Advanced Topics in Numerical Analysis: High Performance Computing

More distributed memory algorithms

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

April 29, 2019

Outline

Organization issues

Summary of previous class

Distributed memory algorithms

Partitioning and Load Balancing

Organization

Scheduling:

- Homework assignment #5 due tonight; one last assignment posted this week (due in 2 weeks)
- How are your final projects coming along?

Topics today:

- Collective communication in MPI: more examples
- Algorithms: Sorting
- Partitioning and balancing

Outline

Organization issues

Summary of previous class

Distributed memory algorithms

Partitioning and Load Balancing

MPI Collectives

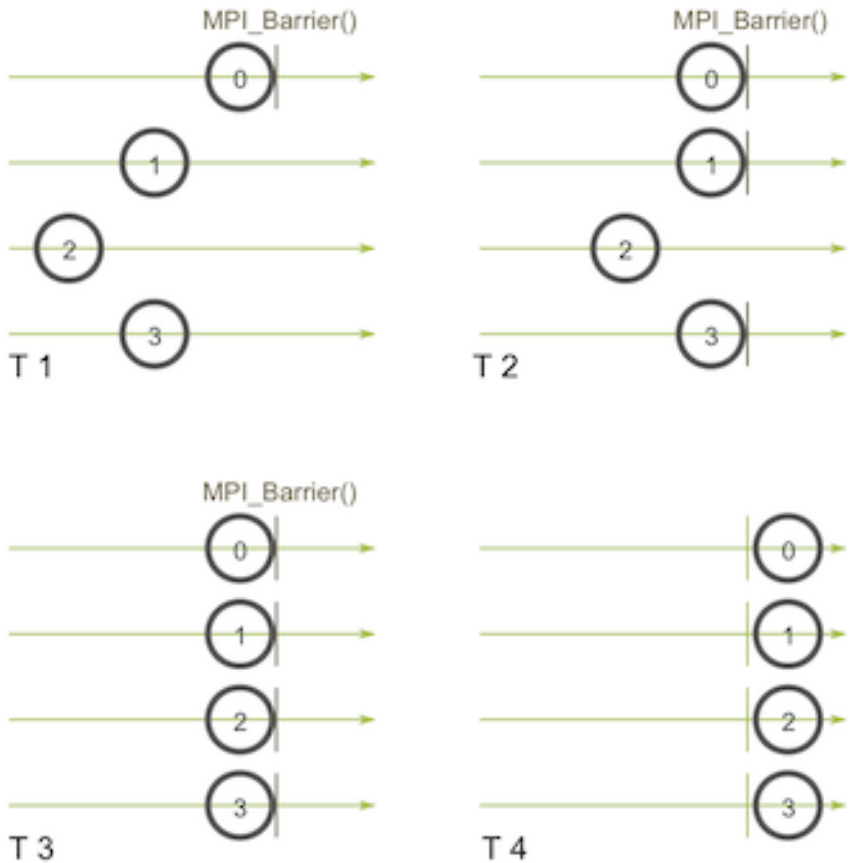
- Calls that involve more than 2 processes (also called point-to-point)
- Could also be done with Sends and Recvs, but more efficient and concenient
- Can be one-to-all or all-to-all
- Every process needs to see the collective call to avoid hangs!
- Actual implementation depends on MPI library (and possibly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

