4)}(x) + \frac{h^4}{4!} f^{(5)}(\xi_2) \]

- Adding both equations leads to
  \[ f''(x) = \frac{1}{h^2} [f(x + h) - 2f(x) + f(x - h)] - \frac{h^4}{12} [...]] \]
Numerical differentiation - summary

- Forward difference formula:
  \[ f'(x) = \frac{f(x + h) - f(x)}{h} \]

- Central difference formula for the 1st derivative:
  \[ f'(x) = \frac{1}{2h} [f(x + h) - f(x - h)] \]

- Central difference formula for the 2nd derivative:
  \[ f''(x) = \frac{1}{h^2} [f(x + h) - 2f(x) + f(x - h)] \]

Differential equations - terminology

- Differential equations: equations containing the derivative of a function as a variable
  - An ordinary differential equation (ODE) only contains functions of one independent variable
  - A partial differential equation (PDE) contains functions of multiple independent variables and their partial derivatives
- The order of a differential equation is that of the highest derivative that it contains
- The goal is to find a function \( y(t) \) whose derivatives fulfill the given differential equations, e.g.
  \[ y^{(n)}(t) = f(t, y, y', y'', ..., y^{(n-1)}) \]
Finite Differences Approach for Solving Differential Equations

- Idea: replace the derivatives in the DE by an according approximation formula
  - Typically central differences
    \[ y'(t) = \frac{1}{2h} [y(t + h) - y(t - h)] \]
    \[ y''(t) = \frac{1}{h^2} [y(t + h) - 2y(t) + y(t - h)] \]
- Example: Boundary value problem of an ordinary differential equation
  \[ \frac{d^2 y}{dx^2} = f(x, y, \frac{dy}{dx}) \quad a \leq x \leq b \]
  \[ y(a) = \alpha \quad y(b) = \beta \]

Finite Differences Approach (II)

- For simplicity, let's assume the points are equally spaced
  \[ x_i = a + ih \quad 0 \leq i \leq n + 1 \quad h = \frac{(b - a)}{n + 1} \]
- A two point boundary value problem becomes then
  \[ \begin{cases} 
    y_0 = \alpha \\
    \frac{1}{h^2} (y_{i+1} - 2y_i + y_{i-1}) = f(x, y, \frac{1}{2h} (y_{i+1} - y_{i-1})) \\
    y_{n+1} = \beta
  \end{cases} \quad (x:1) \]
- Equation (x:1) leads to a system of equations
- Solving the system of linear equations gives the solution of the ODE at the distinct points \( x_0, x_1, \ldots, x_n, x_{n+1} \)
Example (I)

- Solve the following two point boundary value problem using the finite difference method with $h=0.2$

\[
\begin{align*}
\frac{d^2 y}{dx^2} + 2 \frac{dy}{dx} + 10x &= 0 \quad 0 \leq x \leq 1 \\
y(0) &= 1 \\
y(1) &= 2
\end{align*}
\]

- Since $h=0.2$, the mesh points are

\[
x_0 = 0, x_1 = 0.2, x_2 = 0.4, x_3 = 0.6, x_4 = 0.8, x_5 = 1.0
\]

- Thus, $y_0 = y(x_0) = 1, y_5 = y(x_5) = 2, y_1 - y_4$ are unknown

Example (II)

- Discrete version of the ODE using central differences:

\[
\begin{align*}
\frac{1}{h^2} (y_{i+1} - 2y_i + y_{i-1}) + 2 \frac{1}{2h} (y_{i+1} - y_{i-1}) + 10x_i &= 0 \\
\frac{1}{0.2^2} (y_{i+1} - 2y_i + y_{i-1}) + 2 \frac{1}{0.4} (y_{i+1} - y_{i-1}) + 10x_i &= 0 \\
25(y_{i+1} - 2y_i + y_{i-1}) + 5(y_{i+1} - y_{i-1}) + 10x_i &= 0 \\
20y_{i+1} - 50y_i + 30y_{i+1} &= -10x_i
\end{align*}
\]
Example (III)

\[
i = 1: \quad 20y_0 - 50y_1 + 30y_2 = -10x_1 \quad \rightarrow \quad 20 - 50y_1 + 30y_2 = -10 \cdot 0.2
\]

\[
i = 1: \quad -50y_1 + 30y_2 = -22
\]

\[
i = 2: \quad 20y_1 - 50y_2 + 30y_3 = -4
\]

\[
i = 3: \quad 20y_2 - 50y_3 + 30y_4 = -6
\]

\[
i = 4: \quad 20y_3 - 50y_4 = -68
\]

\[
\begin{bmatrix}
-50 & 30 \\
20 & -50 & 30 \\
20 & -50 & 30 \\
20 & -50 &
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 
\end{bmatrix}
= 
\begin{bmatrix}
-22 \\
-4 \\
-6 \\
-68 
\end{bmatrix}
\]

or

\[
A \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = b
\]

