

HIGH-PERFORMANCE COMPUTING IN APPLIED MATHEMATICS

MTH 410/510

Practical application:

numerical solution to 1D heat equation using MPI

Consider the one-dimensional heat equation

$$\frac{\partial u}{\partial t} = k \frac{\partial^2 u}{\partial x^2} + f(x, t), \quad 0 < x < L, \quad t > 0 \quad (1)$$

with homogeneous boundary conditions

$$u(0, t) = 0 \quad (2)$$

$$u(L, t) = 0 \quad (3)$$

and the initial condition

$$u(0, x) = u_0(x) \quad (4)$$

We may use a forward difference formula for the time derivative

$$\frac{\partial u}{\partial t}(x_0, t_0) = \frac{u(x_0, t_0 + \Delta t) - u(x_0, t_0)}{\Delta t} + O(\Delta t)$$

and a centered difference formula for the second order spatial derivative

$$\frac{\partial^2 u}{\partial x^2}(x_0, t_0) = \frac{u(x_0 + \Delta x, t_0) - 2u(x_0, t_0) + u(x_0 - \Delta x, t_0)}{(\Delta x)^2} + O(\Delta x)^2$$

If we neglect the truncation error ($\max\{O(\Delta t), O(\Delta x)^2\}$) we obtain the discrete formulation of the continuous problem

$$\begin{aligned} \frac{\bar{u}(x_0, t_0 + \Delta t) - \bar{u}(x_0, t_0)}{\Delta t} &= k \frac{\bar{u}(x_0 + \Delta x, t_0) - 2\bar{u}(x_0, t_0) + \bar{u}(x_0 - \Delta x, t_0)}{(\Delta x)^2} \\ &+ f(x_0, t_0) \end{aligned} \quad (5)$$

By solving (5) for $\bar{u}(x_0, t_0 + \Delta t)$ we may advance in time the numerical solution from $t = t_0$ to $t = t_0 + \Delta t$.

For the numerical implementation we proceed as follows:

- Divide the spatial domain $[0, L]$ into n equal intervals of length $\Delta x = L/n$
- Consider a constant discretization time step Δt .
- Denote

$$x_j = j\Delta x, \quad t_m = m\Delta t$$

The exact solution $u(x_j, t_m)$ is numerically approximated as

$$u(x_j, t_m) \approx \bar{u}(x_j, t_m) \stackrel{\text{def}}{=} u_j^{(m)}$$

The numerical solution is obtained from (5) with $x_0 = x_j, t_0 = t_m$

$$\frac{u_j^{(m+1)} - u_j^{(m)}}{\Delta t} = k \frac{u_{j+1}^{(m)} - 2u_j^{(m)} + u_{j-1}^{(m)}}{(\Delta x)^2} + f(x_j, t_m) \quad (6)$$

”Forward march in time” the numerical solution by solving for $u_j^{(m+1)}$ the *partial difference equation* (6)

$$u_j^{(m+1)} = u_j^{(m)} + s \left(u_{j+1}^{(m)} - 2u_j^{(m)} + u_{j-1}^{(m)} \right) + \Delta t f(x_j, t_m) \quad (7)$$

where

$$s = k \frac{\Delta t}{(\Delta x)^2}$$

is a dimensionless parameter.

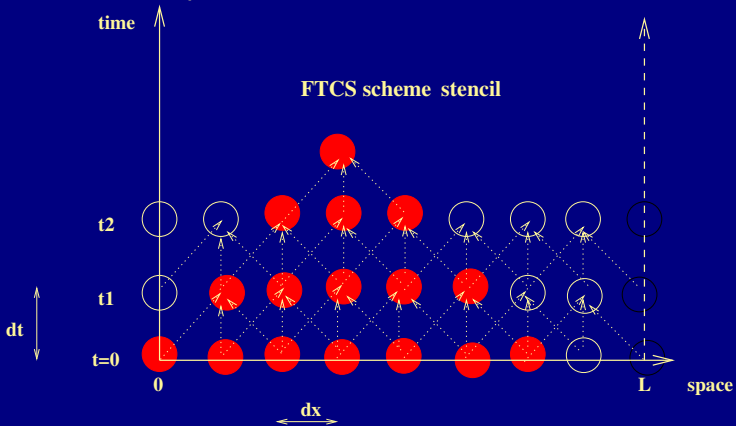
The finite difference scheme (7) is known as the forward time central space (FTCS) discretization to the heat equation.

