Advanced Topics in Numerical Analysis: High Performance Computing

Cross-listed as MATH-GA.2012-001 and CSCI-GA 2945.001

Benjamin Peherstorfer
Courant Institute, NYU
pehersto@cims.nyu.edu

Spring 2020, Monday, 1:25–3:15PM, WWH #512

April 20, 2020

Slightly adapted from Georg Stadler's lectures.

Today

Last time

- GPU: Reduction and convolution
- Start with MPI

Today

- MPI

Announcements

- Homework 5 has been posted and is due May 4
- Extra office hours to discuss final projects instead of lecture on April 27 (see next slide)

Final projects

Final projects

- Instead of having a lecture and office hour on April 27, there will be extra office hours for discussing the final projects
  - Fri, April 24 from 12 - 1pm
  - Thu, April 30 from 9 - 10am
- It is strongly encouraged that at least one person from each group gives a brief status report of the final project

Presentations on Monday, May 4

- There are 14 final projects
- Groups get 5 minute time slots for presentation
- Have you slides ready to go! One presenter per group. At most 3-4 slides per group.
- Briefly state the problem you worked on, the challenges, how you approached them, what speedup results you obtained
- We will have time for 1-2 quick questions after each presentation

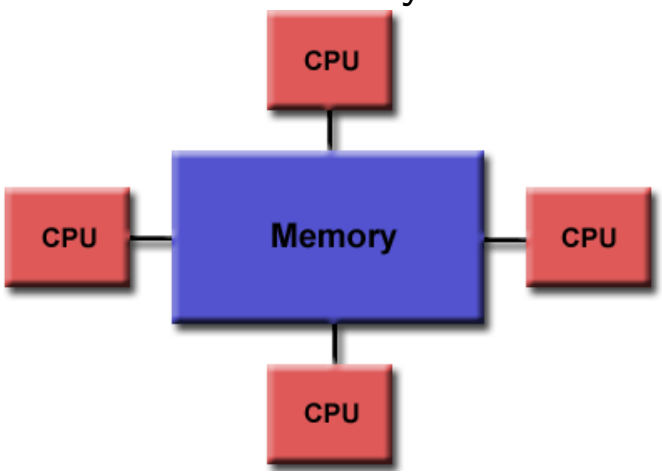
If we are delayed, we finish presentations on Mon, May 11

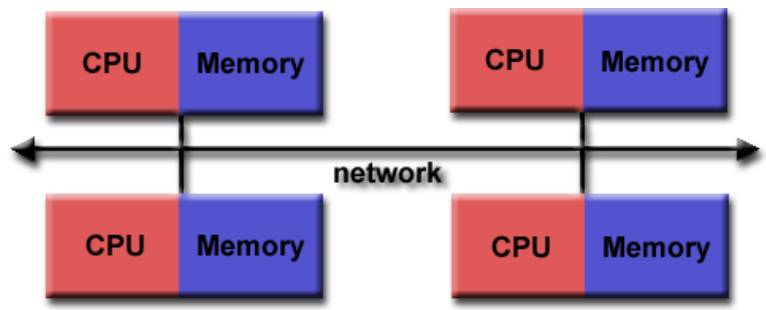
Final projects (and order we go through them on May 4)

May 4	Bharadwaj, Sachin	Parallel linear solvers for PDEs
May 4	Ilango,Jude Naveen Raj Mohan Ram,Agnitha	Parallel algorithms on graphs
May 4	Justin Finkel	Gaussian process regression
May 4	Natesh,Sachin R	Particles immersed in Stokes flow
May 4	Tang,Xuan	Sparse matrix multiplication
May 4	Law,Frederick Toler,Evan Hunter Weady,Scott	Parallel FFT for PDEs and parallel in time
May 4	Wu,Chia-Hao	Molecular dynamics
May 4	Sun,Yue Rapelli,Ilyeech Kishore	Parallel SGD
May 4	Li,Zhen Yu,Cong Xia,Yu	Parallel FFT
May 4	Luo,Maoyi	Parallelize the FBP algorithm
May 4	Bajwa,Gurkirat Singh	Delaunay triangulation
May 4	Braun,Joschka Julius Åstrand,Oliver	JAX
May 4	Kanishka,Ketan	Interpolation points
May 4	Sunwoo,Antony	k-means

Programming models

- Flynn's taxonomy:
  - Single instruction–single data (SISD)
  - Single instruction–multiple data (SIMD)
  - Multiple instruction–multiple data (MIMD)
- Distributed memory vs. shared memory parallelism





- Programming models: OpenMP vs. Message passing interface (MPI); and combinations thereof

Principles of Message-Passing Programming

- The logical view of a machine supporting the message-passing paradigm consists of p processes, each with its own exclusive address space.
- Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) require cooperation of two processes – the process that has the data and the process that wants to access the data.
- These two constraints make underlying costs very explicit to the programmer.

MPI: the Message Passing Interface

The minimal set of MPI routines.

```
MPI_Init (&argc,&argv)
```

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Finalize ()
```

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Comm_size (comm,&size)
```

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD

```
MPI_Comm_rank (comm,&rank)
```

- Returns the rank of the calling MPI process within the specified communicator.
- Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD.
- This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
```

Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
```

Receive a message and block until the requested data is available in the application buffer in the receiving task.

Distributed Memory Performance Model

Postal model:

- Hockney, Jesshope: Parallel Computers 2: architecture, programming and algorithms (1988)
- Simple model for distributed memory point-to-point communication.
- Parameters: latency (t_s), per-word-transfer time (t_w)

$$t_{comm} = t_s + t_w\ m$$

where, m is the message size in bytes.

Messaging patterns

Pingpong: pass messages between two processes

```
for (long repeat  = 0; repeat < Nrepeat; repeat++) {
    MPI_Status status;
    if (repeat % 1 == 0) { // even iterations
      if (rank == proc0)
        MPI_Send(msg, Nsize, MPI_CHAR, proc1, repeat, comm);
      else if (rank == proc1)
        MPI_Recv(msg, Nsize, MPI_CHAR, proc0, repeat, comm,
            &status);
    }
    else { // odd iterations
      if (rank == proc0)
        MPI_Recv(msg, Nsize, MPI_CHAR, proc0, repeat, comm,
            &status);
      else if (rank == proc1)
        MPI_Send(msg, Nsize, MPI_CHAR, proc1, repeat, comm);
    }}
```

- proc0 sends at even iterations and proc1 receives and vice versa for odd iterations
- Measure latency of passing messages

DEMO: mpi05.cpp

Buffering

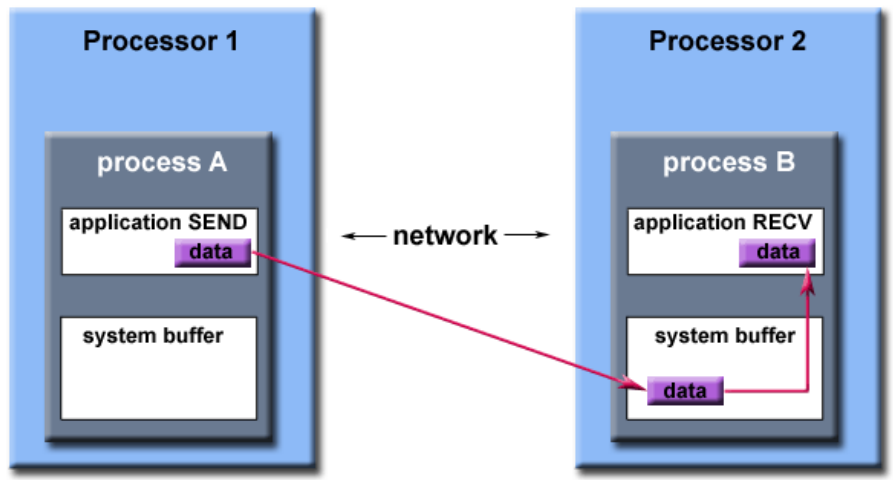
Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case.
- Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.

Examples:

- A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
- Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases.



