Lecture 11: Parallel finite differences

Overview

- 1D heat equation $u_t = \kappa u_{xx} + f(x,t)$ as example
- Recapitulation of the finite difference method
- Recapitulation of parallelization
- Jacobi method for the steady-state case: $-u_{xx} = g(x)$
- Relevant reading: Chapter 13 in *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**

The heat equation

- 1D Example: temperature history of a thin metal rod $u(x,t)$, for $0 < x < 1$ and $0 < t \leq T$
  - Initial temperature distribution is known: $u(x,0) = I(x)$
  - Temperature at both ends is zero: $u(0,t) = u(1,t) = 0$
  - Heat conduction capability of the metal rod is known
  - Heat source is known
- The 1D partial differential equation:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} + f(x,t)$$

  - $u(x,t)$: the unknown function we want to find
  - κ: known heat conductivity constant
  - $f(x,t)$: known heat source distribution
- More compact notation: $u_t = \kappa u_{xx} + f(x,t)$

The finite difference method

- A uniform spatial mesh: $x_0, x_1, x_2, \ldots, x_n$, where $x_i = i\Delta x$, $\Delta x = \frac{1}{n}$
- Introduction of discrete time levels: $t_\ell = \ell\Delta t$, $\Delta t = \frac{T}{m}$
- Notation: $u_i^\ell = u(x_i, t_\ell)$
- Derivatives are approximated by finite differences
  - $$\frac{\partial u}{\partial t} \approx \frac{u_i^{\ell+1} - u_i^\ell}{\Delta t}$$
  - $$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell}{\Delta x^2}$$

An explicit scheme for 1D heat equation

- Original equation:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} + f(x,t)$$

- Approximation after using finite differences:

$$\frac{u_i^{\ell+1} - u_i^{\ell}}{\Delta t} = \kappa \frac{u_{i-1}^{\ell} - 2u_i^{\ell} + u_{i+1}^{\ell}}{\Delta x^2} + f(x_i, t_\ell)$$

- An explicit numerical scheme for computing $u^{\ell+1}$ based on u^{ℓ}:

$$u_i^{\ell+1} = u_i^{\ell} + \kappa \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^{\ell} - 2u_i^{\ell} + u_{i+1}^{\ell}\right) + \Delta t f(x_i, t_\ell)$$

  for all inner points $i = 1, 2, \ldots, n-1$

  - $u_0^{\ell+1} = u_n^{\ell+1} = 0$ due to the boundary condition

Serial implementation

- Two data arrays: u refers to $u^{\ell+1}$, u_prev refers to u^{ℓ}
- Enforce the initial condition:

```
x = dx;
for (i=1; i<n; i++) {
  u_prev[i] = I(x);
  x += dx;
}
```

- The main computation is a time-stepping process:

```
t = 0;
while (t<T) {
  x = dx;
  for (i=1; i<n; i++) {
    u[i] = u_prev[i]
            +kappa*dt/(dx*dx)*(u_prev[i-1]-2*u_prev[i]+u_prev[i+1])
            +dt*f(x,t);
    x += dx;
  }
  u[0] = u[n] = 0.;  /* enforcement of the boundary condition*/
  tmp = u_prev; u_prev = u; u = tmp; /* shuffle of array pointers */
  t += dt;
}
```

Parallelism

- Important observation: $u_i^{\ell+1}$ only depends on u_{i-1}^{ℓ}, u_i^{ℓ} and u_{i+1}^{ℓ}
- So, computations of $u_i^{\ell+1}$ and $u_j^{\ell+1}$ are independent of each other
- Therefore, we can use several processors to divide the work of computing $u^{\ell+1}$ on all the inner points

Work division

- The $n-1$ inner points are divided evenly among P processors:
  - Number of points assigned to processor p ($p = 0, 1, 2, \dots, P-1$):

$$n_p = \begin{cases} \left\lfloor \frac{n-1}{P} \right\rfloor + 1 & \text{if } p < \text{mod}(n-1, P) \\ \left\lfloor \frac{n-1}{P} \right\rfloor & \text{else} \end{cases}$$

  - Maximum difference in the divided work load is 1

Blockwise decomposition

- Each processor is assigned with a contiguous subset of all x_i
  - Start of index i for processor p:

$$i_{\mathrm{start},p} = 1 + p\left\lfloor \frac{n-1}{P} \right\rfloor + \min(p, \mathrm{mod}(n-1, P))$$

  - Processor p is responsible for computing $u_i^{\ell+1}$ from $i = i_{\mathrm{start},p}$ until $i = i_{\mathrm{start},p} + n_p - 1$

Need for communication

- Observation: computing $u^{\ell+1}_{i_{\mathrm{start},p}}$ needs $u^{\ell}_{i_{\mathrm{start},p}-1}$, which belongs to the left neighbor, $u^{\ell}_{i_{\mathrm{start},p}}$ is needed by the left neighbor
- Similarly, computing $u^{\ell+1}$ on the rightmost point needs a value of u^{ℓ} from the right neighbor, u^{ℓ} on the rightmost point is needed by the right neighbor
- Therefore, two one-to-one data exchanges are needed on every processor per time step
  - Exception: processor 0 has no left neighbor
  - Exception: processor $P-1$ has no right neighbor