Solving \(Ay=b\) using Bi-CGSTAB

Given \(A, b\) and an initial guess \(y_0\)

\[
\begin{align*}
\rho_0 &= \alpha = \omega_0 = 1 \\
v_0 &= p_0 = 0 \\
\text{for } i = 1, 2, \ldots,
\end{align*}
\]

\[
\begin{align*}
\rho_i &= r_i^T r_{i-1} \\
\beta &= \frac{\rho_i}{\rho_{i-1}} \\
p_i &= r_{i-1} + \beta(p_{i-1} - \omega_{i-1}v_{i-1}) \\
v_i &= Ap_i \\
\alpha &= \frac{r_i^T v_i}{r_0^T v_0} \\
s &= r_{i-1} - \alpha v_i \\
t &= As \\
\omega_i &= \frac{t^T t}{t^T s} \\
y_i &= y_{i-1} + \alpha p_i + \omega_i s \\
r_i &= s - \alpha t
\end{align*}
\]

Matrix-vector multiplication
Scalar product in parallel

- Scalar product:
  \[ s = \sum_{i=0}^{N-1} a[i] * b[i] \]

- Parallel algorithm
  \[ s = \sum_{i=0}^{N/2-1} (a[i] * b[i]) + \sum_{i=N/2}^{N-1} (a[i] * b[i]) \]
  \[ = \sum_{i=0}^{N/2-1} (a_{local}[i] * b_{local}[i]) + \sum_{i=N/2}^{N-1} (a_{local}[i] * b_{local}[i]) \]
  - requires communication between the processes

Matrix-vector product in parallel

- Process 0
  \[ \begin{bmatrix} -50 & 30 \\ 20 & -50 & 30 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} rhs_1 \\ rhs_2 \end{bmatrix} \]

- Process 1
  \[ \begin{bmatrix} 20 & -50 & 30 \\ 20 & -50 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} rhs_3 \\ rhs_4 \end{bmatrix} \]

\[ -50x_1 + 30x_2 = rhs_1 \]
\[ 20x_1 - 50x_2 + 30x_3 = rhs_2 \]
\[ 50x_2 + 30x_3 + 50x_4 = rhs_3 \]
\[ 20x_1 - 50x_4 = rhs_4 \]

- Process 0 needs \( x_3 \)
- Process 1 needs \( x_2 \)
Matrix vector product in parallel (II)

- Introduction of ghost cells
  - Process zero
  - Process one

- Looking at the source code, e.g ...
  \[
  p_i = r_{i-1} + \beta (p_{i-1} - \omega_{i-1} v_{i-1}) \\
  v_i = Ap_i \\
  \]

- ...since the vector used in the matrix vector multiplication changes every iteration, you always have to update the ghost cells before doing the calculation

Matrix vector product in parallel (III)

- ... so the parallel algorithm for the same area is:
  \[
  p_i = r_{i-1} + \beta (p_{i-1} - \omega_{i-1} v_{i-1}) \\
  \text{Update the ghost-cells of } p, \text{ e.g.} \\
  - \text{Process 0 sends } p(2) \text{ to Process 1} \\
  - \text{Process 1 sends } p(3) \text{ to Process 0} \\
  v_i = Ap_i \\
  \]

- ...
2D Example - Laplace equation (I)

- 2-D Laplace equation
  \[
  \frac{\partial^2}{\partial x^2} u(x, y) + \frac{\partial^2}{\partial y^2} u(x, y) = 0
  \]

- Central discretization leads to
  \[
  \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} = 0
  \]

2-D Example: Laplace equation (II)

- Parallel domain decomposition
  - Data exchange at process boundaries required
- Halo cells / Ghost cells
  - Copy of the last row/column of data from the neighbor process
Example - 2-D Laplace equation (IV)

