MPI Special Topics:
Domain Decomposition
Partial Differential Equations

Martin Siegert, SFU

MPI Special Topics Series

This is about:

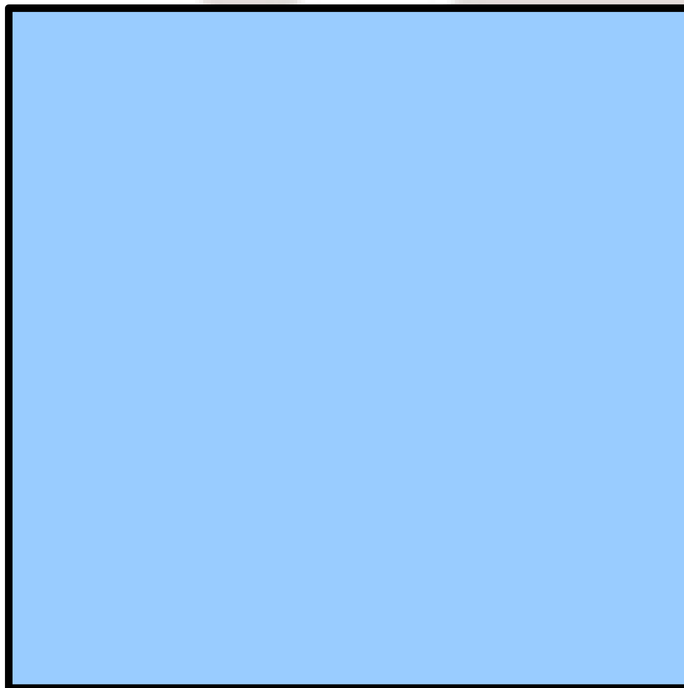
- Learn parallel programming using MPI
- Methods for parallelizing a program
- Use an example to illustrate MPI programming

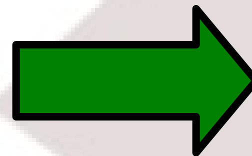
This is not about:

- Advanced Special Topics
- Advanced Parallel Programming Techniques

Domain Decomposition

- Divide problem size into ***np*** pieces
  (***np***: no. of processors)
- assign each piece to one processor





0	1	2
3	4	5
6	7	8

Domain Decomposition

- typical parallelization method for partial differential equations (PDE), molecular dynamics (MD) simulations
- only method for problem sizes that are too big for a single node: memory
- communication required at the boundary of the domains

WestGrid

Domain Decomposition

- several geometries: slab, squares (*2d*), cubes (*3d*) ...

0
1
2
3
4
5

slab

0	1	2
3	4	5
6	7	8

square

- geometries can restrict no. of processors
- appropriate geometry depends on problem, software

Domain Decomposition Partial Differential Eqs.

Example: “Allan-Cahn equation” (Model A)

$$\partial_t \varphi(\vec{r},t) = \nabla^2 \varphi(\vec{r},t) + \varphi(\vec{r},t) - \varphi^3(\vec{r},t)$$

Describes domain growth in magnets, phase separation:

