Introduction to Message Passing Interface (MPI)

Bruno C. Mundim

SciNet HPC Consortium

December 8, 2023

Outline

- Scientific MPI Example: 1D Diffusion Equation
- Non-blocking communications

Scientific MPI Example

Scientific MPI Example

Consider a diffusion equation with an explicit **finite-difference**, **time-marching** method.

Imagine the problem is too large to fit in the memory of one node, so we need to do **domain decomposition**, and use **MPI**.

Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

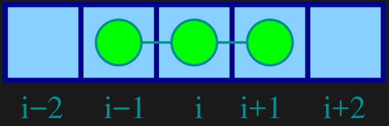
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

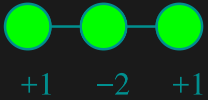
  are usually numerically solved by finite differencing the discretized values.

- Implicitly or explicitly involves interpolating data and taking the derivative of the interpolant.

- Larger ‘stencils’ → More accuracy.

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$





Diffusion equation in higher dimensions

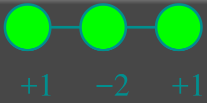
Spatial grid separation: Δx. Time step Δt.

Grid indices: i, j. Time step index: (n)

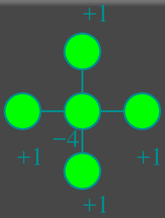
1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



$$\left.\frac{\partial T}{\partial t}\right|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

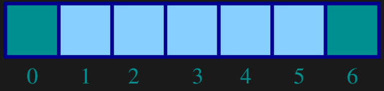
$$\left.\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)\right|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

Stencils and Boundaries

- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution:

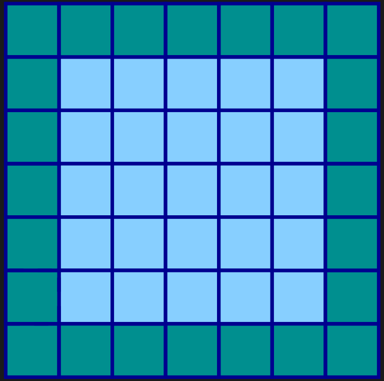
  Guard cells:
  - Pad domain with these guard cells so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



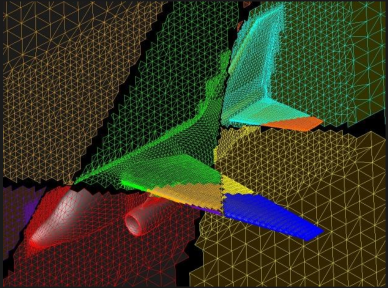
- Number of guard cells $n_g = 1$
- Loop from $i = n_g \dots N - 2n_g$.

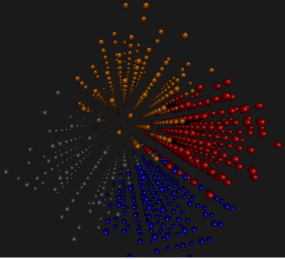
2D

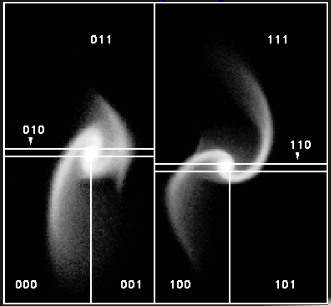


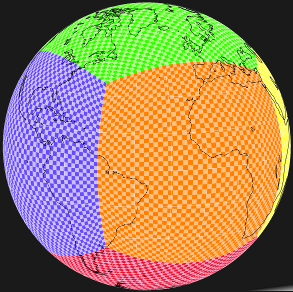
Domain decomposition

- A very common approach to parallelizing on distributed memory computers.
- Subdivide the domain into contiguous subdomains.
- Give each subdomain to a different MPI process.
- No process contains the full data!
- Maintains locality.
- Need mostly local data, ie., only data at the boundary of each subdomain will need to be sent between processes.