- Process mapping and determining neighbor processes
  - \( n_p_x \): no of procs in x direction
  - \( n_p_y \): no of procs in y direction
  - \( n_{\text{left}} = \text{rank} - 1 \)
  - \( n_{\text{right}} = \text{rank} + 1 \)
  - \( n_{\text{up}} = \text{rank} + n_p_x \)
  - \( n_{\text{down}} = \text{rank} - n_p_x \)

- At boundaries: set the rank of the according neighbor to \texttt{MPI_PROC_NULL}
  - a message sent to \texttt{MPI_PROC_NULL} will be ignored by the MPI library
- Easier: use cartesian topology functions

\[
\begin{array}{cccccc}
  & 8 & 9 & 10 & 11 \\
0,2 & 0 & 1,2 & 2,2 & 3,2 \\
0,1 & 1,1 & 2,1 & 3,1 \\
0,0 & 1,0 & 2,0 & 3,0 \\
\end{array}
\]

Laplace equation - communication in y-direction

- \( u(i,j) \) is stored in a matrix
  - \( n_{\text{local}} \): no of local points in x direction
  - \( n_{\text{local}} \): no of local points in y direction

- Dimension of \( u \) on an inner process (= not being at a boundary):
  \[ u(n_{\text{local}} + 2, n_{\text{local}} + 2) \]

- with
  \[ u(1:n_{\text{local}}, 1:n_{\text{local}}) \]

containing the local data
Laplace equation - communication in y-direction

```c
MPI_Request req[8];

MPI_Irecv(&u[1][nylocal+1], nxlocal, MPI_DOUBLE, nup,
        tag, comm, &req[0]);
MPI_Irecv(&u[1][0], nxlocal, MPI_DOUBLE, ndown,
        tag, comm, &req[1]);
MPI_Isend(&u[1][nylocal], nxlocal, MPI_DOUBLE, nup,
        tag, comm, &req[2]);
MPI_Isend(&u[1][1], nxlocal, MPI_DOUBLE, nup,
        tag, comm, &req[3]);

// Waitall might be postponed until communication
// in x-direction has also been posted
MPI_Waitall (4, req, MPI_STATUSES_IGNORE);
```

Laplace equation - communication in x-direction

- Problem: the data which we have to send is not contiguous in the memory
- Logical view of the matrix
  ![Image of logical view](image)
- Layout in memory of the same matrix (in C)
  ![Image of memory layout](image)
Laplace equation - communication in x-direction

- How to implement the halo-cell exchange in x-direction?
  - Send/Recv every element in a separate message
    + works
    - very slow
  - derived datatypes
    - copy the data into a separate vector/array and send this array
      + works
    - a more general interface is provided by MPI to pack data into a contiguous buffer before sending