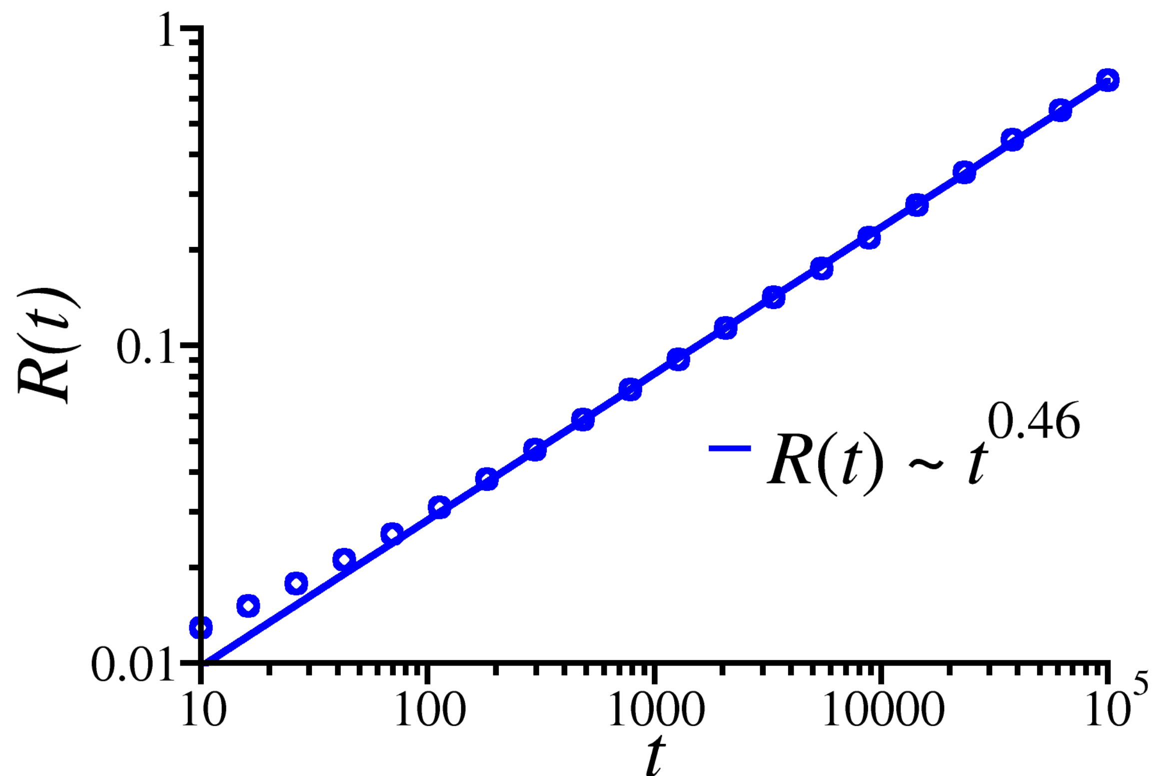
unstable solution: $\varphi = 0$
two stable solutions: $\varphi = 1$ and $\varphi = -1$

Initial configuration $\varphi(\vec{r},t) = 0\,(+\text{noise})$

⇒ +1 and -1 domains grow; domain size $\sim t^{1/2}$

$L \times L = 1920^2$

1¼h walltime on 12 processors on Tantalus



Domain Decomposition Partial Differential Eqs.

$$\partial_t \varphi(\vec{r},t) = \nabla^2 \varphi(\vec{r},t) + \varphi(\vec{r},t) - \varphi^3(\vec{r},t)$$

Discretization:

time stepping: Euler method

$$\partial_t \varphi(\vec{r},t) = \mathrm{rhs}(t) \rightarrow [\varphi(\vec{r},t+\Delta t) - \varphi(\vec{r},t)]/\Delta t = \mathrm{rhs}(t)$$

for serial code Cranck-Nicholson is usually better:

$$[\varphi(\vec{r},t+\Delta t) - \varphi(\vec{r},t)]/\Delta t = [\mathrm{rhs}(t+\Delta t) + \mathrm{rhs}(t)]/2$$

but: implict method requires solving nonlinear equations **globally!**

Domain Decomposition Partial Differential Eqs.

$$\partial_t \varphi(\vec{r}, t) = \nabla^2 \varphi(\vec{r}, t) + \varphi(\vec{r}, t) - \varphi^3(\vec{r}, t)$$

Discretization:

Finite differences for ∇^2 :