Path of a message buffered at the receiving process

Blocking:

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Non-blocking:

- Return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Blocking Send	Non-blocking Send

`myvar = 0;`
 	
`for (i=1; i<ntasks; i++) {`
`  task = i;`
`  MPI_Send (&myvar ... ... task ...);`
`  myvar = myvar + 2`
 	
`  /* do some work */`
 	
`  }`
	`myvar = 0;`
	
	`for (i=1; i<ntasks; i++) {`
	`  task = i;`
	`  MPI_Isend (&myvar ... ... task ...);`
	`  myvar = myvar + 2;`
	
	`  /* do some work */`
	
	`  MPI_Wait (...);`
	`  }`

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
```

- Identifies an area in memory to serve as a send buffer.
- Processing continues immediately without waiting for the message to be copied out from the application buffer.
- A communication request handle is returned for handling the pending message status.
- The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
```

- Identifies an area in memory to serve as a receive buffer.
- Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer.
- A communication request handle is returned for handling the pending message status.
- The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

```
MPI_Wait (&request,&status)
```

- MPI_Wait blocks until a specified non-blocking send or receive operation has completed.

```
MPI_Test (&request,&flag,&status)
```

- MPI_Test checks the status of a specified non-blocking send or receive operation.
- The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not.

Deadlock-free send/recv patterns

Blocking send and recv

Process-0	Process-1
MPI_Sendrecv(...)	MPI_Sendrecv(...)
// ··· can use recv array now	// ··· can use recv array now

Blocking send, non-blocking recv

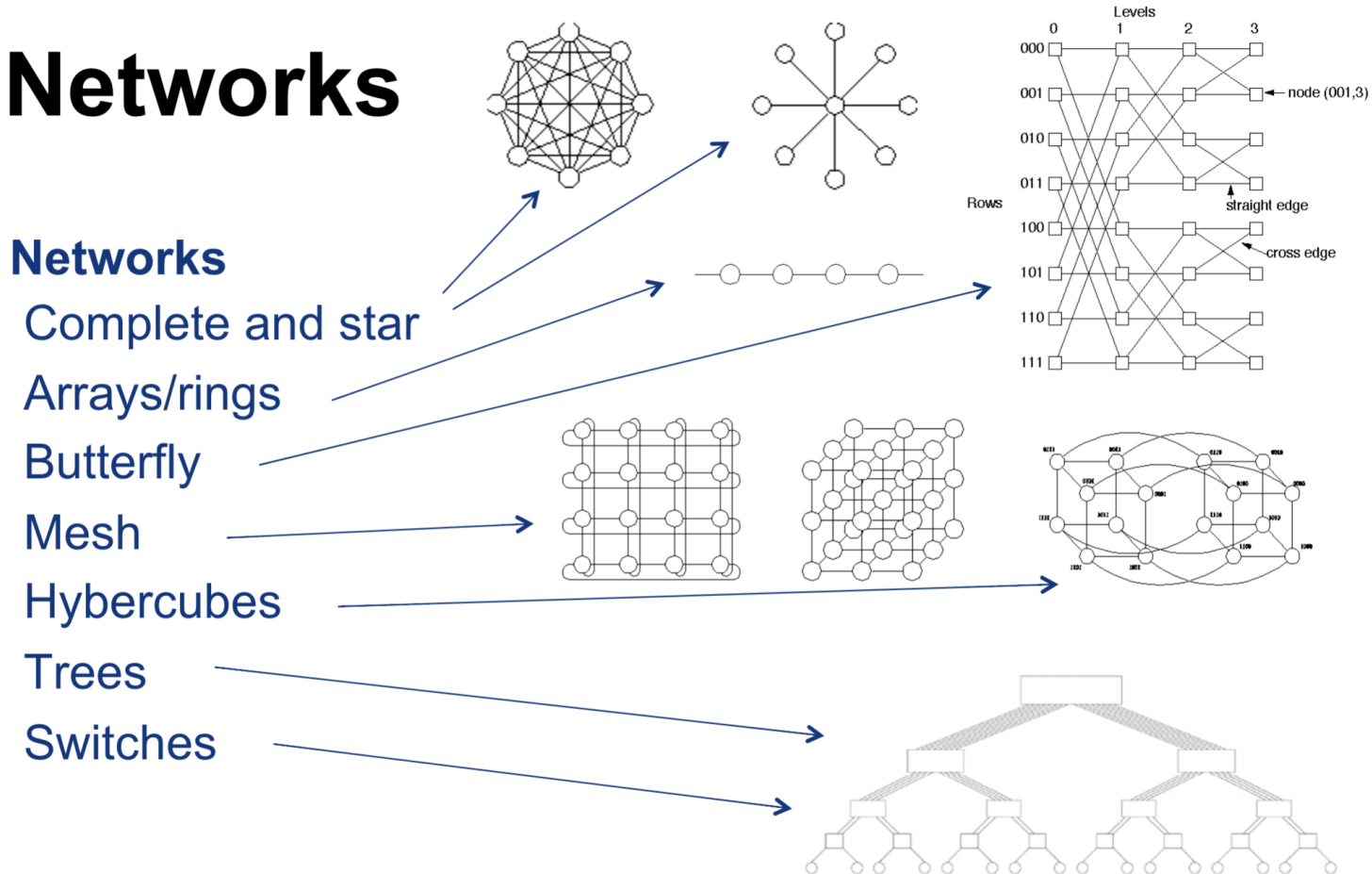
Process-0	Process-1
MPI_Irecv(... , recv_request)	MPI_Irecv(... , recv_request)
MPI_Send(...)	MPI_Send(...)
// ··· other code	// ··· other code
MPI_Wait(recv_request, ...)	MPI_Wait(recv_request, ...)
// ··· can use recv array now	// ··· can use recv array now

DEMO: mpi07.cpp

MPI Collectives

- Calls that involve more than 2 processes (also called point-to-point)
- Could also be done with Sends and Recvs, but more efficient and concenient
- Can be one-to-all or all-to-all
- Every process needs to see the collective call to avoid hangs!
- Actual implementation depends on MPI library (and possibly on the network type)

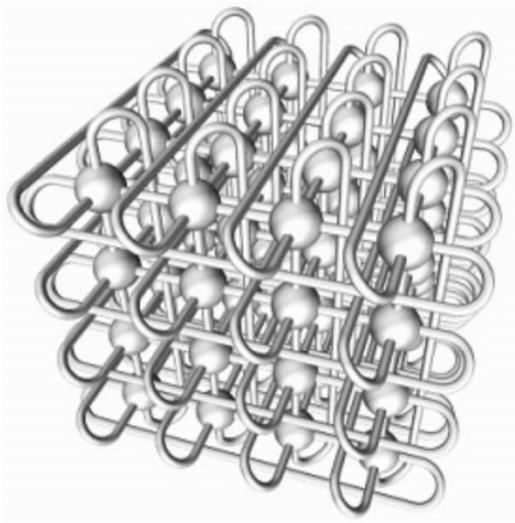
Network types: topologies



Network types: metrics

- Diameter: maximum distance between any two nodes
- Connectivity: number of links needed to remove to isolate a node
- Bisection width: number of links to be removed to break network into equal parts
- Cost: Total number of links

Examples

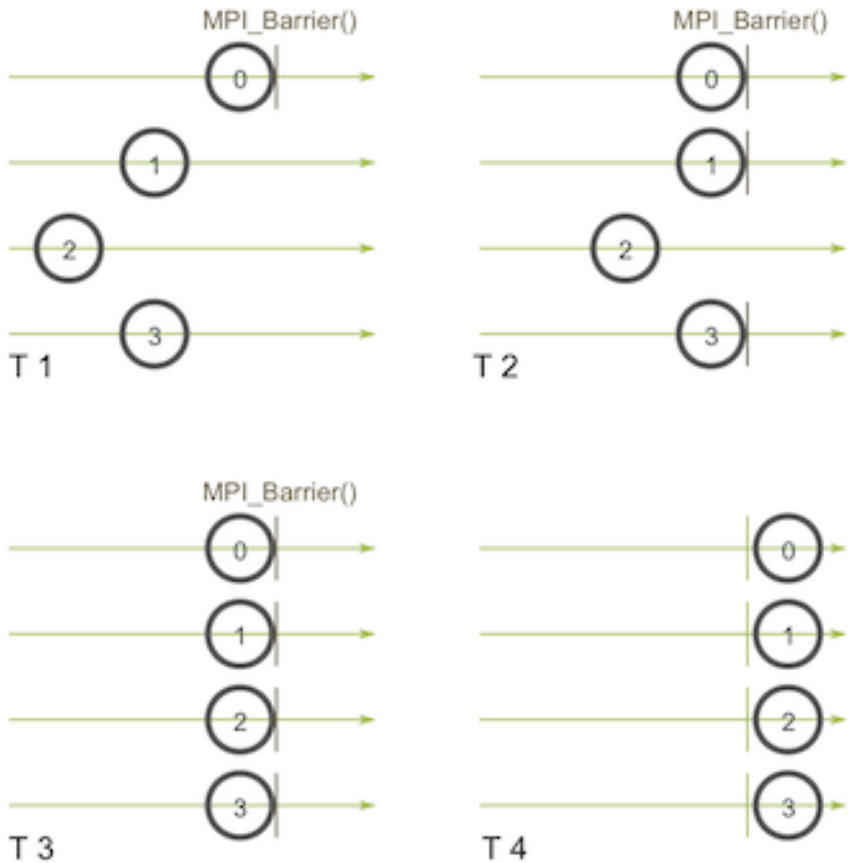


Network	Diameter	Bisection Width	Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

MPI Barrier

Synchronizes all processes. Other collective functions implicitly act as a synchronization. Used for instance for timing.