Using derived datatypes

```c
MPI_Datatype coldat;

// Create a derived datatype describing a column
// of your vector
MPI_Type_vector ( nylocal, 1, nxlocal, MPI_DOUBLE,
                 &coldat );
MPI_Type_commit ( &coldat );

// use that datatype for the communication to your left
// and right neighbors
MPI_Irecv ( &(u[0][1]), 1, coldat, nleft, tag,
            comm, &req[4] );
MPI_Irecv ( &(u[nylocal+1][1], coldat, nright, tag,
            comm, &req[5] );
MPI_Isend ( &(u[1][1]), 1, coldat, nleft, tag,
            comm, &req[6] );
MPI_Isend ( &(u[nylocal][1]), 1, coldat, nright, tag,
            comm, &req[7] );
...```

"// use that datatype for the communication to your left
// and right neighbors
MPI_Irecv ( &(u[0][1]), 1, coldat, nleft, tag,
            comm, &req[4] );
MPI_Irecv ( &(u[nylocal+1][1], coldat, nright, tag,
            comm, &req[5] );
MPI_Isend ( &(u[1][1]), 1, coldat, nleft, tag,
            comm, &req[6] );
MPI_Isend ( &(u[nylocal][1]), 1, coldat, nright, tag,
            comm, &req[7] );
..."
Packing a message

MPI_Pack (void* inbuf, int incount, MPI_Datatype dat, 
            void *outbuf, int *pos, MPI_Comm comm);

- MPI_Pack copies incount elements of type dat from 
inbuf into the user provided buffer outbuf
  - outbuf has to be large enough to hold the data
  - pos contains the position of the last packed data in 
    outbuf. Has to be initialized to zero before first usage
  - can be called several times to pack independent pieces 
    of data
- Send and receive a message, which has been packed using 
  the MPI datatype MPI_PACKED

Packing a message (II)

outbuf before pack, pos=0

pos

MPI_Pack(inbuf1,1,MPI_INT,outbuf,&pos,comm);

outbuf after 1st pack, pos=6

MPI internat header

pos

MPI_Pack(inbuf2,1,MPI_FLOAT,outbuf,&pos,comm);

outbuf after 1st pack, pos=10

pos
Unpacking a message

MPI_Unpack(void *inbuf, int insize, int* pos, 
void* outbuf, int outcount, 
MPI_Datatype dat, MPI_Comm comm);

- MPI_Unpack copies outcount elements of type dat from 
inbuf into the user provided buffer outbuf 
  - inbuf holds the whole message 
  - pos contains the position of the last unpacked data in 
inbuf. Has to be initialized to zero before first usage 
  - can be called several times to pack independent pieces 
of data

Determining the size of the pack-buffer

MPI_Pack_size(int incount, MPI_Datatype dat, 
MPI_Comm comm, int *size);

- MPI_Pack_size returns the size in bytes of the 
required buffer to pack incount elements of type 
dat using MPI_Pack 
  - size might not be identical to 
    incount *sizeof(original datatype) 
  - several calls to MPI_Pack_size required, if you 
    plan to pack more than one type of dat 
    - sum up the returned sizes 
    - you can use size e.g. to malloc a buffer
```c
double *sbufleft, *sbufright, *rbufleft, *rbufright;
int bufsize, posleft=0, posright=0;

/* determine the required buffer sizes and allocate the buffers */
MPI_Pack_size (nylocal, MPI_DOUBLE, comm, &bufsize);
sbufleft = malloc(bufsize);
sbufright = malloc(bufsize);
rbufleft = malloc(bufsize);
rbufright = malloc(bufsize);

/* Pack the data before sending */
for (i=1; i<nylocal+1; i++) {
    MPI_Pack (u[1][i], 1, MPI_DOUBLE, sbufleft &posright, comm);
    MPI_Pack (u[1][i], 1, MPI_DOUBLE, sbufleft &posleft, comm);
}

/* Execute now the real communication */
MPI_Irecv (rbufleft, bufsize, MPI_PACKED, nleft, tag , comm, &req[0]);
MPI_Irecv (rbufright, bufsize, MPI_PACKED, nright, tag , comm, &req[1]);
MPI_Isend (sbufleft, posleft, MPI_PACKED, nleft, tag , comm, &req[2]);
MPI_Isend (sbufright, posright, MPI_PACKED, nright, tag , comm, &req[3]);
MPI_Waitall (4, req, MPI_STATUSES_IGNORE);

/* Unpack the received data */
posright = posleft = 0;
for (i=1; i<nylocal+1; i++) {
    MPI_Unpack (rbufright, bufsize, &posright,
               u[1][i], 1, MPI_DOUBLE, comm);
    MPI_Unpack (rbufleft, bufsize, &posleft,
               u[0][i], 1, MPI_DOUBLE, comm);
}
```

Laplace equation - communication in x-direction (I)

Laplace equation - communication in x-direction (II)
Derived data types vs. pack/unpack

- Advantages of derived datatypes:
  - avoids temporary buffers
  - code potentially shorter
  - gives the MPI library the possibility to optimize the according operations

- Advantages of pack/unpack
  - might lead to performance advantages if the same packed buffer has to be sent to multiple targets
  - many users find pack/unpack intuitive
    - similar to simply copying the data items into a temporary buffer

MPI Parallel Programming Summary

- Execution of the same executable $np$-times
- processes-specific codes sections can rely on the rank of a process
- individual communication (e.g. MPI_Send/Recv) vs. collective communication (e.g. MPI_Bcast)
- blocking communication (e.g. MPI_Send/Recv) vs. non-blocking communication (e.g. MPI_Isend/Irecv)
- group and communicator management (e.g. MPI_Comm_split/MPI_Comm_create)
- process topologies (e.g. MPI_Cart_create)
MPI Parallel Programming Summary

- Data distribution (e.g. 1-D block column/row wise, 2-D)
- Discontiguous data items (derived data types, pack/unpack)