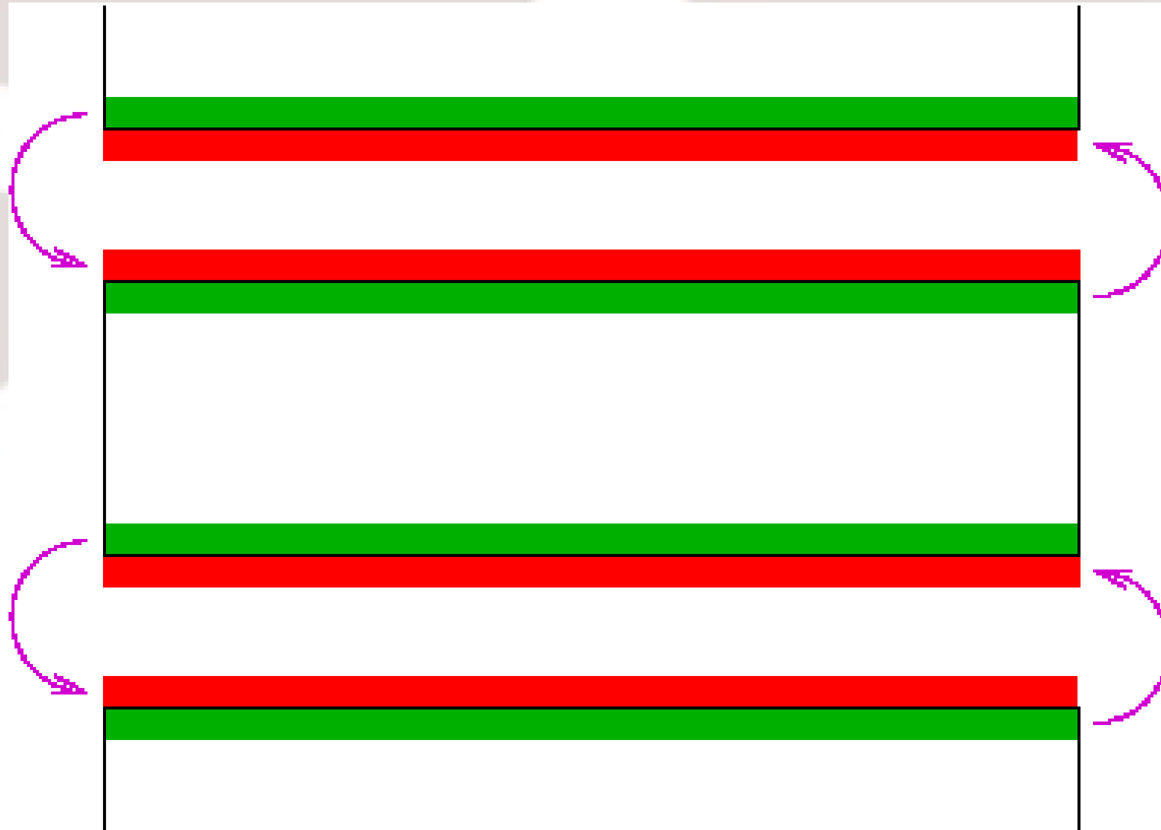
$$\nabla^2 \varphi(\vec{r}, t) \rightarrow$$

$$\frac{1}{\Delta x^2}[\varphi(x+\Delta x, y, t) + \varphi(x-\Delta x, y, t)$$
$$\varphi(x, y+\Delta x, t) + \varphi(x, y-\Delta x, t) - 4\varphi(x, y, t)]$$

Slab geometry $\Rightarrow$

Communication required at domain boundaries!

Domain Decomposition Partial Differential Eqs.



⇒ two send-receive operations in each time step

WestGrid

Parallel Performance

Cost of Communication:

- No. of variables to be transferred in each time step: $N_{trans} = 2 \times L \times np$
  communication time: $t_{com} \sim L$
- Total CPU time: $t_{cpu} \sim L^2 / np$
- Ratio transfer-time/cpu-time:

$$t_{com} / t_{cpu} \sim np / L \overset{L \to \infty}{\to} 0$$

$\Rightarrow$ Domain decomposition becomes more efficient the larger the system size!

Domain Decomposition Setup (F)

```
program allan-cahn
...
   lprocs = l/numprocs
   l = lprocs*numprocs
!  allocate storage
   jmin = myid*lproc
   jmax = (myid + 1)*lproc + 1
   j1 = jmin + 1
   jl = jmax - 1
   allocate(phi(0:l+1,jmin:jmax), stat=istat_alloc)
   if (istat_alloc > 0) then
      call MPI_Abort(MPI_COMM_WORLD,istat_alloc,mpierr)
      stop 'memory allocation failed'
   end if
...
```

red columns

green columns

Domain Decomposition Setup (C)

```
int main(int argc,char *argv[]){
...
   lprocs = l/numprocs;
   l = lprocs*numprocs;
/* allocate storage */
   imin = myid*lproc;
   imax = (myid + 1)*lproc + 1;
   i1 = imin + 1;
   il = imax - 1;
   phi = alloc_darray2d(imin,imax,0,lp1);
   if (phi == NULL) {
      MPI_Abort(MPI_COMM_WORLD, -1);
      exit(-1);
   }
...
```

} red rows

} green rows

Send/Receive Operations MPI_Send

```
MPI_Send(buf,num,<MPI_Type>,dest,tag,
            comm,mpierr)
```

- sends **num** data elements of type **<MPI_Type>** to process **dest**; **tag** identifies message.
- blocking send: will not return until data are copied out of the send buffer.