`MPI_Barrier(MPI_Comm communicator)`



How would you implement a Barrier?

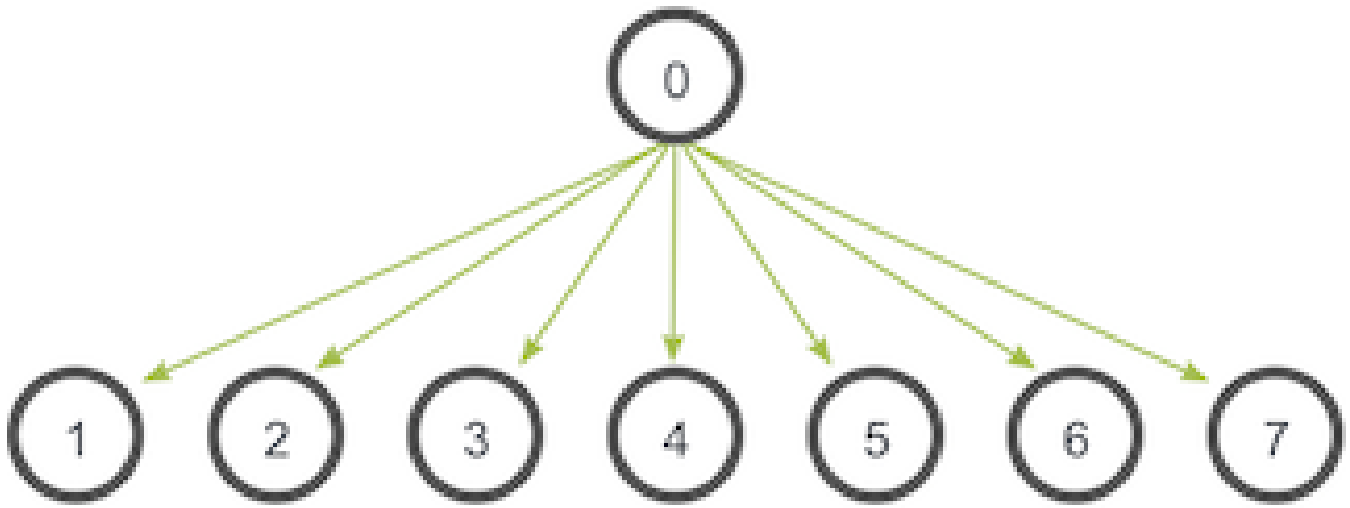
How would you implement a Barrier?

- Program that passed a token around all processes in a ring-like fashion.
- This type of program is one of the simplest methods to implement a barrier since a token can't be passed around completely until all processes work together.

MPI Broadcast

Broadcasts data from one to all processors. Every processor calls same function (although its effect is different).

```
MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root,
MPI_Comm communicator)
```



Actual implementation depends on MPI library.

Our own broadcast

The root process sends the data to everyone else while the others receive from the root process:

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int
    root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

DEMO: mpi08.cpp

Our own broadcast

The root process sends the data to everyone else while the others receive from the root process:

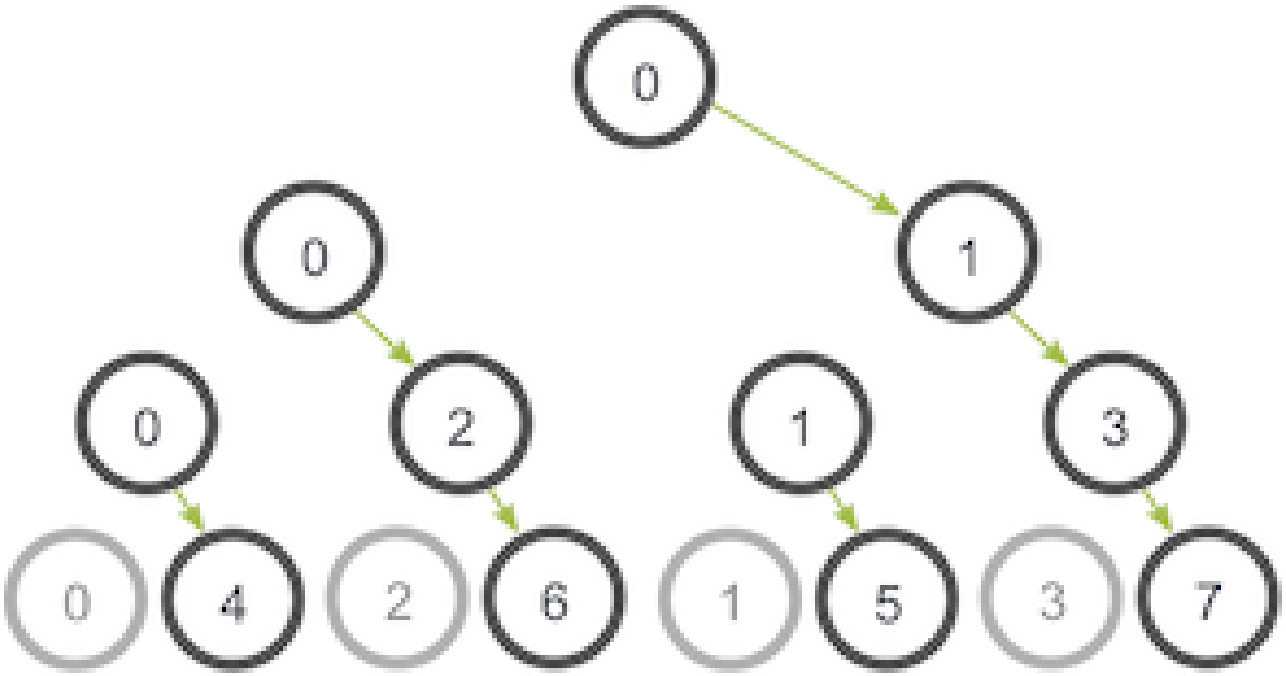
```
void my_bcast(void* data, int count, MPI_Datatype datatype, int
    root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

DEMO: mpi08.cpp

Question: Is our broadcast efficient? DEMO: mpi09.cpp

MPI_Bcast uses a tree structure to broadcast



The network utilization doubles at every subsequent stage of the tree communication until all processes have received the data.

MPI Reduce

Reduces data from all to one processors. Every processor calls same function.

```
MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm communicator)
```

Possible Reduce operators:

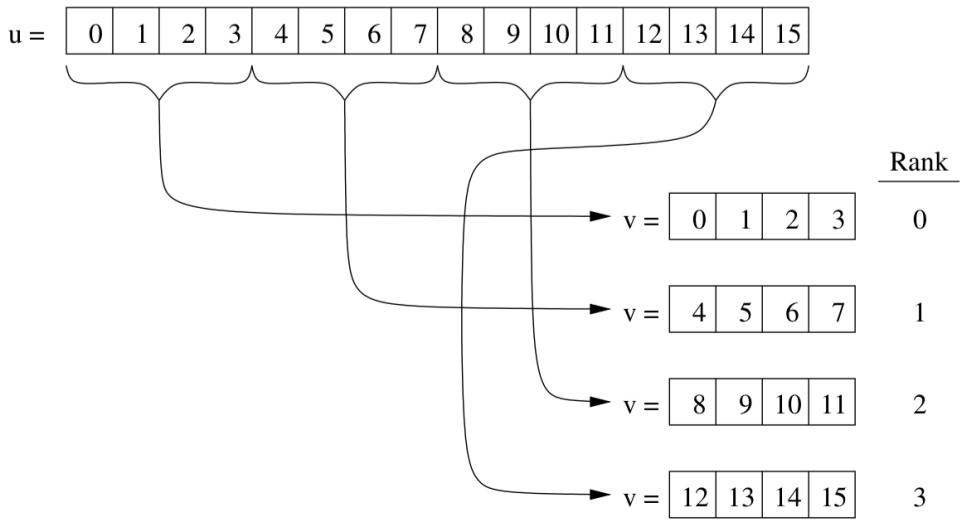
- MPI_MAX: Returns the maximum element.
- MPI_MIN: Returns the minimum element.
- MPI_SUM: Sums the elements.
- MPI_PROD: Multiplies all elements.
- MPI_LAND: Performs a logical and across the elements.
- MPI_LOR: Performs a logical or across the elements.
- MPI_BAND: Performs a bitwise and across the bits of the elements.
- MPI_BOR: Performs a bitwise or across the bits of the elements.
- MPI_MAXLOC: Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC: Returns the minimum value and the rank of the process that owns it.

DEMO: mpi10.cpp, dot-product

MPI Scatter

Broadcasts different data from one to all processors. Every processor calls same function.