Tutorial: http://mpitutorial.com/tutorials/

MPI Barrier

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

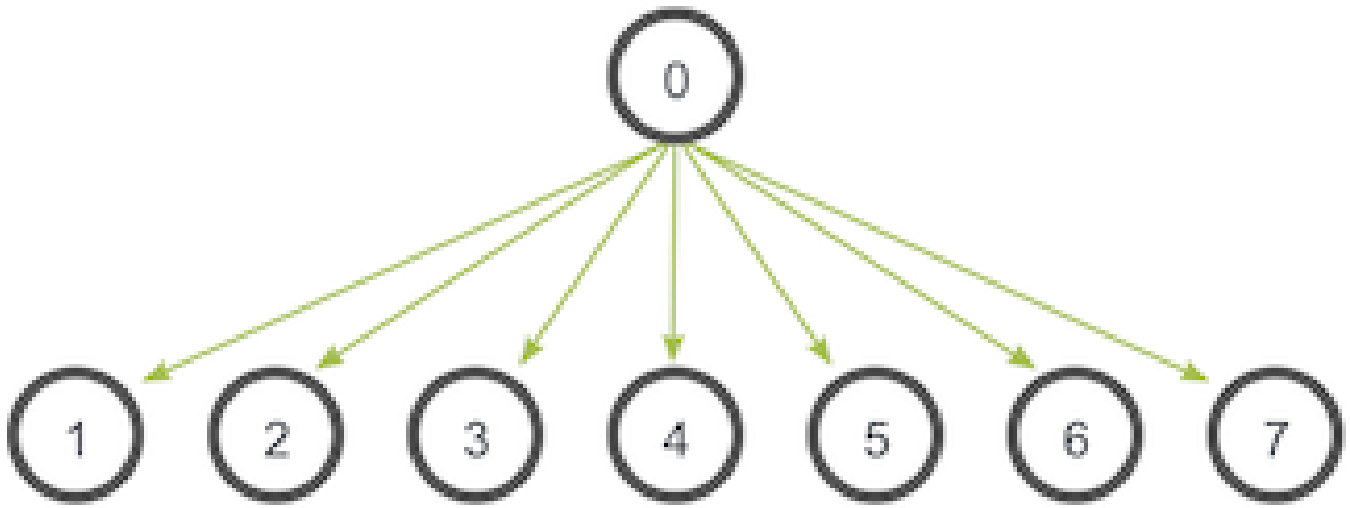
`MPI_Barrier(MPI_Comm communicator)`



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)
```

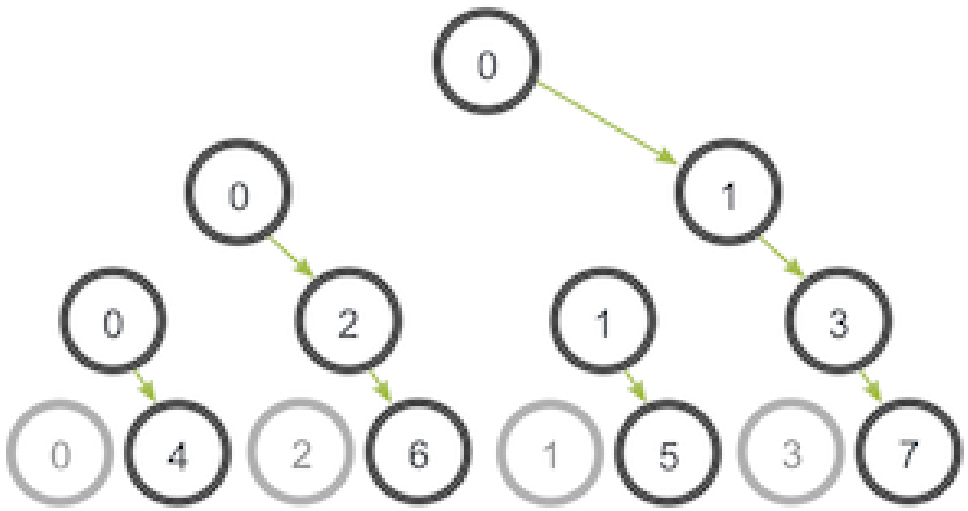


Actual implementation depends on MPI library.

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.

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:

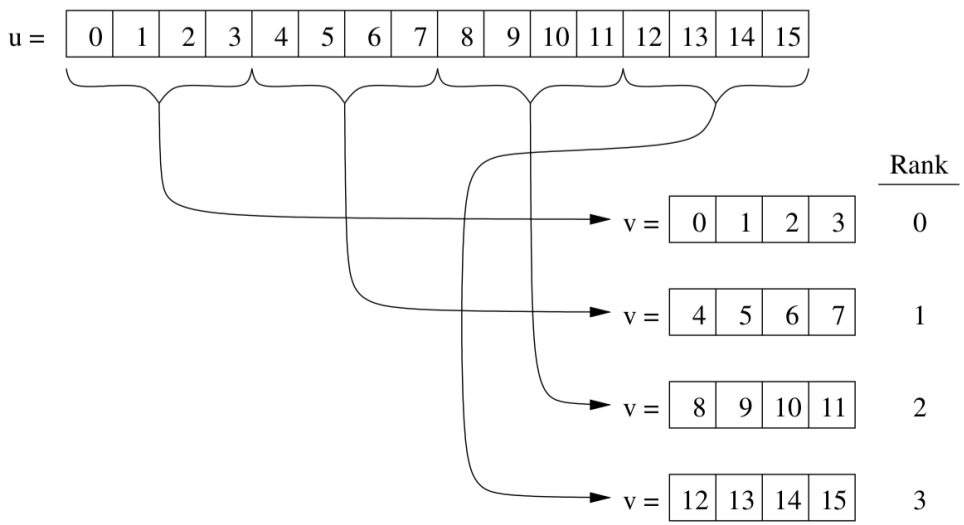
- MPI_MAX: Returns the maximum element.
- MPI_MIN: Returns the minimum element.
- MPI_SUM: Sums the elements.
- MPI_PROD: Multiplies all elements.
- MPI_LAND: Performs a logical and across the elements.
- MPI_LOR: Performs a logical or across the elements.
- MPI_BAND: Performs a bitwise and across the 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.

`MPI_Allreduce()`: Provides result of reduction too all processors.

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.

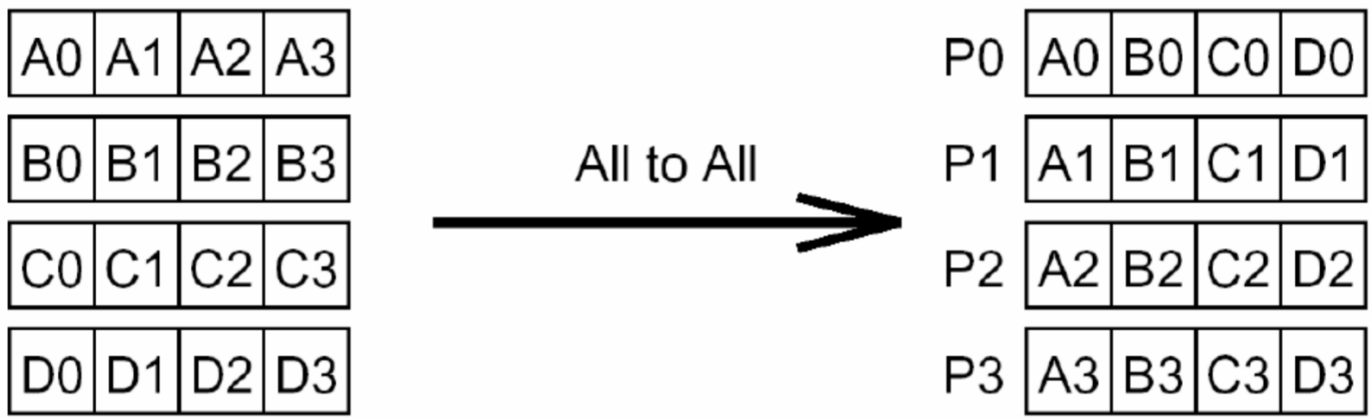
```