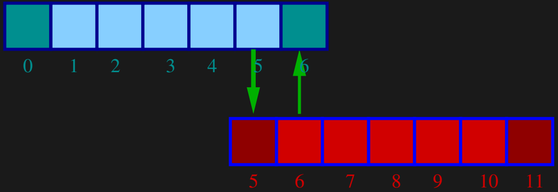






Guard cell exchange

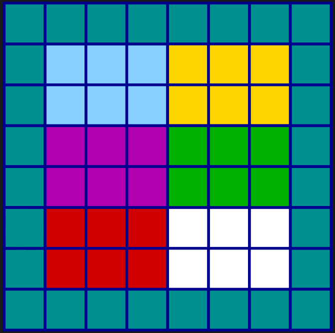
- In the domain decomposition, the stencils will jut out into a neighbouring subdomain.
- Much like the boundary condition.
- One uses guard cells for domain decomposition too.
- If we managed to fill the guard cell with values from neighbouring domains, we can treat each coupled subdomain as an isolated domain with changing boundary conditions.



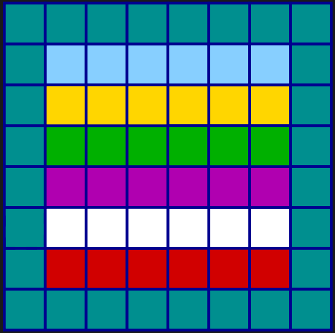
- Could use even/odd trick, or sendrecv.

2D diffusion with MPI

How to divide the work in a 2D grid?



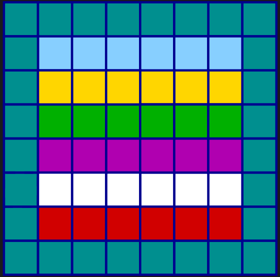
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.



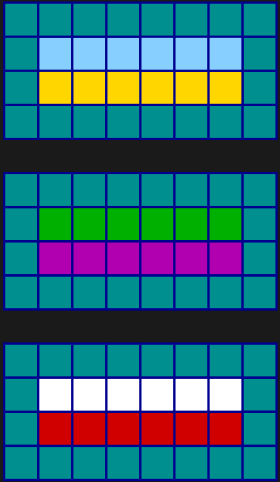
- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

Let’s look at the easiest domain decomposition.

Serial:



Parallel ($P = 3$):



Communication pattern:

- Copy upper stripe to upper neighbour bottom guard cell.
- Copy lower stripe to lower neighbout top guard cell.
- Contiguous cells: can use `count` in `MPI_Sendrecv`.
- Similar to 1d diffusion.

Hands-on: 1D MPI Diffusion

- Serial code:

```
$ cd $SCRATCH/intro-mpi/diffusion
$ # source ../setup
$ make diffusionc # or diffusionf
$ ./diffusionc
```

- `cp diffusion.c diffusionc-mpi.c`
  or
  `cp diffusion.f90 diffusionf-mpi.f90`
- Make an MPI-ed version!
- Build with `make diffusionc-mpi` or `make diffusionf-mpi`.
- Test on 1..8 processors

Plan of Attack

- Switch off graphics (in Makefile, change USEPGPLOT=-DPGPLOT to USEPGPLOT=);
- Add standard MPI calls: init, finalize, comm_size, comm_rank;
- Figure out how many points each process is responsible for (~totpoints/size);
- Figure out neighbors;
- Start at 1, but end at totpoints/size;
- At end of step, exchange guardcells; use sendrecv;
- Get total error.

Non-blocking communications

MPI Non-Blocking Communications

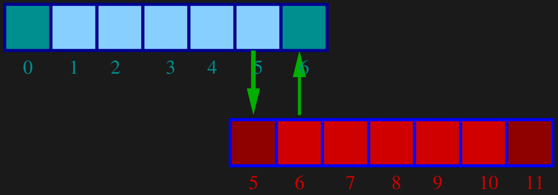
- Mechanism for overlapping/interleaving communications and useful computations
- Avoid deadlocks
- Can avoid system buffering, memory-to-memory copying and improve performance

MPI Non-Blocking Functions: `MPI_Isend`, `MPI_Irecv`

- Returns immediately, posting request to system to initiate communication.
- However, communication is not completed yet.
- Cannot tamper with the memory provided in these calls until the communication is completed.