```
MPI_Scatter(void* sendbuff, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm communicator)
```



Send arguments must be provided on all processors, but sendbuf can be NULL. Send/recv count are per processor. Variable-sized variant is `MPI_Scatterv`.

MPI Gather

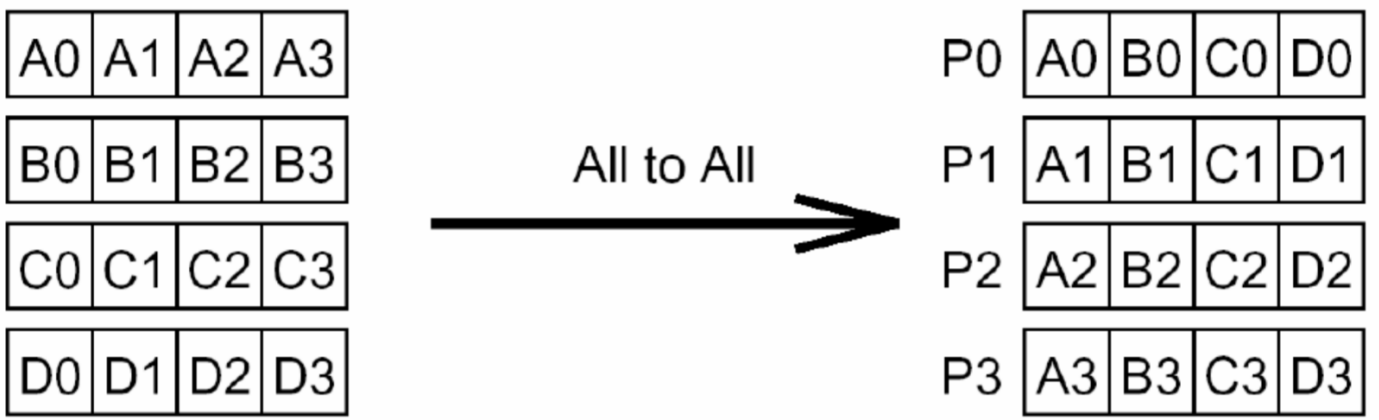
Gathers different data from all to one processors. Every processor calls same function. Gather is (more or less) the opposite of scatter.

```
MPI_Gather(void* sendbuff, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm communicator)
```

MPI All-to-all

Shares data from each to each processor.