MPI_Gather(void* sendbuff, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype, int root, MPI_Comm communicator)
```

`MPI_Allgather()` gathers from all processors to all processors. Variable-sized variant is `MPI_Gatherv`.

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 or sorting. Variable-sized variant is called `MPI_Alltoallv`.

Outline

Organization issues

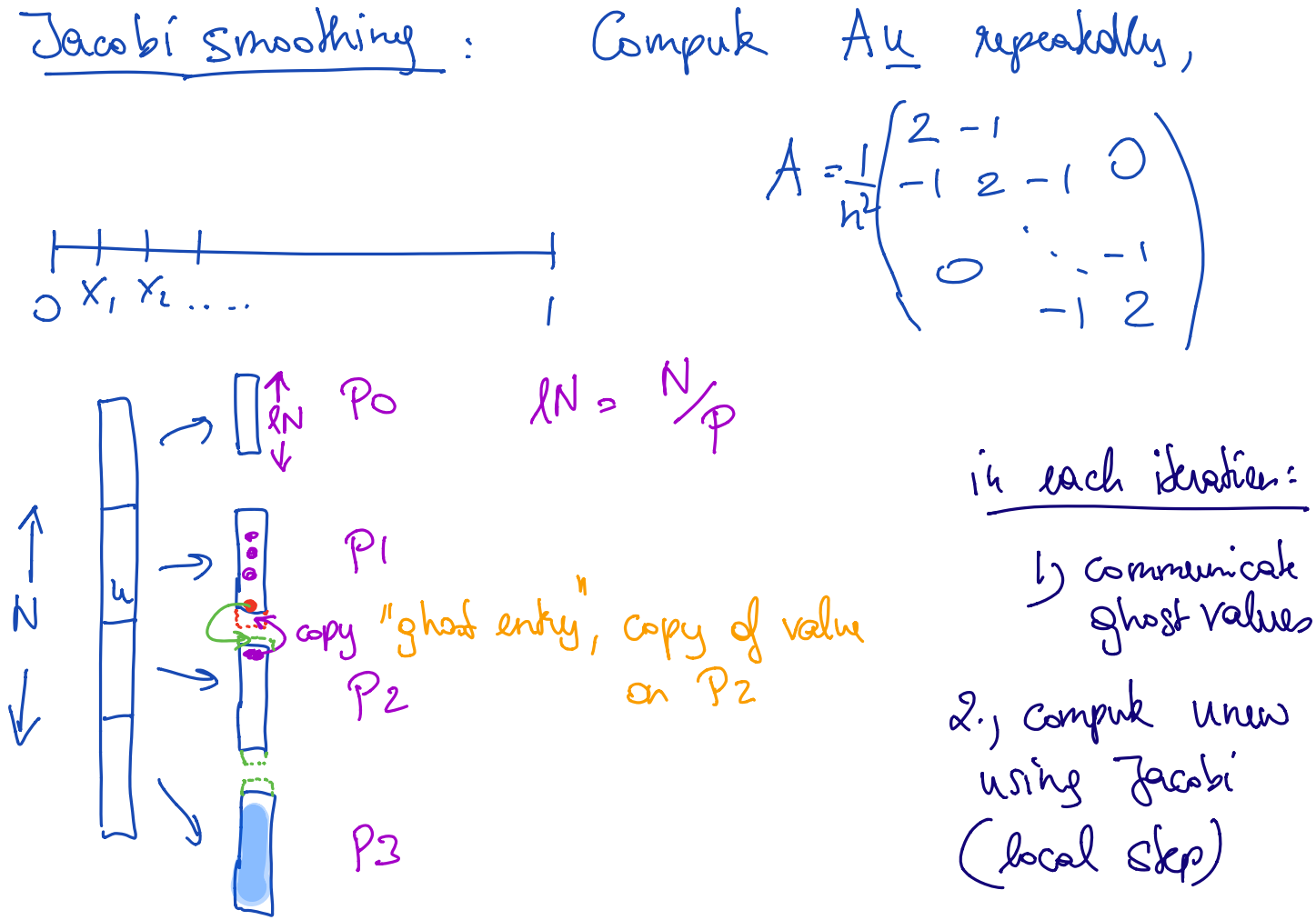
Summary of previous class

Distributed memory algorithms

Partitioning and Load Balancing

Parallel Jacobi in 1D

Blocking Send and Recv



https://github.com/NYU-HPC19/lecture12

Parallel Jacobi in 1D

Non-blocking Send and Recv

non-blocking version:

1.) Start communicating ghost values using iSend & iRecv

2.) Jacobi for u_new for interior points

3.) finish communication (check with MPI_Wait

4.) update 2 remaining entries (first & last in each vector)

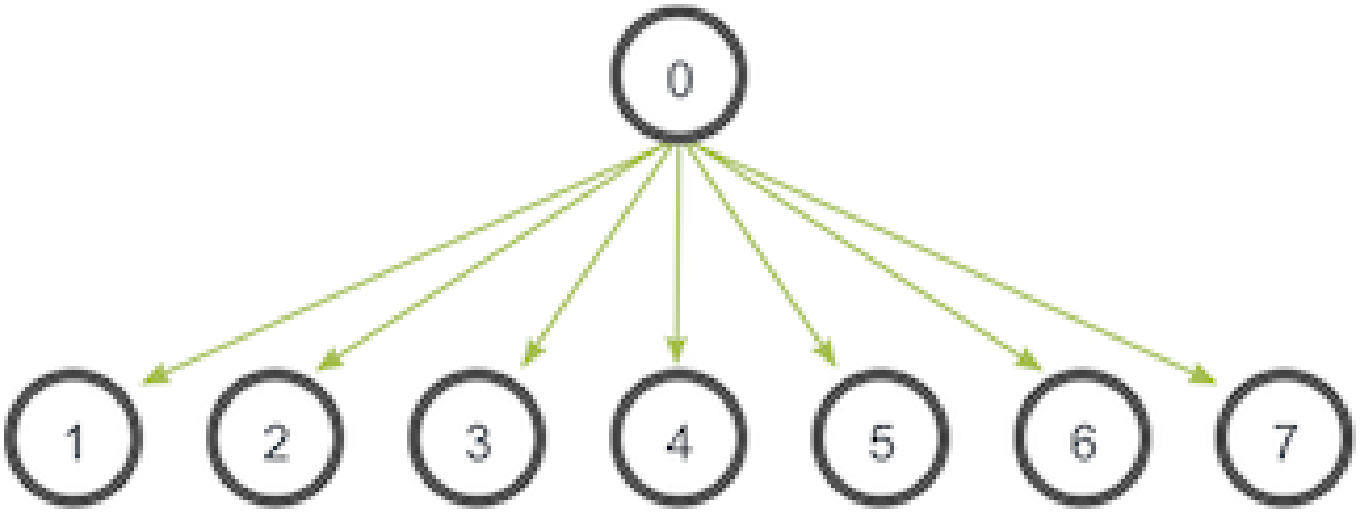
"interleaf communication with computation"

https://github.com/NYU-HPC19/lecture12

Collective Communication Algorithms