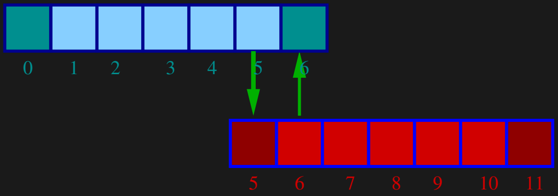
Diffusion: Had to wait for communications to compute

- Could not compute end points without guardcell data
- All work halted while all communications occurred
- Significant parallel overhead.

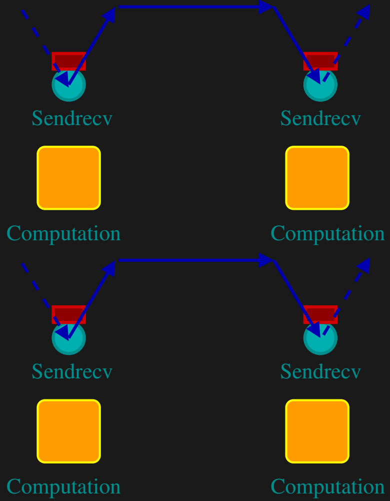


Diffusion: *Had* to wait?

- But inner zones could have been computed just fine.
- Ideally, would do inner zones work while communications is being done; then go back and do end points.



Blocking Communication/Computation Pattern

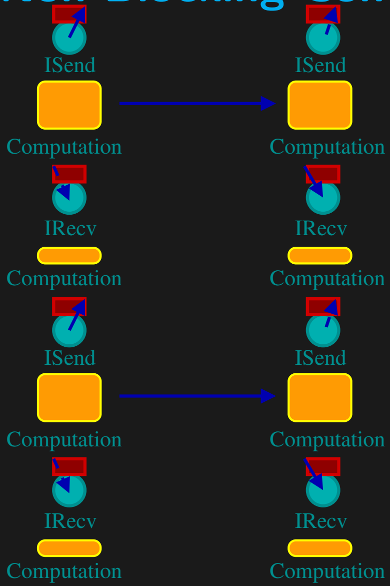


We have the following sequence of communication and computation:

- The code exchanges guard cells using `Sendrecv`
- The code **then** computes the next step.
- The code exchanges guard cells using `Sendrecv` again.
- etc.

We can do better.

Non-Blocking Communication/Computation Pattern



- The code start a send of its guard cells using `ISend`.
- Without waiting for that send's completion, the code computes the next step for the inner cells (while the guard cell message is *in flight*)
- The code then receives the guard cells using `IRecv`.
- Afterwards, it computes the outer cell's new values.
- Repeat.

Nonblocking Sends

- Allows you to get work done while message is *in flight*.
- Must not alter send buffer until send has completed.
- C:

```
MPI_Isend(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
```

- FORTRAN:

```
MPI_ISEND(BUF,INTEGER COUNT,INTEGER DATATYPE,INTEGER DEST,INTEGER TAG, INTEGER COMM, INTEGER REQUEST,
                                                                                    INTEGER ERROR)
```

MPI: Non-Blocking Isend & Irecv

```
err = MPI_Isend(sendptr, count, MPI_TYPE, destination, tag, Communicator, MPI_Request)
err = MPI_Irecv(rcvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Request)
```

- sendptr/rcvptr: pointer to message
- count: number of elements in ptr
- MPI_TYPE: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- destination/source: rank of receiver/sender
- tag: unique id for message pair
- Communicator: MPI_COMM WORLD or user created
- MPI_Request: Identify comm operations

How to tell if message is completed?

- `int MPI_Wait(MPI_Request *request,MPI_Status *status);`
- `MPI_WAIT(INTEGER REQUEST,INTEGER STATUS(MPI_STATUS_SIZE),INTEGER ERROR)`
- `int MPI_Waitall(int count,MPI_Request *array_of_requests, MPI_Status *array_of_statuses);`
- `MPI_WAITALL(INTEGER COUNT,INTEGER ARRAY_OF_REQUESTS(*),INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),INTEGER ERROR)`

Also: `MPI_Waitany`, `MPI_Test` ...