```
MPI_Alltoall(void *sendbuf, int count, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm
comm)
```



Example: matrix-transpose. Variable-sized variant is called `MPI_Alltoallv`.
DEMO: mpi11.cpp, transpose matrix

Example: Jacobi solver

Jacobi

DEMO: mpi12.cpp

Example: Jacobi solver

Jacobi
DEMO: mpi12.cpp

Jacobi non-blocking
DEMO: mpi13.cpp

Example: Jacobi solver

Jacobi
DEMO: mpi12.cpp

Jacobi non-blocking
DEMO: mpi13.cpp

Jacobi hybrid
DEMO: mpi14.cpp

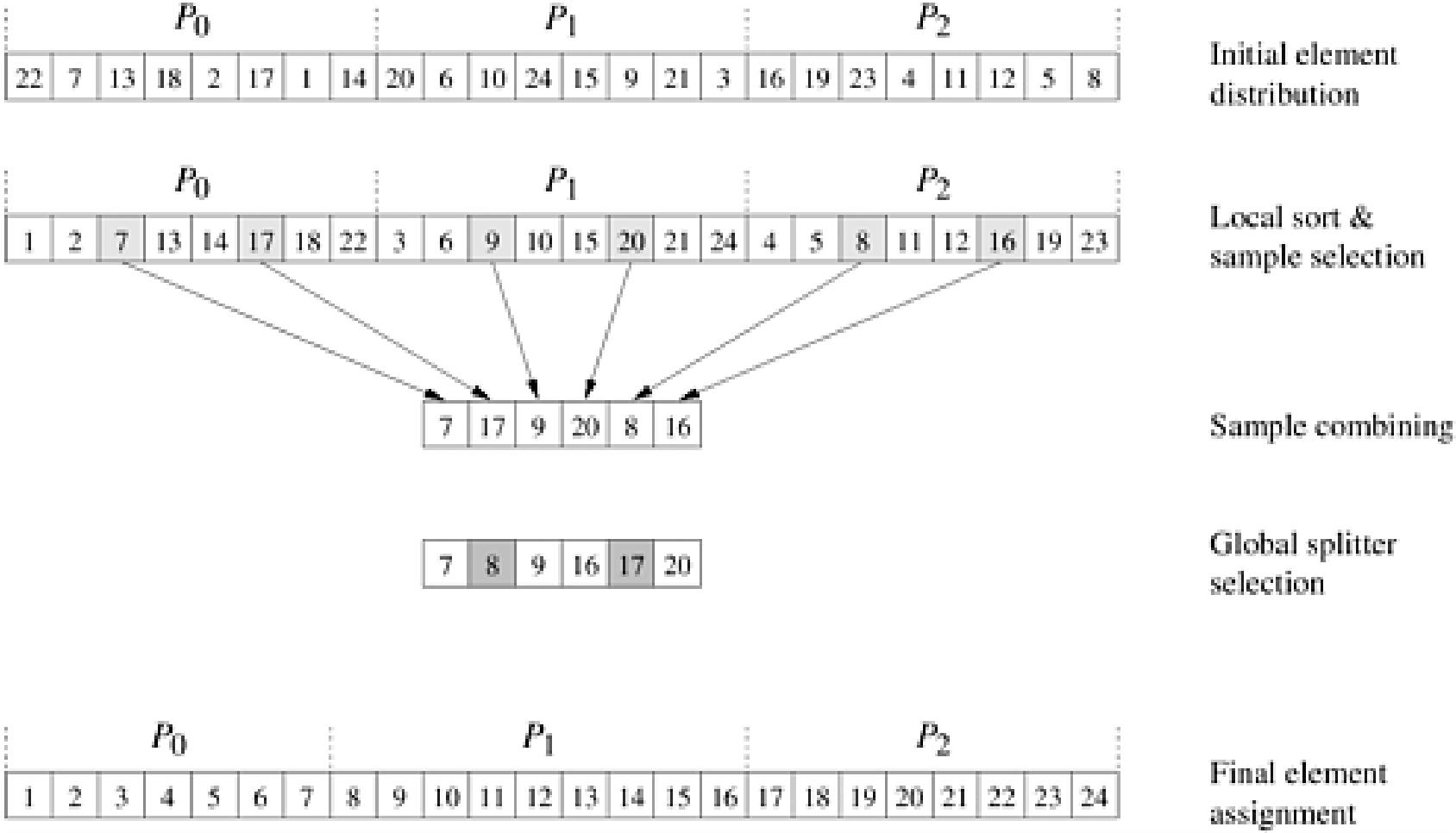
Sample sort

Bucket sort

- The range $[a, b]$ of input numbers is divided into m equal sized intervals, called buckets.
- Each element is placed in its appropriate bucket.
- If the numbers are uniformly divided in the range, the buckets can be expected to have roughly identical number of elements.
- Elements in the buckets are locally sorted.

Steps for parallel bucket sort (sample sort)

- *Select local samples:* Each of the P processors sorts the local array and selects a set of S entries(typically $S = P - 1$) uniformly and communicates these entries to the root processor, who sorts the resulting SP entries and determines $P - 1$ splitters $\{S_1, \ldots, S_{P-1}\}$, which it broadcasts to all other processors.
- *Distribute to buckets:* Each processor determines the "buckets" to which each of its N elements belong; for instance, the first bucket contains all the numbers $\leq S_1$, the second one are all the entries that are in $(S_1, S_2]$ and so on. The numbers contained in each bucket are then communicated; processor 0 receives every processor's first bucket, processor 1 gets processor's second bucket, and so on.