Book: Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar (Chapter 4)

Naive broadcast algorithm



Communication cost: $T(m, p) = t_s \times p + t_w \times m \times p$

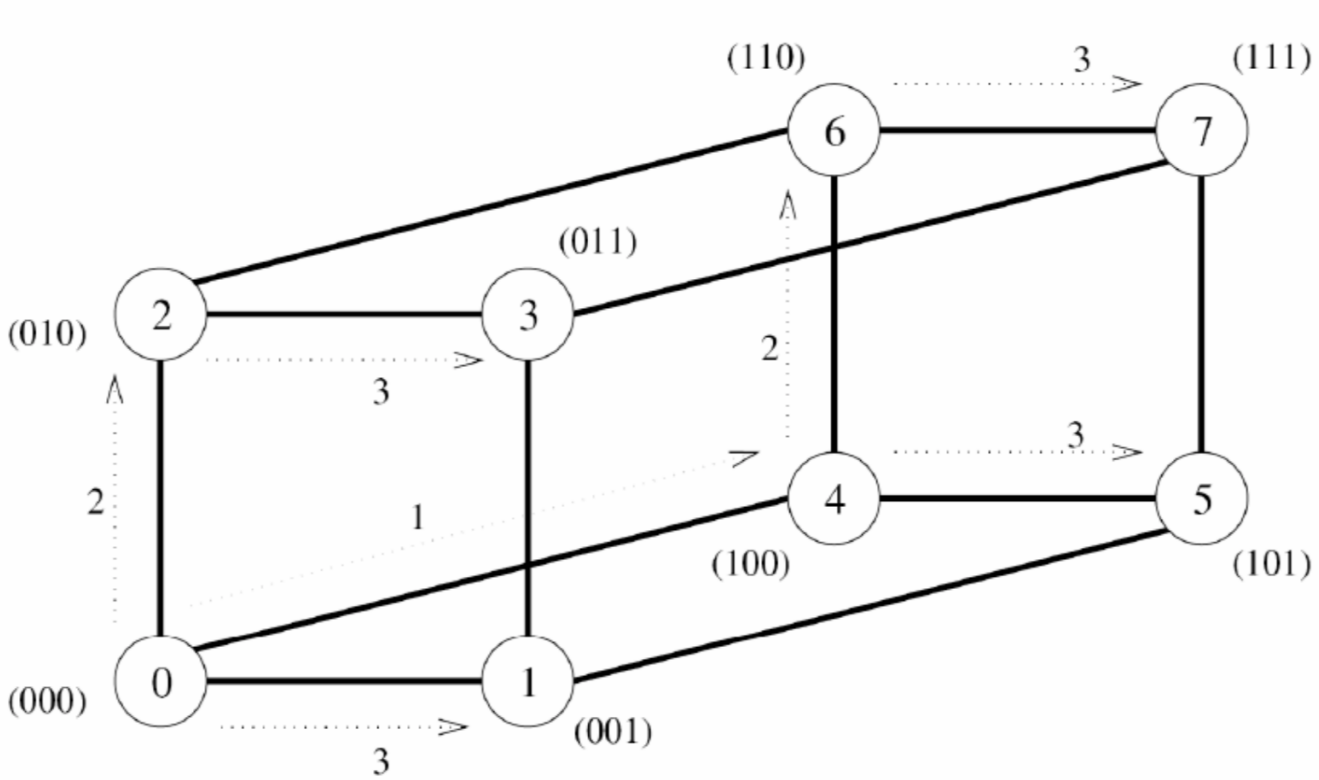
where,

$t_s =$ latency
$t_w = 1/$bandwidth
$p =$ number of processes
$m =$ message size

Hypercube Broadcast Algorithms

Tree based algorithm and assuming an uncongested network

- in each stage communicate along one dimension of the hypercube



Cost: $T(m,p) = t_s \times \log p + t_w \times m \log p$

Similar idea for MPI_Reduce, MPI_Allreduce, MPI_Allgather, MPI_Alltoall etc.

Distributed Sorting Algorithms

Book: Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar (Chapter 9)

Bucket sort

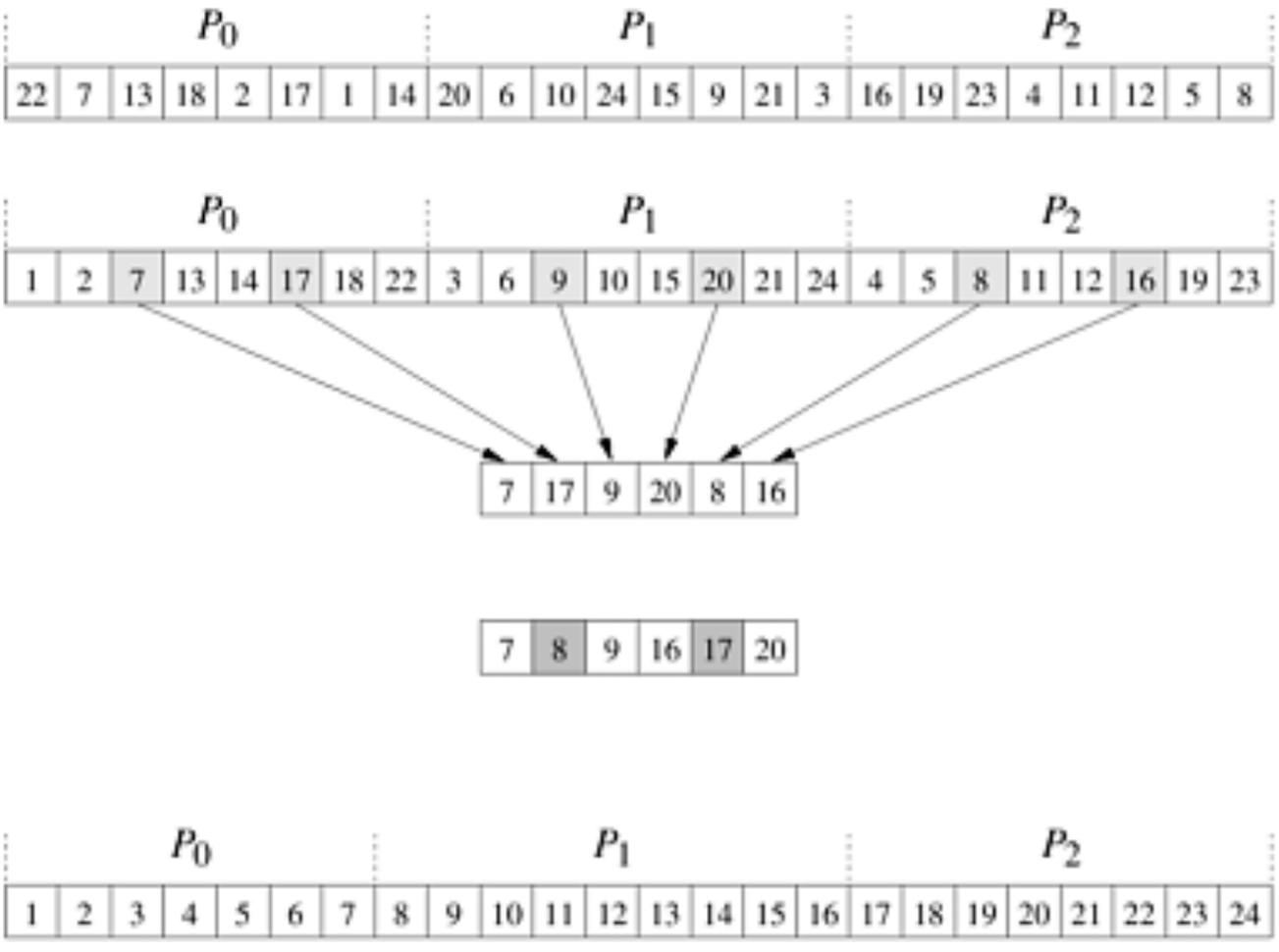
- partition input array evenly across p processes
  - assign n/p elements to each process
- Create p buckets
- Each process sorts its elements to buckets
- Alltoallv(), pth processor gets pth bucket
- local sort of elements in pth bucket
- **Works only for uniform distribution of elements.**