Send/Receive Operations MPI_Recv

```
MPI_Recv(buf,num,<MPI_Type>,source,
         tag,comm,status,mpierr)
```

- receives **num** data elements of type **<MPI_Type>** from process **source** with message id **tag**.
- blocking receive: will not return until data are stored in the receive buffer.
- **num** must be at least as large as was specified in the matching **MPI_Send**.
- **status** contains additional information

Send/Receive Operations Deadlock

```
sendto = (myid+1)%numprocs;
recvfrom = (myid-1+numprocs)%numprocs;
MPI_Send(sendbuf, n, MPI_DOUBLE, sendto, id,
         MPI_COMM_WORLD);
MPI_Recv(recvbuf, n, MPI_DOUBLE, recvfrom,recvfrom,
         MPI_COMM_WORLD,status);
```

This can hang forever!

Non-Blocking Send/Receive MPI_Isend

```
MPI_Isend(buf,num,<MPI_Type>,dest,tag,
        comm,request,mpierr)
```

- sends **num** data elements of type **<MPI_Type>** to process **dest**; **tag** identifies message.
- posts a non-blocking send: returns immediately, but must not use send buffer until a completion function indicates that the operation is complete.

Non-Blocking Send/Receive MPI_Irecv

```
MPI_Irecv(buf,num,<MPI_Type>,source,
              tag,comm,request,mpierr)
```

- receives **num** data elements of type **<MPI_Type>** from process **source** with message id **tag**.
- posts a non-blocking receive: returns immediately, but must not use receive buffer until a completion function indicates that the operation is complete.

Non-Blocking Send/Receive MPI_Wait, MPI_Waitall

```
MPI_Wait(request,status,mpierr)
```

- blocks until the send or recv operation identified by `request` is complete.

```
MPI_Waitall(count, request_array,
                   status_array,mpierr)
```

- blocks until all `count` operations identified by the requests in `request_array` are complete.

Non-Blocking Send/Receive

```
MPI_Irecv(recvbuf, n, MPI_DOUBLE, recvfrom,
          recvfrom, MPI_COMM_WORLD,req[0]);
MPI_Isend(sendbuf, n, MPI_DOUBLE, sendto,
          id, MPI_COMM_WORLD,req[1]);
MPI_Waitall(2,req,status_array);
```

Is safe!

MPI_Sendrecv

```
MPI_Sendrecv(sendbuf,n_send,<S_Type>,
             dest,sendtag,
             recvbuf,n_recv,<R_Type>,
             source,recvtag,
             comm,status,mpierr)
```

- blocks until both the send and receive operations are complete.

Send/Receive Operations Avoiding Deadlock

```
sendto = (myid+1)%numprocs;
recvfrom = (myid-1+numprocs)%numprocs;
MPI_Sendrecv(sendbuf, n, MPI_DOUBLE, sendto, myid,
             recvbuf, n, MPI_DOUBLE, recvfrom,
             recvfrom, MPI_COMM_WORLD, status);
```

Safe!

MPI standard guarantees that `MPI_Sendrecv` does not deadlock.

MPI Summary

- Blocking Send and Receive:

  `MPI_Send` and `MPI_Recv`

- Non-Blocking Send and Receive

  `MPI_Isend` and `MPI_Irecv`

- Completion Operations

  `MPI_Wait` and `MPI_Waitall`

- SendRecv

  `MPI_Sendrecv`

Questions?

- www.westgrid.ca/downloads/PPT/PDE.pdf
- www.westgrid.ca/downloads/allan-cahn-C.tar.gz
- www.westgrid.ca/downloads/allan-cahn-F.tar.gz
- MPI – The Complete Reference (Vol. 1)