- *Local sort:* Each processor uses a local sort and writes the result to disc.



[Figure: http://parallelcomp.uw.hu/ch09lev1sec5.html]

How solve large linear systems?

Many linear solvers are available: factorization-based solvers (LU, Choleski), fast direct solvers (for specific problems), Krylov solvers (CG, MINRES, GMRES,...), optimal complexity ($\mathcal{O}(n)$) solvers for certain problems (multigrid, FMM)

Solver choice depends on:

- is the system sparse or dense? how sparse?
- symmetric? positive definite? explicitly available?
- properties of the matrix? what do I know about the eigenvalues?
- do I have a good preconditioner?
- do I need the exact solution or can I allow for ε-errors?
- what computing resources do I have? can I store the matrix?
- how fast/often do I need to solve systems?

Partitioning and Load Balancing

Thanks to Marsha Berger for letting me use many of her slides. Thanks Zoltan website for many of these slides and pictures.

https://cs.sandia.gov/Zoltan/Zoltan.html

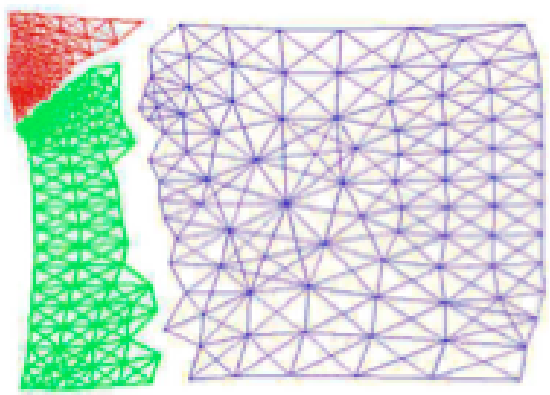
Partitioning

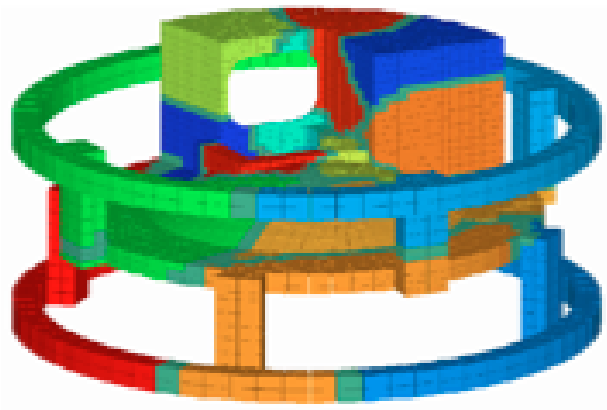
- Decompose computation into tasks to equi-distribute the data and work, minimize processor idle time.

  applies to grid points, elements, matrix rows, particles, ...

- Map to processors to keep interprocessor communication low.