For the FTCS to be stable we must require

$$s \leq \frac{1}{2} \Rightarrow \Delta t \leq \frac{(\Delta x)^2}{2k}$$

which imposes a restriction on the time step Δt . We say that the FTCS scheme is *conditionally stable*.

FTSC data dependence



Serial FTSC algorithm

State vector $u(0 : n)$

$u(0)$ = left boundary value

$u(n)$ = right boundary value

$t = 0$

$u = \text{initcond}(x, n)$

do while ($t < tend$)

do $i = 1 : n - 1$

$$unext(i) = (1 - 2 * s) * u(i) + s * (u(i - 1) + u(i + 1)) + dt * f(x(i), t)$$

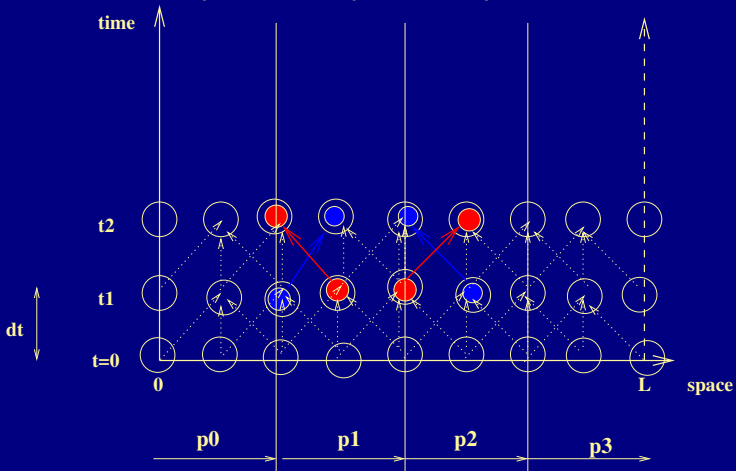
end do

$$u(1 : n - 1) = unext(1 : n - 1)$$

$t = t + dt$

end do

Domain decomposition for parallel implementation



Data distribution

Local spatial domain:

$$loc_n = n/np, \quad loc_x(0 : loc_n + 1)$$

$$loc_x(i) = my_rank * loc_n + i, i = 0 : loc_n + 1$$

Local state vector

$$loc_u(0 : loc_n + 1)$$

Use a ghost point for process $np - 1$

Use $loc_u(0)$ as local left boundary and $loc_u(loc_n + 1)$ as right boundary.

Process i updates $loc_u(1 : loc_n)$

At the end of the time integration process 0 gathers local solutions into the global solution and prints it to a file.

Communication between processes

At each time step

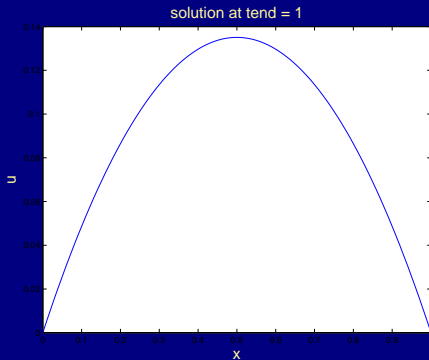
- Process 0
 - 1 send $loc_u(loc_n)$ to process 1
 - 2 receive $loc_u(loc_n + 1)$ from process 1
- Process $i, 1 \leq i \leq np - 2$
 - 1 send $loc_u(1)$ to process $i - 1$
 - 2 send $loc_u(loc_n)$ to process $i + 1$
 - 3 receive $loc_u(0)$ from process $i - 1$
 - 4 receive $loc_u(loc_n + 1)$ from process $i + 1$
- Process $np-1$
 - 1 send $loc_u(1)$ to process $np - 2$
 - 2 receive $loc_u(0)$ from process $np - 1$

Numerical Example

$$k = 1, dx = 0.01, n = 100, s = 0.5, tend = 1$$

$$u_0(x) = x * (1 - x)$$

$$f(x, t) = 2 * \cos(t) - x * (1 - x) * \sin(t)$$



MPI_SENDRECV function

MPI_SEND(buf,count,datatype,dest,tag,comm,ierror)

MPI_RECV(buf,count,datatype,source,tag,comm,status,ierror)

buffer = initial address of the buffer

count = number of entries in buffer

datatype = datatype of entries

A deadlock may occur if send/receive messages are not properly synchronized.