Distributed Sorting Algorithms

Sample Sort

- partition input array evenly across p processes
- sort locally
- select (p-1) splitters/processor (evenly)
  - guarantees no more 2*n/p elements / bucket
- gather(splitters) in P0
- sort splitters in P0 and create buckets
  - block partition using p binsearch on n/p sorted seq.
- broadcast buckets
- bucket sort

Distributed Sample Sort (Example)



Cost Sample Sort

- sort() locally: $\mathcal{O}(n/p \ \log n/p)$
- select p-1 splitters/proc : $\mathcal{O}(p)$
- gather(splitters) in P0: $\mathcal{O}(p^2)$
- sort splitters in P0 and create p-1 new splitters $\mathcal{O}(p^2 \ \log p)$
- broadcast(new splitters) $\mathcal{O}(p \log p)$
- parallel bucket sort
  using new splitters $\mathcal{O}(n/p \log n/p + p \log n/p + n/p)$

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(p^2 \log p\right)}^{\text{sort sample}} + \overbrace{\Theta\left(p \log \frac{n}{p}\right)}^{\text{block partition}} + \overbrace{\Theta(n/p) + O(p \log p).}^{\text{communication}}$$

Outline

Organization issues

Summary of previous class

Distributed memory algorithms

Partitioning and Load Balancing

Partitioning and Load Balancing

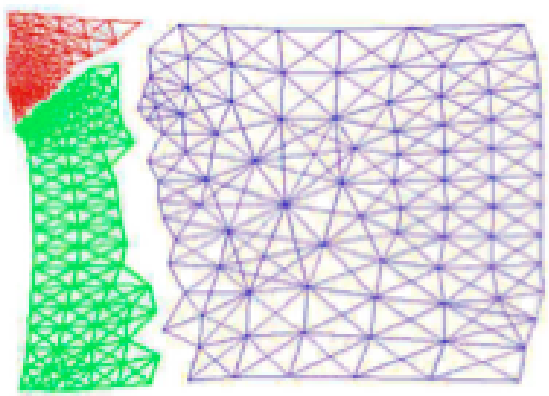
Thanks to Marsha Berger for letting me use many of her slides. Thanks to the Schloegel, Karypis and Kumar survey paper and the Zoltan website for many of these slides and pictures.

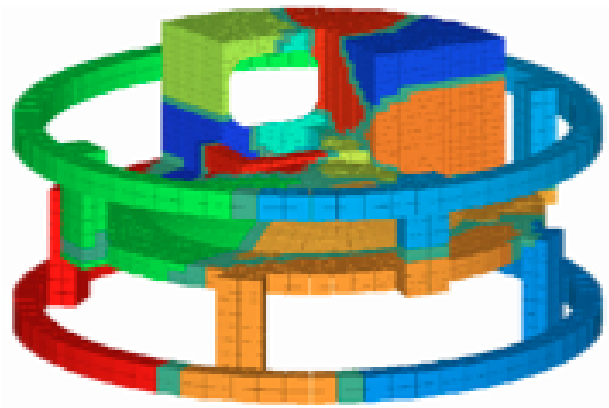
Partitioning

- 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.

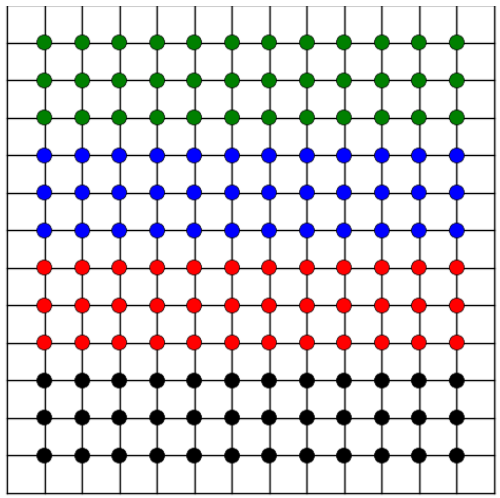


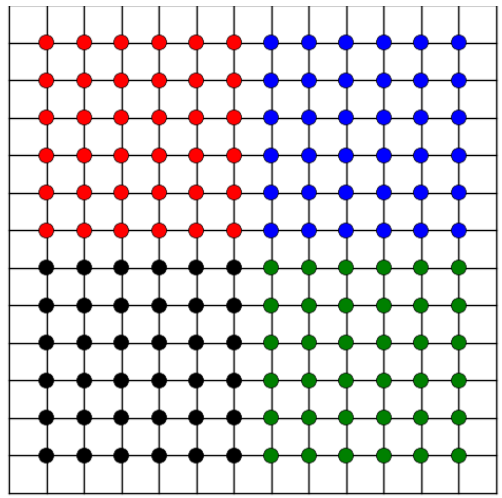


Partitioning

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.

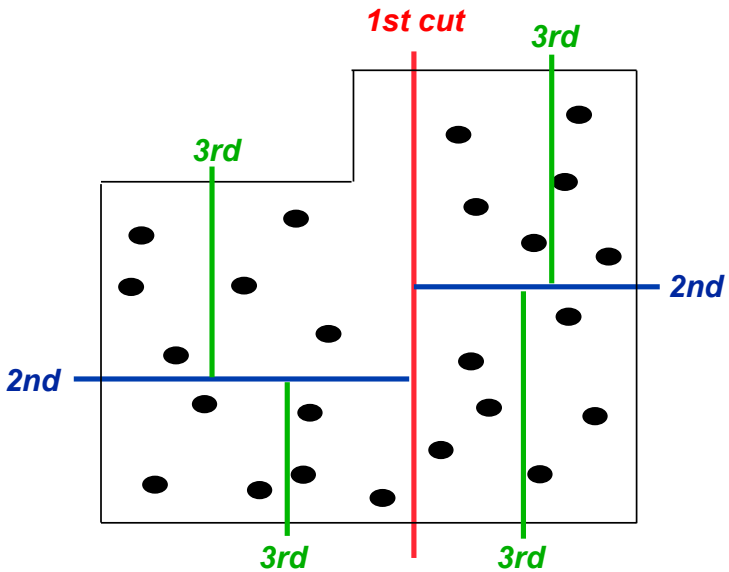
- Dynamic - information not known until runtime, work changes during computation (e.g. adaptive methods), or locality of objects change (e.g. particles move)

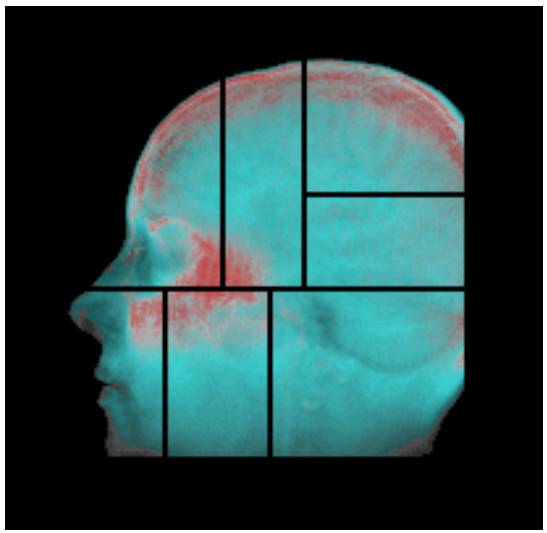
  use on-line algorithms to make decisions mid-execution; must run side-by-side with application, should be parallel, fast, scalable. Incremental algorithm preferred (small changes in input result in small changes in partitions)