  communication to computation ratio comes from both the partitioning and the algorithm.

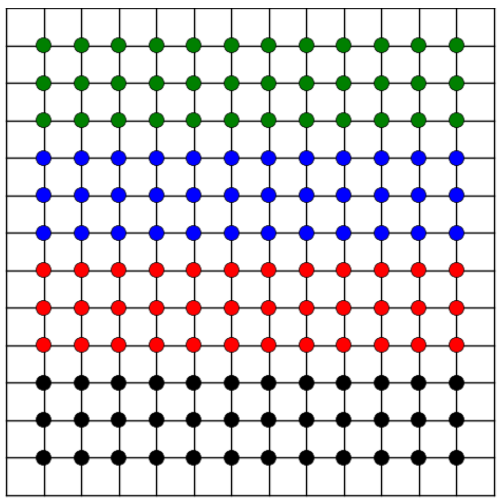


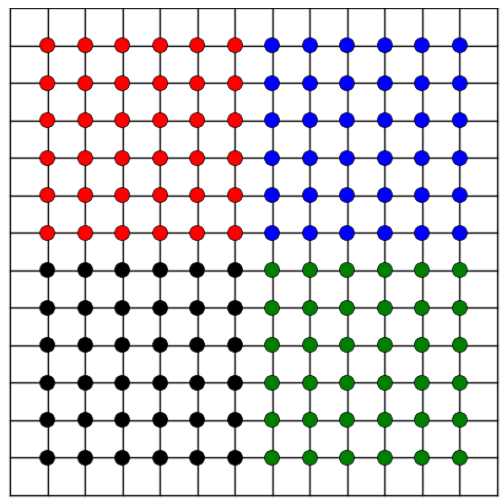


Partitioning

Data decomposition + Owner computes rule:

- Data distributed among the processors
- Data distribution defines work assignment
- Owner performs all computations on its data.
- Data dependencies for data items owned by different processors incur communication





Partitioning

- Static - all information available before computation starts

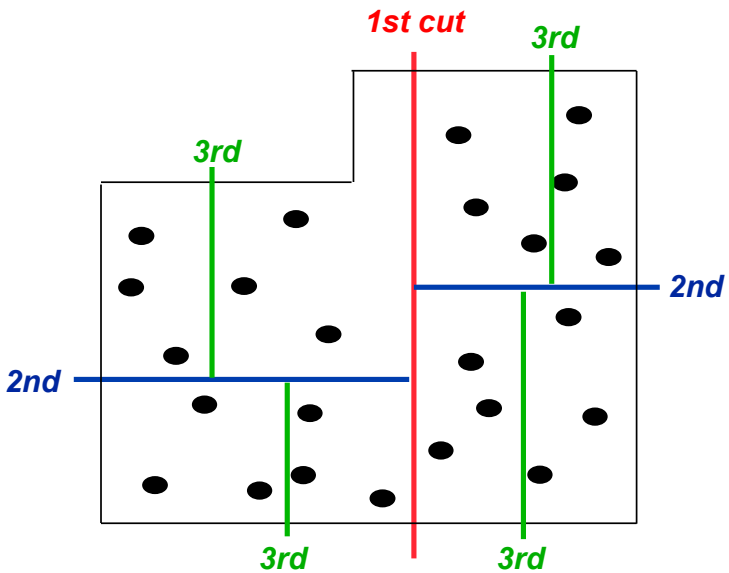
  use off-line algorithms to prepare before execution time; run as pre-processor, can be serial, can be slow and expensive, starts.

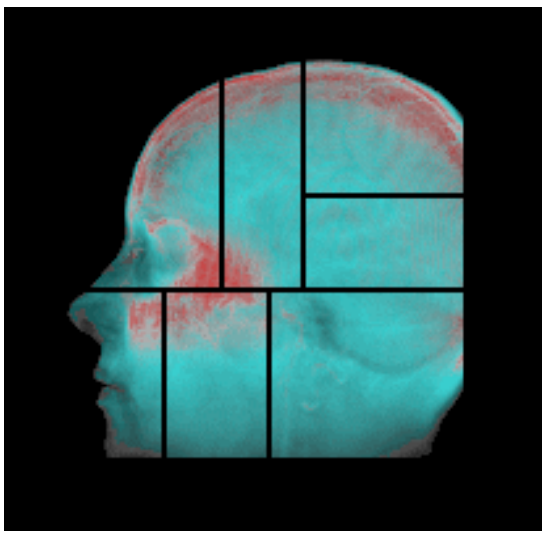
- Dynamic - information not known until runtime, work changes during computation (e.g. adaptive methods), or locality of objects change (e.g. particles move)

  use on-line algorithms to make decisions mid-execution; must run side-by-side with application, should be parallel, fast, scalable. Incremental algorithm preferred (small changes in input result in small changes in partitions)

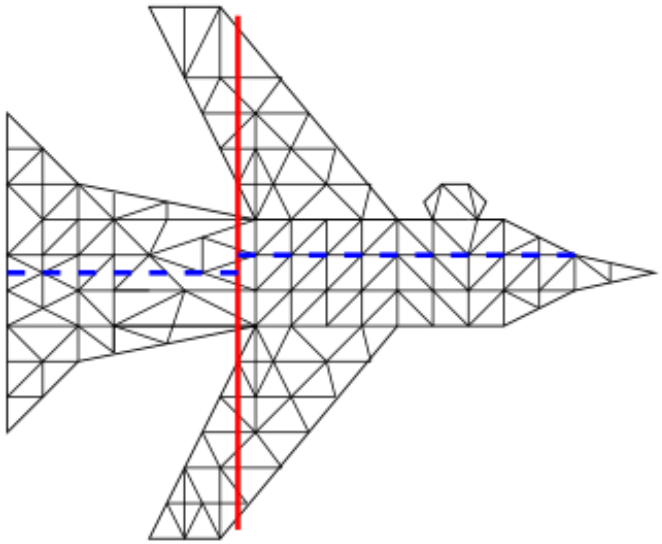
Recursive Coordinate Bisection

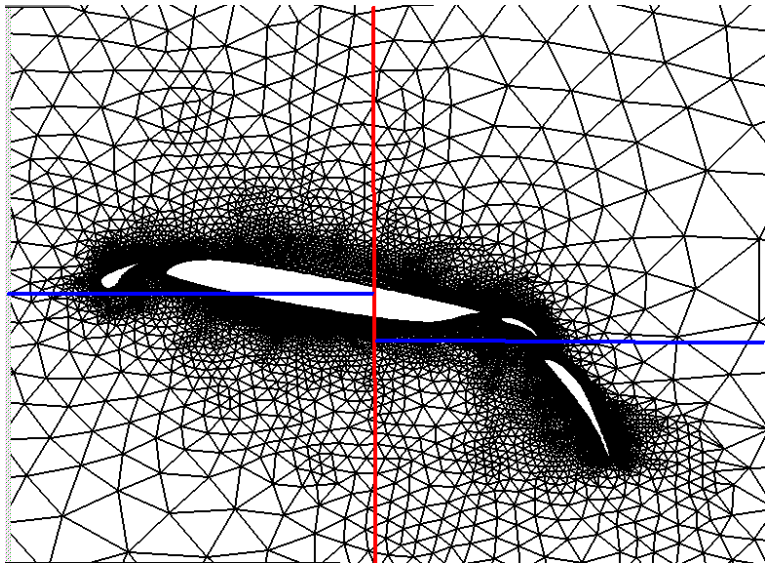
Divide work into two equal parts using cutting plane orthogonal to coordinate axis For good aspect ratios cut in longest dimension.





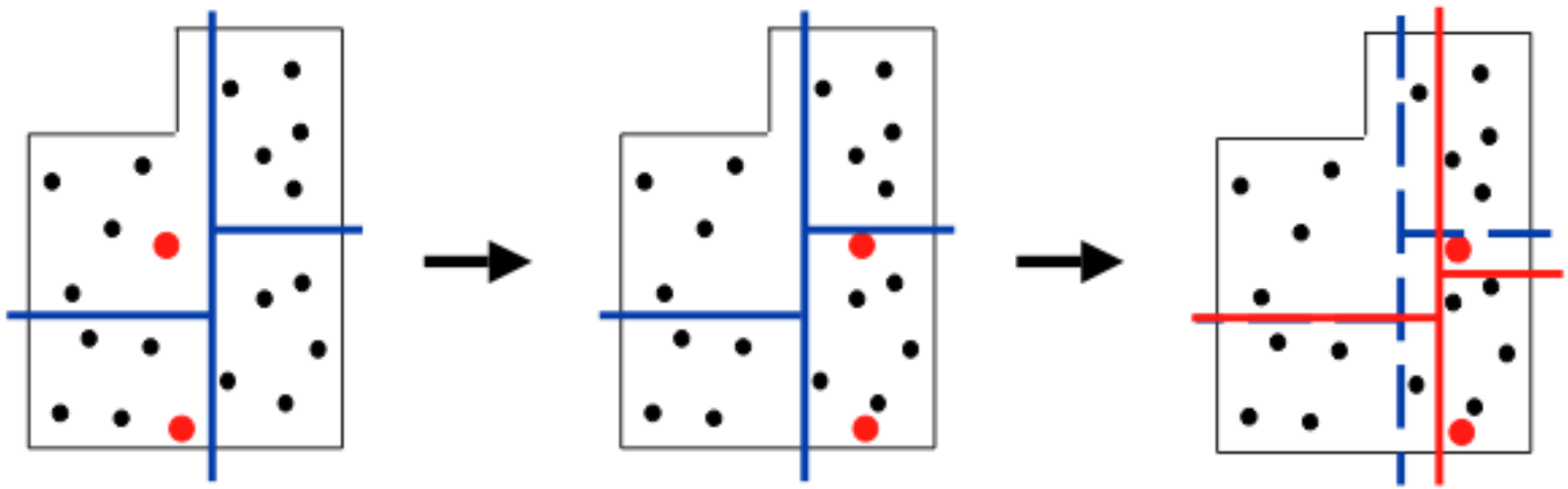
Recursive Coordinate Bisection





+ Conceptually simple, easy to implement, fast.
+ Regular subdomains, easy to describe