MPI_SENDRECV function

MPI_SEND(buf,count,datatype,dest,tag,comm,ierror)

MPI_RECV(buf,count,datatype,source,tag,comm,status,ierror)

buffer = initial address of the buffer

count = number of entries in buffer

datatype = datatype of entries

A deadlock may occur if send/receive messages are not properly synchronized.

If two processors exchange data a simple way to avoid deadlocks is to use the MPI_SENDRECV function

- performs both a send and a receive
- organizes them such that deadlocks are avoided

MPI_SENDRECV function syntax

C:

```
MPI_Sendrecv(send_buf, send_count, send_type, dest, send_tag,  
recv_buf, recv_count, recv_type, source, recv_tag, comm, status)
```

Fortran:

```
MPI_Sendrecv(send_buf, send_count, send_type, dest, send_tag,  
recv_buf, recv_count, recv_type, source, recv_tag, comm, status,  
ierror)
```

- The parameter list is a concatenation of the parameter lists for MPI_Send and MPI_Recv, *comm* parameter appears only once.
- The *destination* and *source* parameters can be the same
- MPI arranges that no deadlock occurs since it knows that the sends and receives will be paired
- *send_buf* \neq *recv_buf*

Example from 1D heat Replace send/receive pair

```
dest = my_rank + 1
source = my_rank+1
call MPI_SEND(loc_u(loc_n),1,MPI_DOUBLE_PRECISION,dest,1,
& MPI_COMM_WORLD,ierror)
call MPI_RECV(loc_u(loc_n+1),1,MPI_DOUBLE_PRECISION,
& source,1, MPI_COMM_WORLD,status,ierror)
```

by

```
dest = my_rank + 1
source = my_rank+1
call MPI_SENDRECV(loc_u(loc_n),1,MPI_DOUBLE_PRECISION
& dest, 1,loc_u(loc_n+1),1,MPI_DOUBLE_PRECISION,
& source,1,MPI_COMM_WORLD,status,ierror)
```

see [heat1dnew.f](#)

MPI_SENDRECV_REPLACE function

C:

```
MPI_Sendrecv_replace(buf, count, datatype, dest, send_tag,  
source, recv_tag, comm, status)
```

Fortran:

```
MPI_SENDRECV_REPLACE(buf, count, datatype, dest, send_tag,  
source, recv_tag, comm, status, ierror)
```

```
x = my_rank
```

```
y = x
```

```
if (my_rank == 0) then
```

```
    call MPI_SENDRECV_REPLACE(y,1,MPI_REAL,1,0,1,0,  
& MPI_COMM_WORLD,status, ierror)
```

```
elseif (my_rank == 1) then
```

```
    call MPI_SENDRECV_REPLACE(y,1,MPI_REAL,0,0,0,0,  
& MPI_COMM_WORLD,status, ierror)
```

```
end if
```

```
print*,'my rank is ',my_rank,'my x=',x,'my y=',y
```

MPI_SENDRECV_REPLACE function

Example: Consider the following problem:

Processors 0 and 1 hold the vector x , initialized locally.

We wish to send the first half of x from process 0 into the second half of x in process 1 and send the second half of x from process 1 into the first half of x in process 0.

MPI_SENDRECV_REPLACE function

Example: Consider the following problem:

Processors 0 and 1 hold the vector x , initialized locally.

We wish to send the first half of x from process 0 into the second half of x in process 1 and send the second half of x from process 1 into the first half of x in process 0.

Solution using MPI_SENDRECV_REPLACE function (assume $x(1:10)$)

```
if (my_rank == 0) then
  call MPI_SENDRECV_REPLACE(x,5,MPI_REAL,1,0,1,0,
& MPI_COMM_WORLD,status, ierr)
elseif (my_rank == 1) then
  call MPI_SENDRECV_REPLACE(x(6),5,MPI_REAL,0,0,0,0,
& MPI_COMM_WORLD,status, ierr)
```

See [sendreceive.f](#); [sendreceive1.f](#)