will look at some geometric methods, graph-based methods, spectral methods, multilevel methods, diffusion-based balancing,...

Recursive Coordinate Bisection

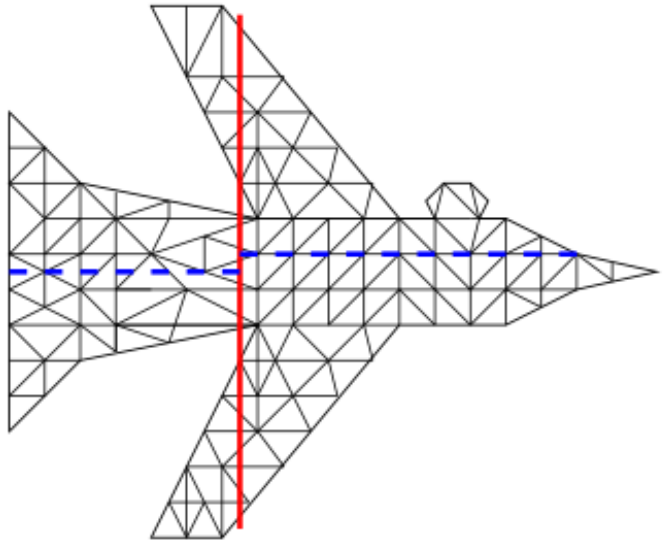
Divide work into two equal parts using cutting plane orthogonal to coordinate axis For good aspect ratios cut in longest dimension.

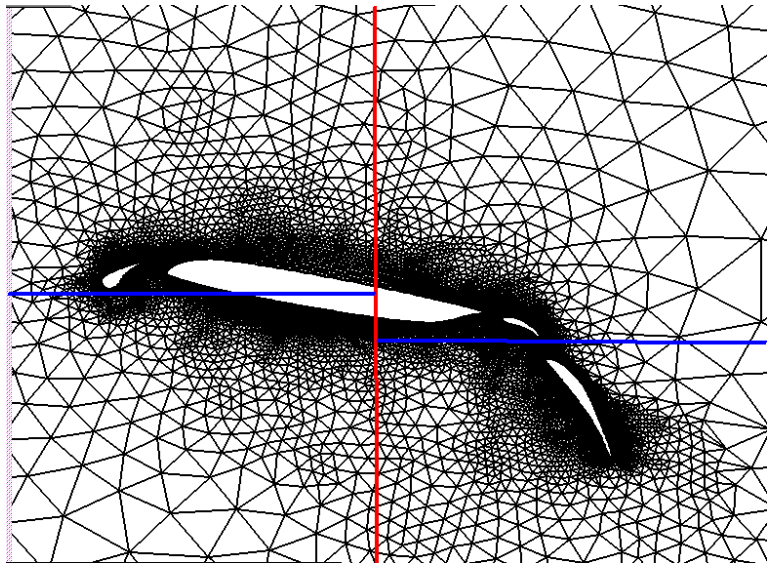




Can generalize to k-way partitions. Finding *optimal* partitions is NP hard. (There are optimality results for a class of graphs as a graph partitioning problem.)

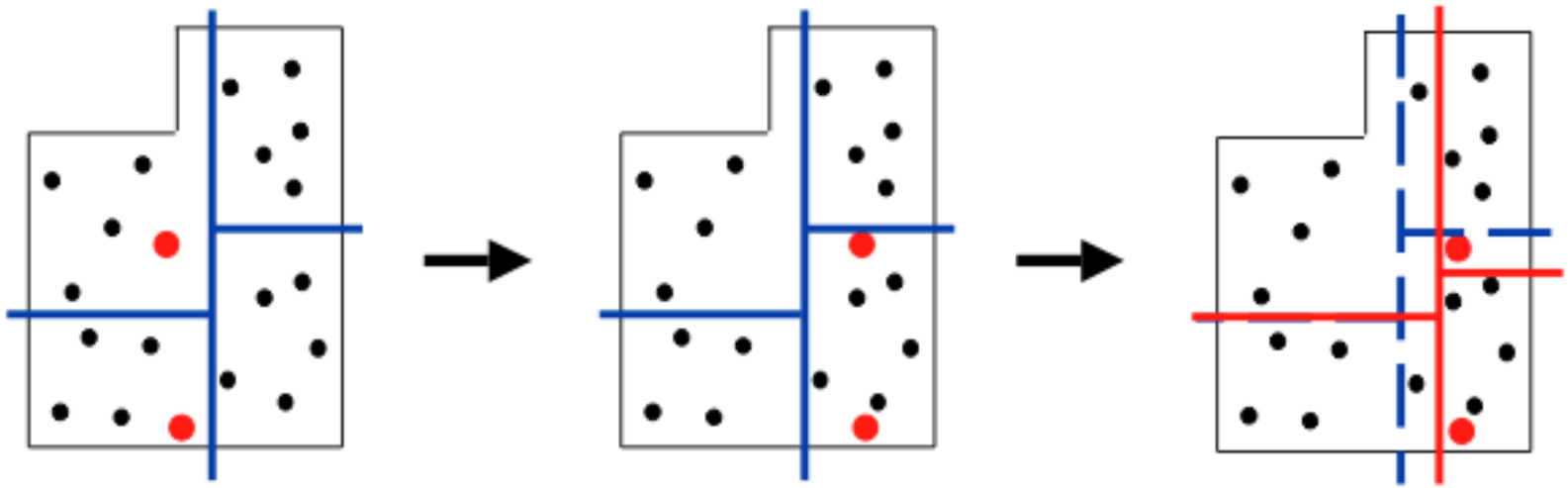
Recursive Coordinate Bisection





+ 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

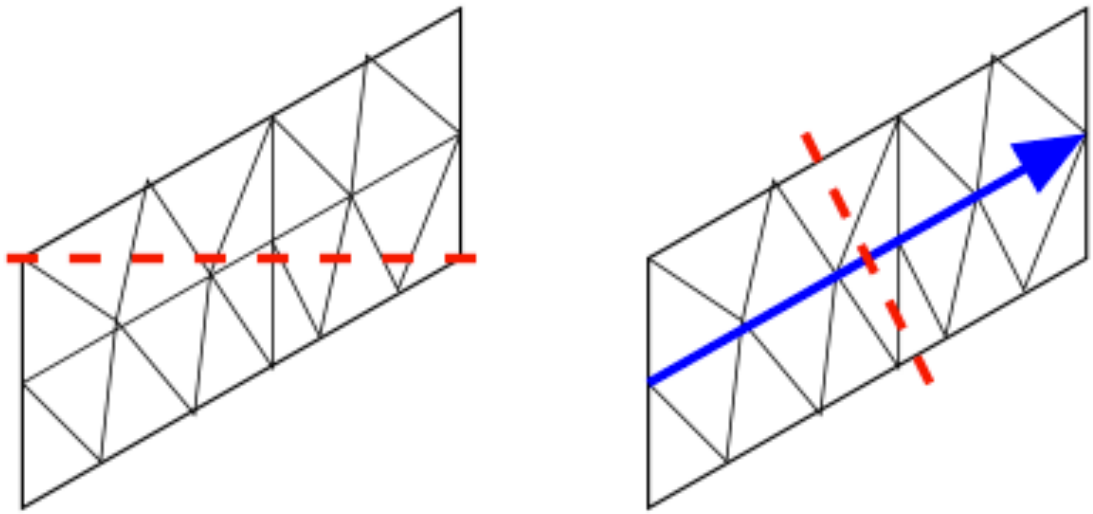
Recursive Coordinate Bisection



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

Recursive Inertial Bisection

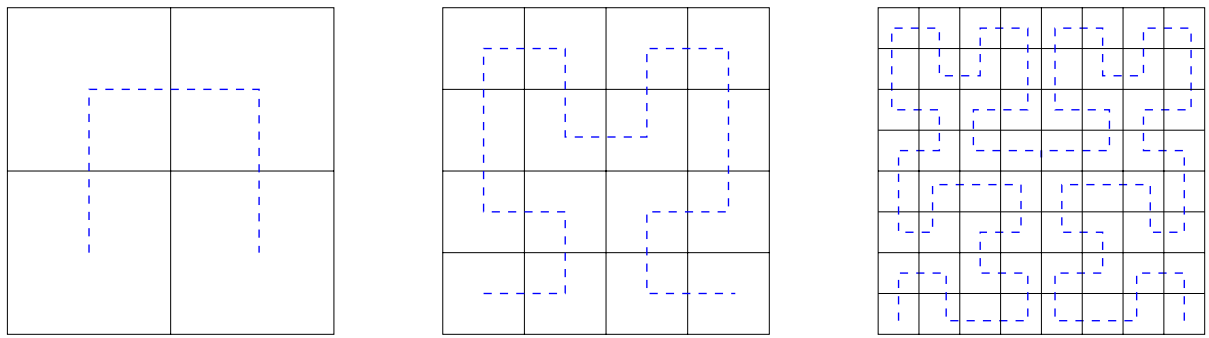
For domains not oriented along coordinate axes can do better if account for the angle of orientation of the mesh.