– Need coordinates of mesh points/particles.
– No control of communication costs.
– Can generate disconnected subdomains

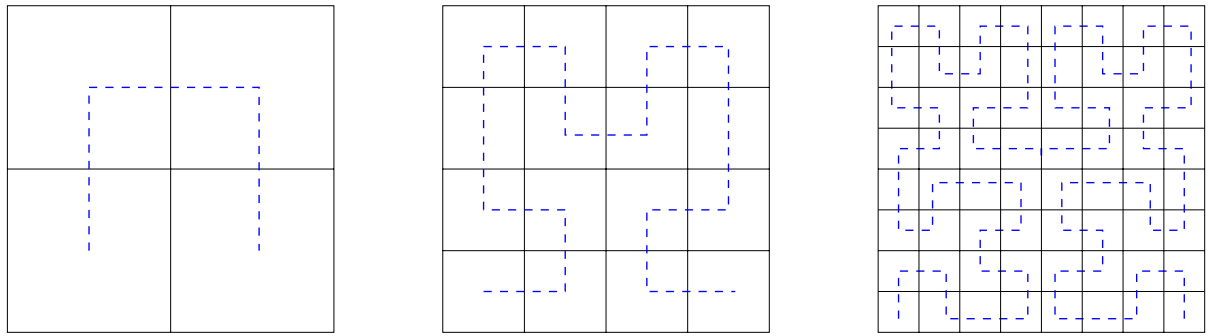
Recursive Coordinate Bisection



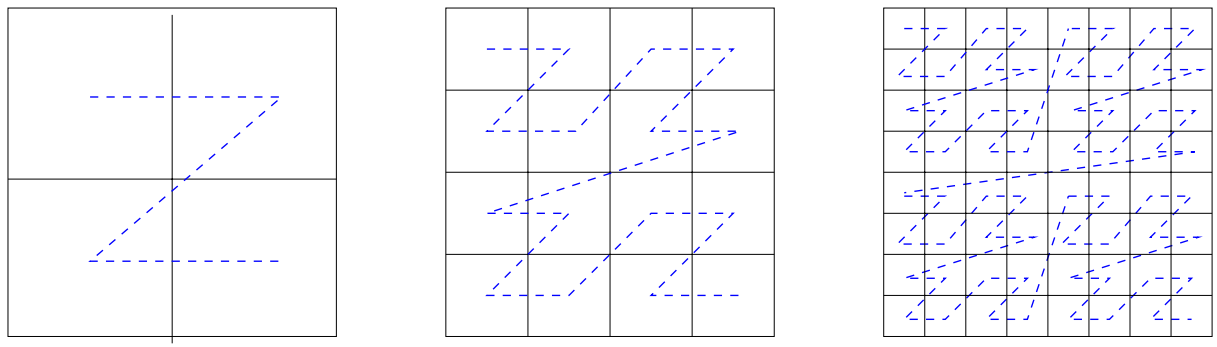
Implicitly incremental - small changes in data result in small movement of cuts

Space-filling Curves

Linearly order a multidimensional mesh (nested hierarchically, preserves locality)



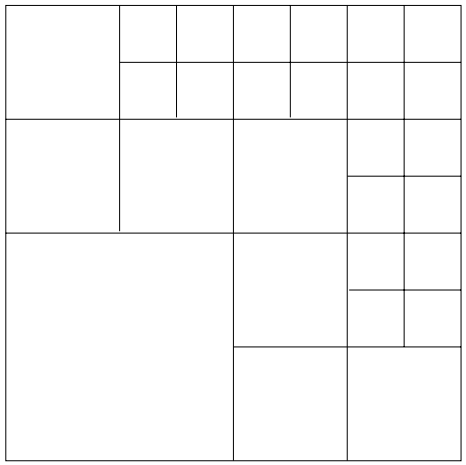
Peano-Hilbert ordering

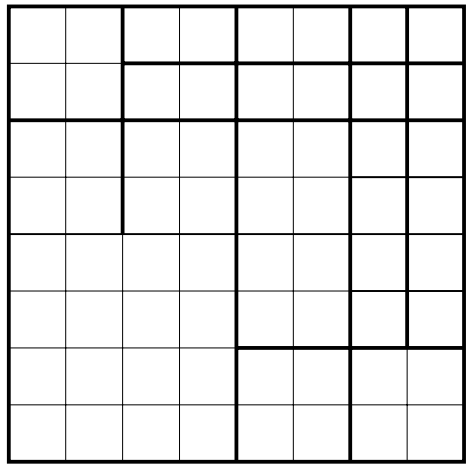


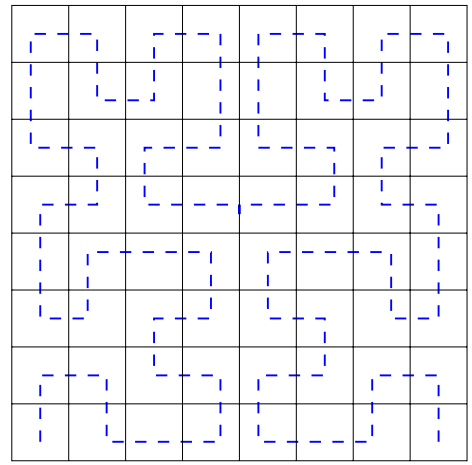
Morton ordering

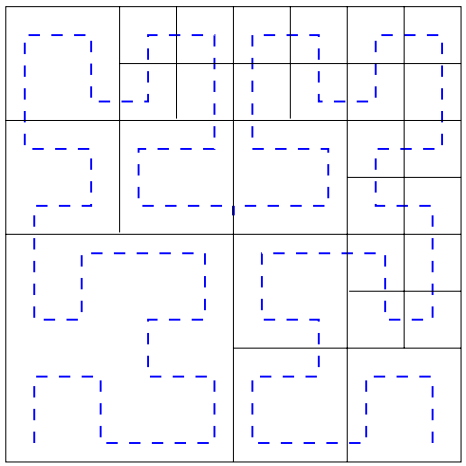
Space-filling Curves

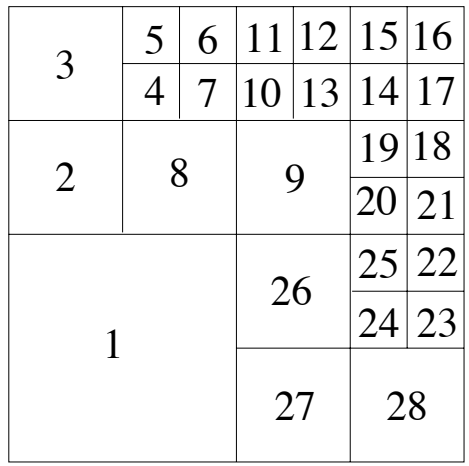
Easily extends to adaptively refined meshes



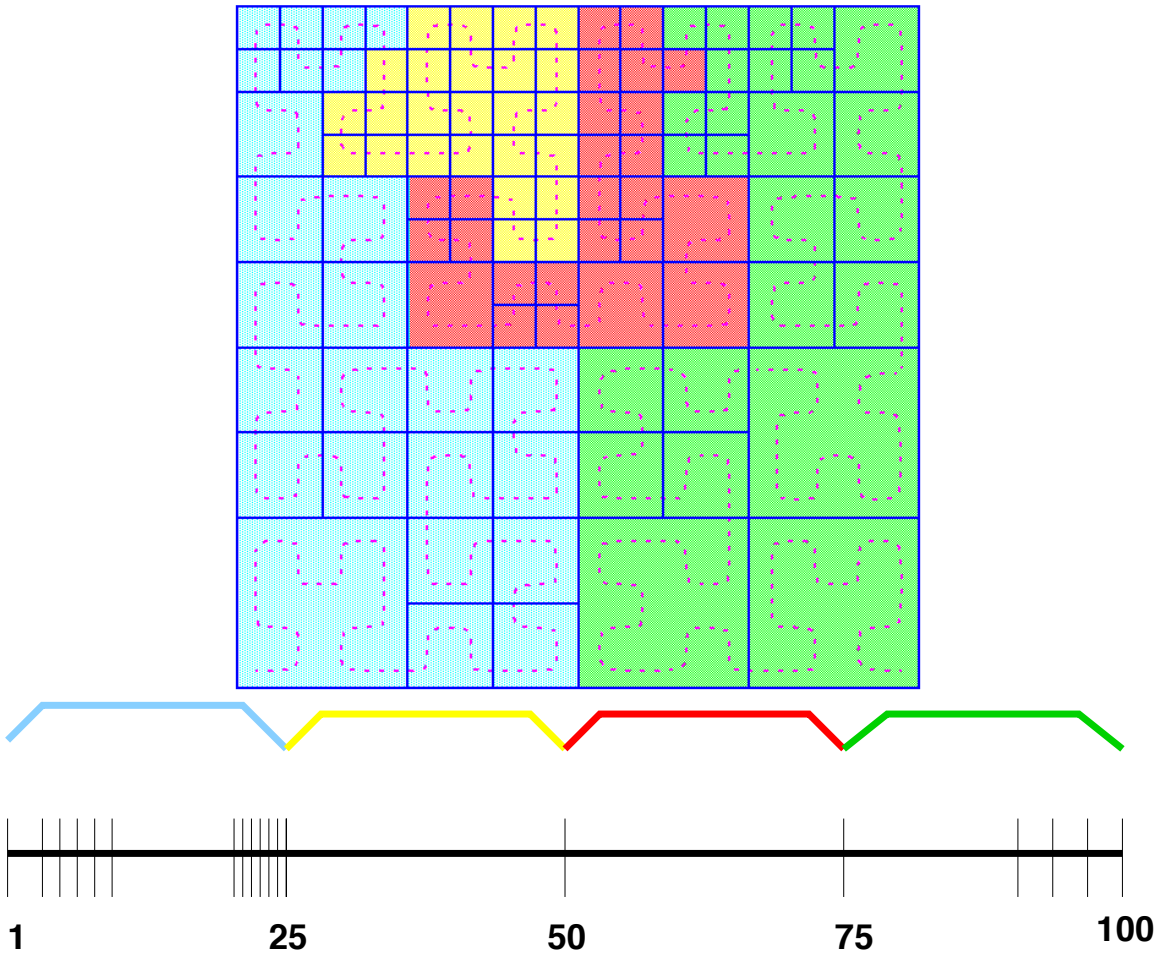






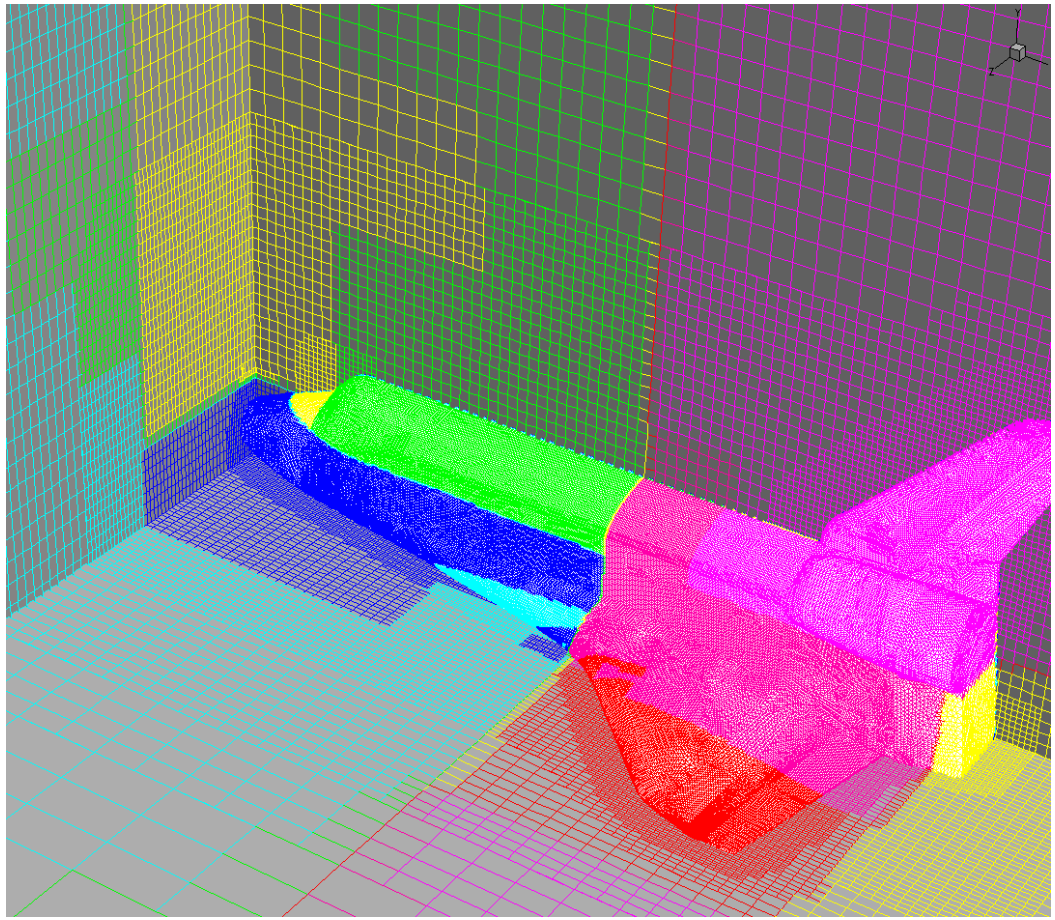


Space-filling Curves



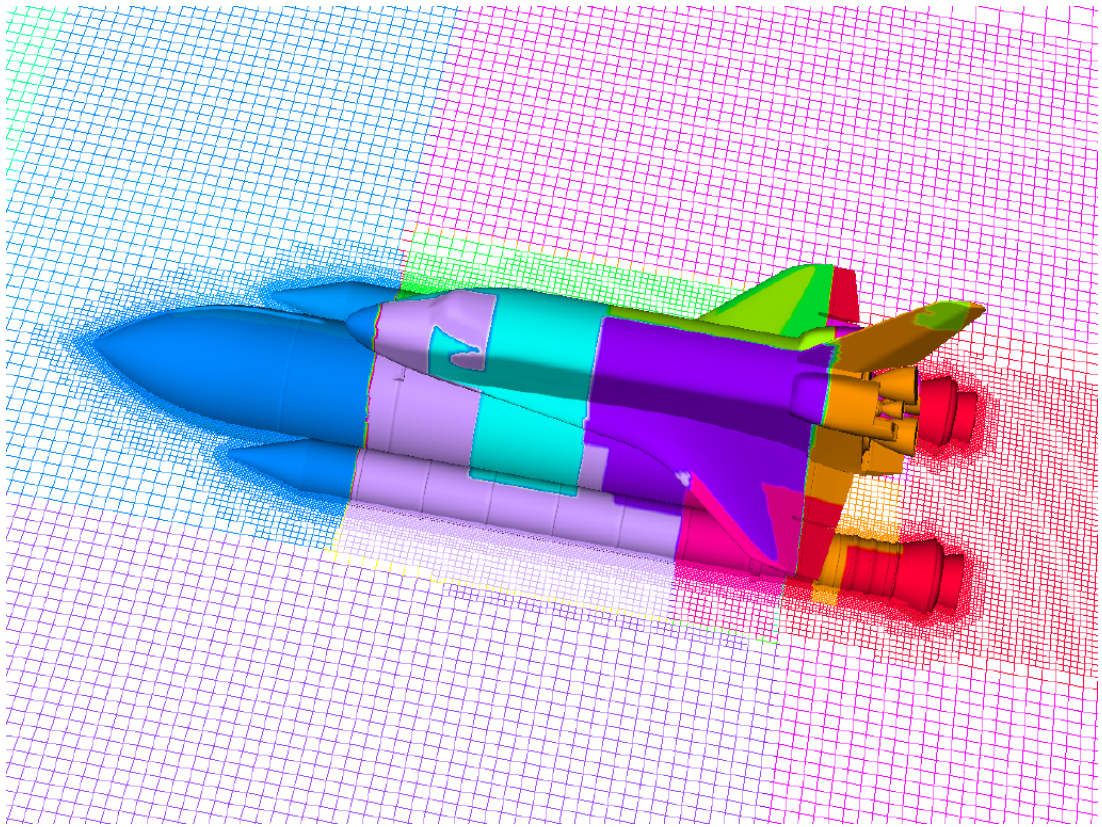
Partition work into equal chunks.

Space-filling Curves



+ Generalizes to uneven work loads - incorporate weights.
+ Dynamic on-the-fly partitioning for any number of nodes.
+ Good for cache performance

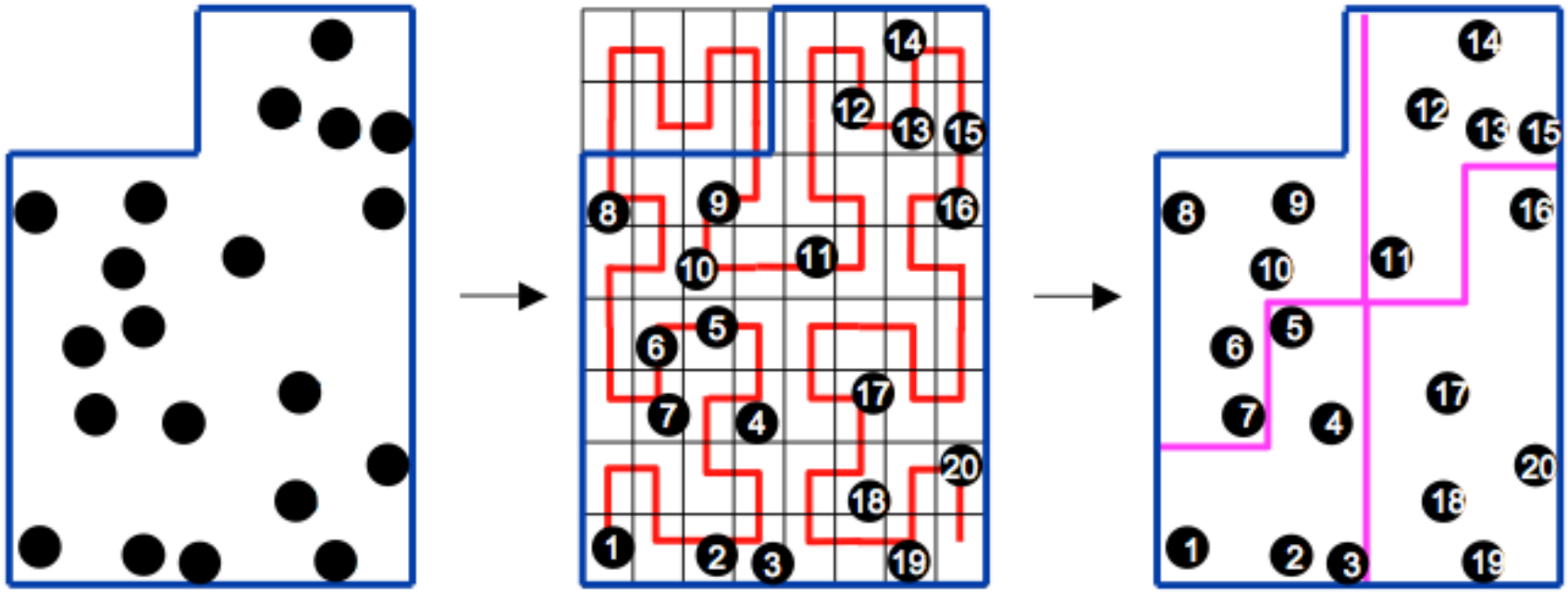
Space-filling Curves



- Red region has more communication - not compact
- Need coordinates

Space-filling Curves

Generalizes to other non-finite difference problems, e.g. particle methods, patch-based adaptive mesh refinement, smooth particle hydro.,



Space-filling Curves

Implicitly incremental - small changes in data results in small movement of cuts in linear ordering