Use of ghost points

- Minimum data structure needed on each processor: two short arrays `u_local` and `u_prev_local`, both of length n_p
  - however, computing $u^{\ell+1}$ on the leftmost point needs special treatment
  - similarly, computing $u^{\ell+1}$ on the rightmost point needs special treatment
- For convenience, we extend `u_local` and `u_prev_local` with two ghost values
  - That is, `u_local` and `u_prev_local` are allocated of length $n_p + 2$
  - `u_prev_local[0]` is provided by the left neighbor
  - `u_prev_local[n_p+1]` is provided by the right neighbor
  - Computation of `u_local[i]` goes from `i=1` until `i=n_p`

MPI communication calls

- When `u_local[i]` is computed for `i=1` until `i=n_p`, data exchanges are needed before going to the next time level
  - `MPI_Send` is used to pass `u_local[1]` to the left neighbor
  - `MPI_Recv` is used to receive `u_local[0]` from the left neighbor
  - `MPI_Send` is used to pass `u_local[n_p]` to the right neighbor
  - `MPI_Recv` is used to receive `u_local[n_p+1]` from the left neighbor
- The data exchanges also impose an implicit synchronization between the processors, that is, no processor will start on a new time level before both its neighbors have finished the current time level

Danger for deadlock

- If two neighboring processors both start with `MPI_Recv` and conitnue with `MPI_Send`, deadlock will arise because none can return from the `MPI_Recv` call
- Deadlock will probably not be a problem if both processors start with `MPI_Send` and continue with `MPI_Recv`
- The safest approach is to use a “rea-black” coloring scheme:
  - Processors with an odd rank are labeled as “red”
  - Processors with an even rank are labeled as “black”
  - On “red” processor: first `MPI_Send` then `MPI_Recv`
  - On “black” processor: first `MPI_Recv` then `MPI_Send`

MPI_Sendrecv

- The `MPI_Sendrecv` command is designed to handle one-to-one data exchange

```
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 )
```

- `MPI_PROC_NULL` can be used if the receiver or sender process does not exist
  - Note that processor 0 has no left neighbor
  - Note that processor $P-1$ has no right neighbor

Use of non-blocking MPI calls

- Another way of avoiding deadlock is to use non-blocking MPI calls, for example, `MPI_Isend` and `MPI_Irecv`
- However, the programmer typically has to make sure later that the communication tasks acutally complete by `MPI_Wait`
- A more important motivation for using non-blocking MPI calls is to exploit the possibility of communication and computation overlap $\longrightarrow$ hiding communication overhead

Stationary heat conduction

- If the heat equation $u_t = \kappa u_{xx} + f(x,t)$ reaches a steady state, then $u_t = 0$ and $f(x,t) = f(x)$
- The resulting 1D equation becomes

$$-\frac{d^2u}{dx^2} = g(x)$$

where $g(x) = f(x)/\kappa$

The Jacobi method

- Finite difference approximation gives

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{\Delta x^2} = g(x_i)$$

  for $i = 1, 2, \ldots, n-1$
- The Jacobi method is a simple solution strategy
  - A sequence of approximate solutions: u^0, u^1, u^2, $\ldots$
  - u^0 is some initial guess
  - Similar to a pseudo time stepping process
  - One possible stopping criterion is that the difference between $u^{\ell+1}$ and u^ℓ is small enough

The Jacobi method (2)

- To compute $u^{\ell+1}$:

$$u_i^{\ell+1} = \frac{\Delta x^2 g(x_i) + u_{i-1}^{\ell} + u_{i+1}^{\ell}}{2}$$

  for $i = 1, 2, \ldots, n-1$ while $u_0^{\ell+1} = u_n^{\ell+1} = 0$

- To compute the difference between $u^{\ell+1}$ and u^{ℓ}:

$$\sqrt{(u_1^{\ell+1} - u_1^{\ell})^2 + (u_2^{\ell+1} - u_2^{\ell})^2 + \ldots + (u_{n-1}^{\ell+1} - u_{n-1}^{\ell})^2}$$

Observations

- The Jacobi method has the same parallelism as the explicit scheme for the time-dependent heat equation
- Data partitioning is the same as before
- In an MPI implementation, two arrays `u_local` and `u_prev_local` are needed on each processor
- Computing the difference between $u^{\ell+1}$ and u^{ℓ} is also parallelizable
  - First, every processor computes

$$\text{diff}_p = (u_{p,1}^{\ell+1} - u_{p,1}^{\ell})^2 + (u_{p,2}^{\ell+1} - u_{p,2}^{\ell})^2 + \ldots + (u_{p,n_p}^{\ell+1} - u_{p,n_p}^{\ell})^2$$

  - Then all the diff_p values are added up by a reduction operation, using `MPI_Allreduce`
  - The summed value thereafter is applied with a square root operation

Comments

- Parallelization of the Jacobi method requires both one-to-one communication and collective communication
- There are more advanced solution strategies (than Jacobi) for solving the steady-state heat equation
  - Parallelization is not necessarily more difficult
- 2D/3D heat equations (both time-dependent and steady-state) can be handled by the same principles

Exercise

- Make an MPI implementation of the Jacobi method for solving a 2D steady-state heat equation