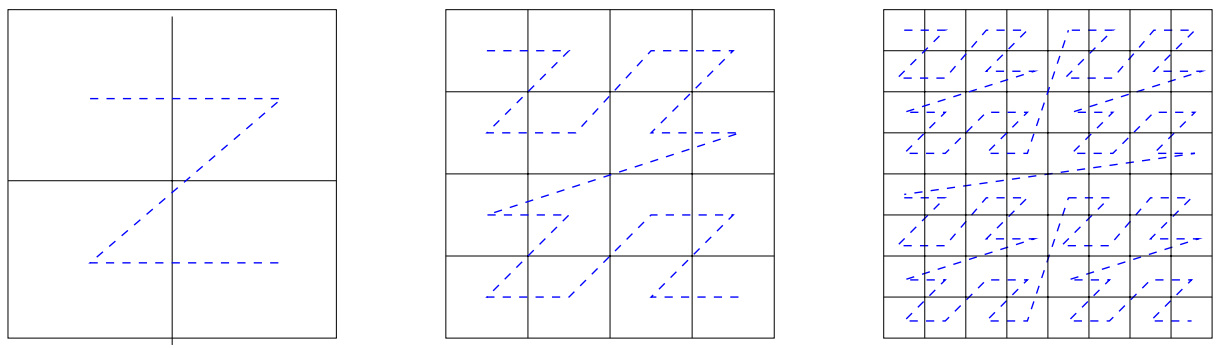
Use bisection line orthogonal to principal inertial axis (treat mesh elements as point masses). Project centers-of-mass onto this axis; bisect this ordered list. Typically gives smaller subdomain boundary.

Space-filling Curves

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



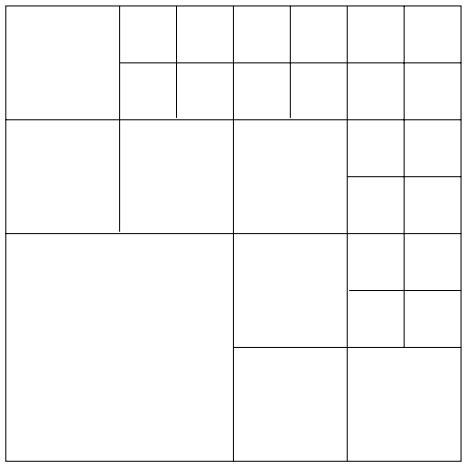
Peano-Hilbert ordering

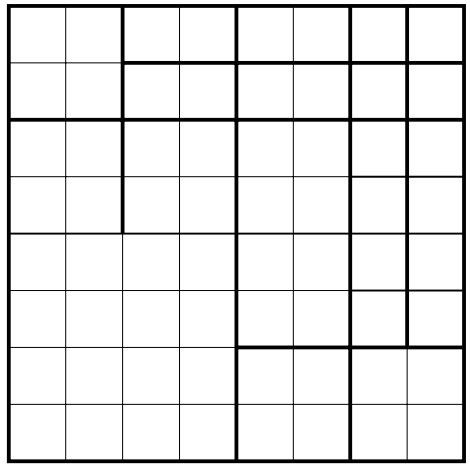


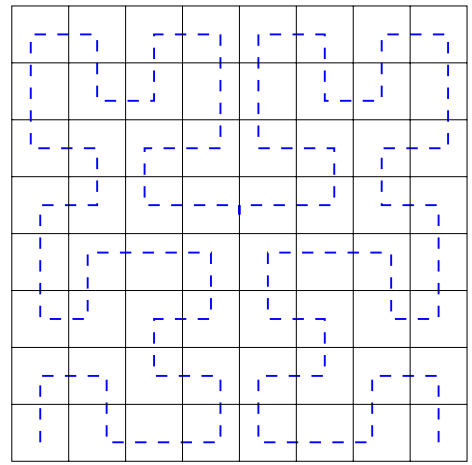
Morton ordering

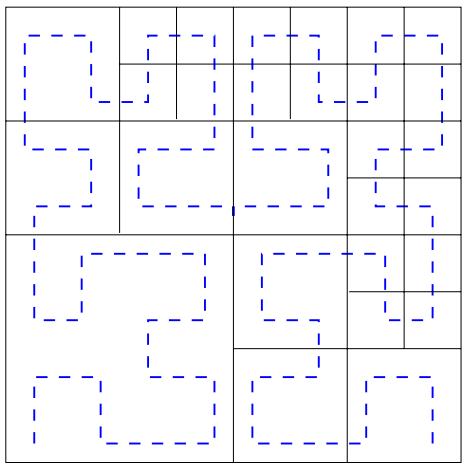
Space-filling Curves

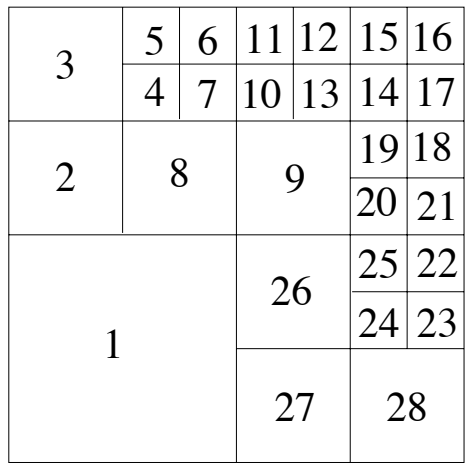
Easily extends to adaptively refined meshes



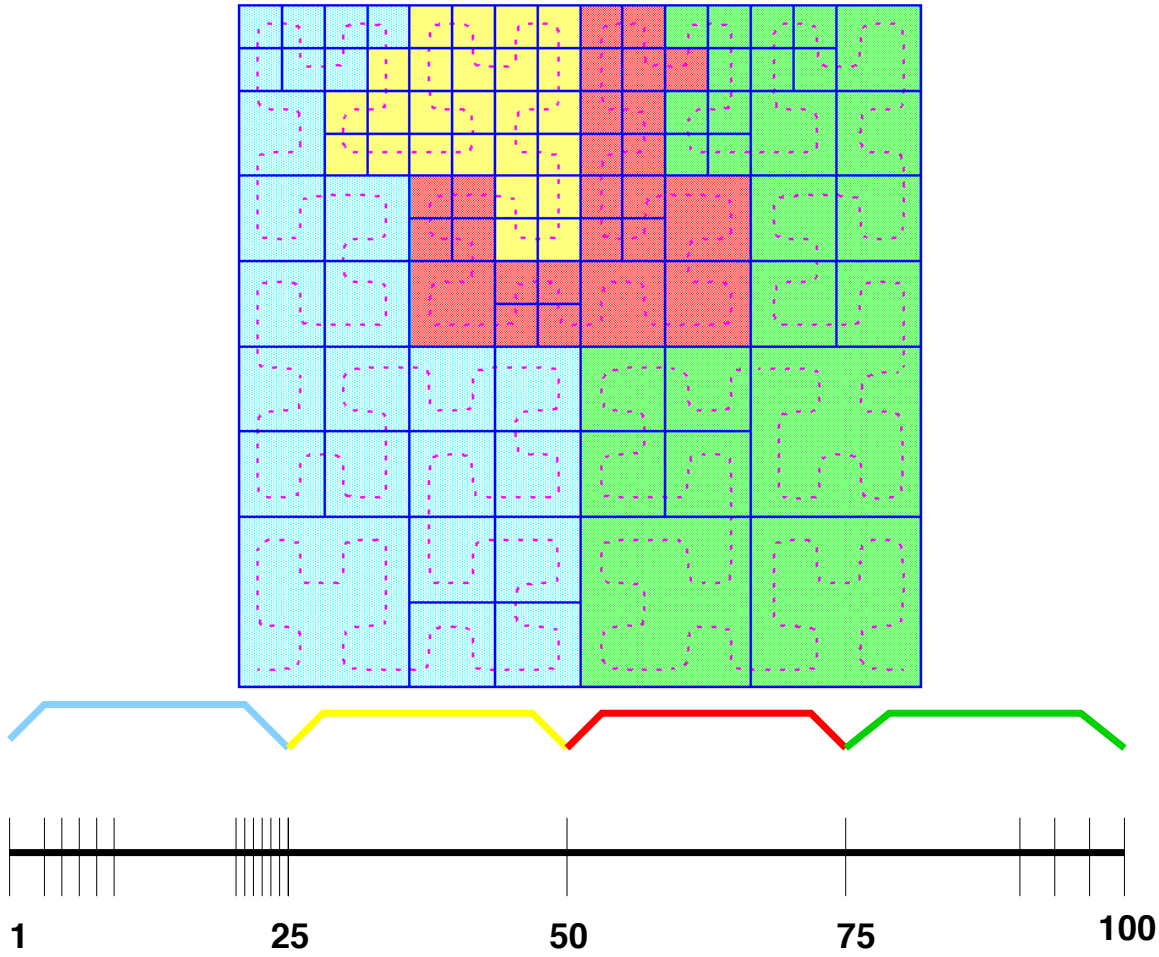






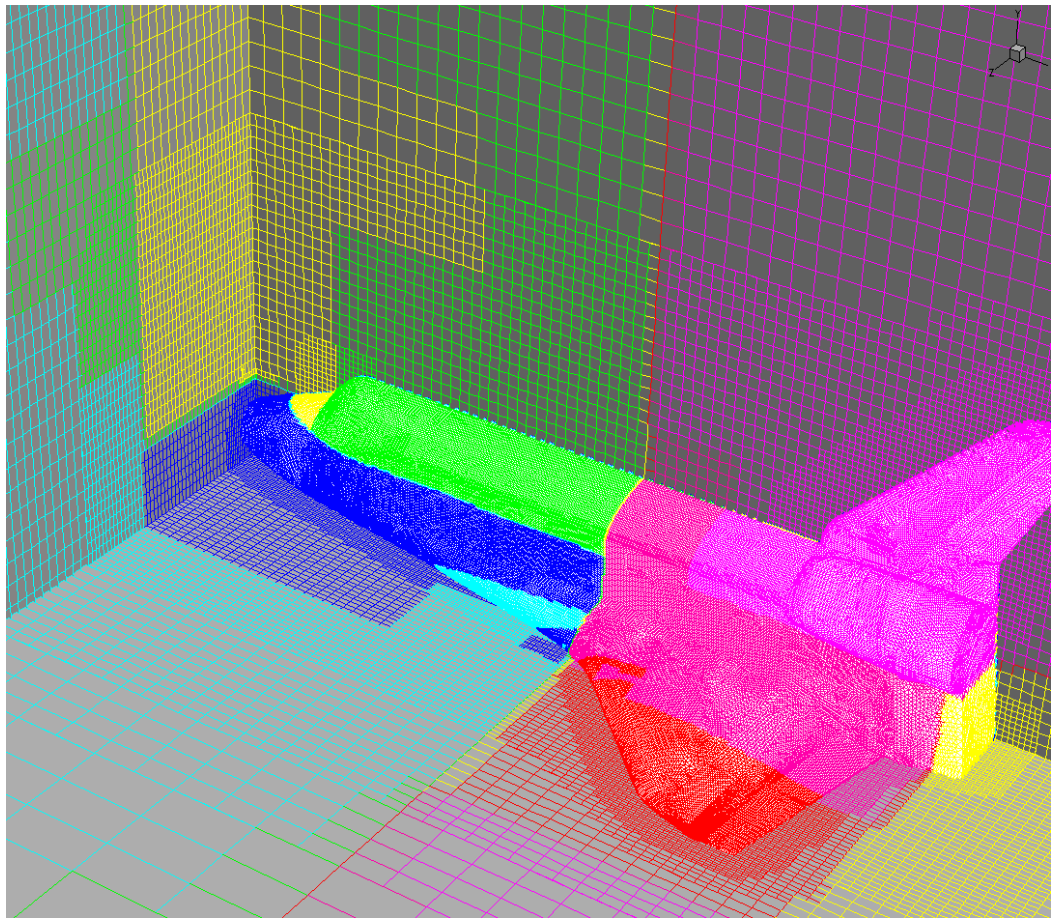


Space-filling Curves



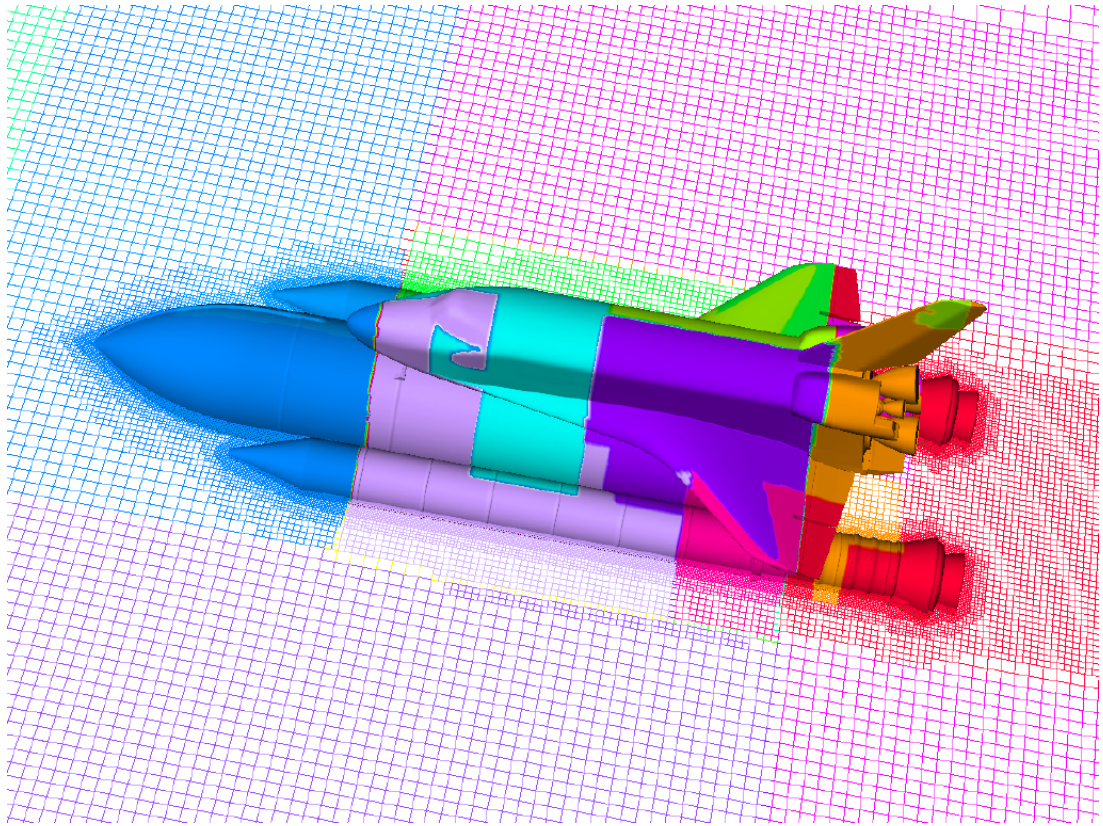
Partition work into equal chunks.

Space-filling Curves



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

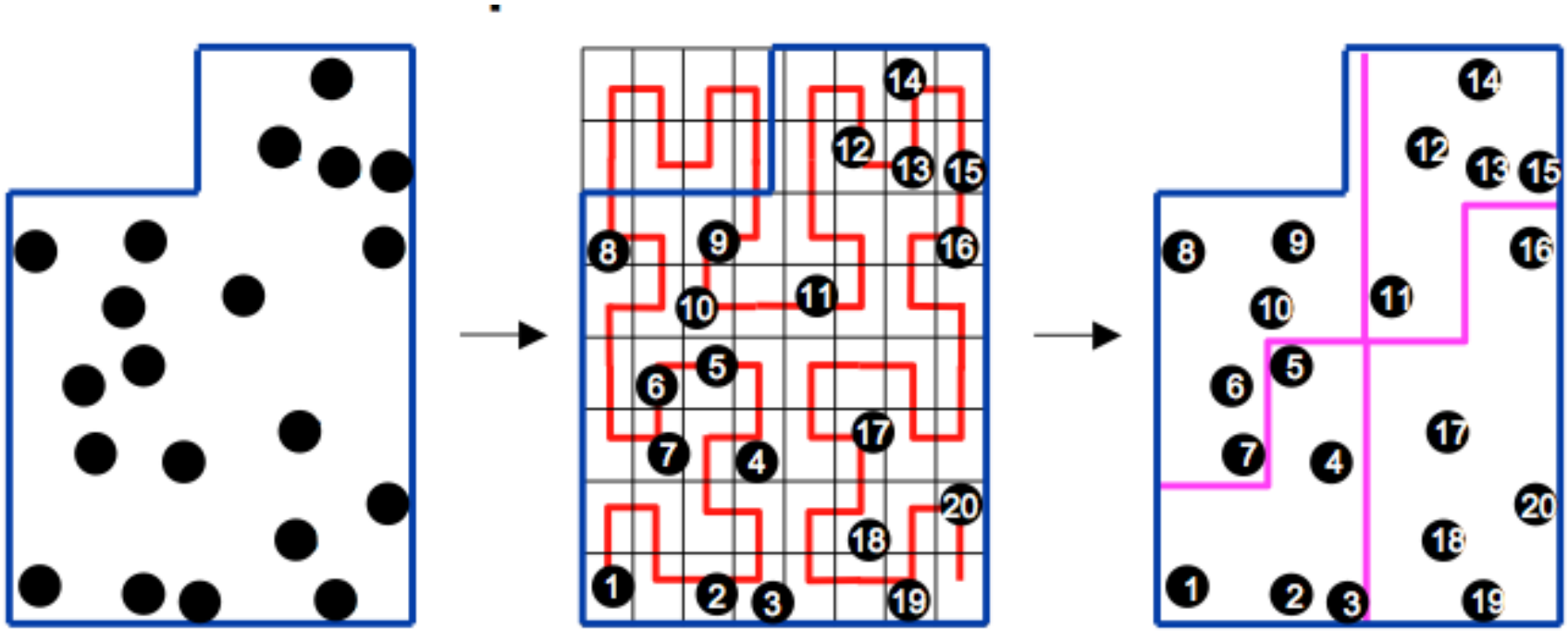
Space-filling Curves



- 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