MPI: Wait & Waitall

Will block until the communication(s) complete

```
err = MPI_Wait(MPI_Request *, MPI_Status *)
err = MPI_Waitall(count, MPI_Request *, MPI_Status*)
```

- MPI_Request: Identify comm operation(s)
- MPI_Status: Status of comm operation(s)
- count: Number of comm operations(s)

MPI: Test

- Does not block, returns immediately
- Provides another mechanism for overlapping communication and computation.

```
err = MPI_Test(MPI_Request *, flag, MPI_Status *)
```

- `MPI_Request`: Identify comm operation(s)
- `MPI_Status`: Status of comm operation(s)
- `flag`: true if comm complete; false if not sent/recv yet

Hands On

- In diffusion directory, cp diffusion{c,f}-mpi.{c,f90} to diffusion{c,f}-mpi-nonblocking.{c,f90}
- Change to do non-blocking IO; post sends/recvs, do inner work, wait for messages to clear, do end points

Tips

Debugging

- Reduce problem size: For example reduce the domain size from 1000 points to 10 or 11, as appropriate. This makes easier to inspect all array elements and pinpoint errors in logic.
- Do only one iteration in the loop: If there is an obvious logic error it will show up in the first step.
- Use only 2 and then 3 MPI tasks initially: Easy to follow the computations and make sure your logic doesn't depend on number parity.
- Use a parallel debugger: it is very helpful when inspecting the logic of the code and following the values being computed step by step. For example, use ARM DDT (https://developer.arm.com/downloads/-/arm-forge) on Niagara.

Scaling

- Do weak scaling: prepare your simulation for one node. Add more nodes but keep the memory footprint per node fixed. Time it. Repeat 3 to 5 times. Select the shortest time.
- Do strong scaling: prepare your simulation for one node. Increase the number of nodes keeping the problem size the same as the original one. Time it. Repeat 3 to 5 times. Select the shortest time.

Conclusion

Recap

- Scientific MPI Example: 1D Diffusion Equation
- Non-blocking communications

Good References

- W. Gropp, E. Lusk, and A. Skjellun, Using MPI: Portable Parallel Programming with the Message-Passing Interface. Third Edition. (MIT Press, 2014).
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface. (MIT Press, 2014).
- The man pages for various MPI commands.
- http://www.mpi-forum.org/docs/

MPI Summary

MPI Summary - C syntax

```
MPI_Status status;

err = MPI_Init(&argc, &argv);

err = MPI_Comm_{size,rank}(Communicator, &{size,rank});

err = MPI_Send(sendptr, count, MPI_TYPE, destination, tag, Communicator);

err = MPI_Isend(sendptr, count, MPI_TYPE, destination, tag, Communicator, MPI_Request);

err = MPI_Recv(rcvptr, count, MPI_TYPE, source, tag, Communicator, &status);

err = MPI_Irecv(rcvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Request);

err = MPI_Sendrecv(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source,
                   tag, Communicator, &status);

err = MPI_Allreduce(&mydata, &globaldata, count, MPI_TYPE, MPI_OP, Communicator);

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_FLOAT, MPI_DOUBLE, MPI_INT, MPI_CHAR...
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```

MPI Summary - FORTRAN syntax

```
integer status(MPI_STATUS_SIZE)

call MPI_INIT(err)

call MPI_COMM_{SIZE,RANK}(Communicator, {size,rank},err)

call MPI_SSEND(sendarr, count, MPI_TYPE, destination, tag, Communicator)

call MPI_ISEND(sendarr, count, MPI_TYPE, destination, tag, Communicator, request, err)

call MPI_RECV(rcvarr, count, MPI_TYPE, destination,tag, Communicator, status, err)

call MPI_IRECV(rcvarr, count, MPI_TYPE, destination, tag, Communicator, request, err)

call MPI_SENDRECV(sendptr, count, MPI_TYPE, destination,tag, recvptr, count, MPI_TYPE, source, &
                  tag, Communicator, status, err)

call MPI_ALLREDUCE(mydata, globaldata, count, MPI_TYPE, MPI_OP, Communicator, err)

Communicator -> MPI_COMM_WORLD
MPI_Type -> MPI_REAL, MPI_DOUBLE_PRECISION, MPI_INTEGER, MPI_CHARACTER
MPI_OP -> MPI_SUM, MPI_MIN, MPI_MAX,...
```