THE UNIVERSITY OF ARIZONA®

# Summer 2009 REU: Introduction to Some Advanced Topics in Computational Mathematics

Moysey Brio & Paul Dostert

July 4, 2009

# Sparse Matrices

In many areas of applied mathematics and modeling, one comes across sparse matrix problems, where we need to solve $A\mathbf{x} = \mathbf{b}$ with $A$ containing primarily zeros.

Storage of $n \times n$ linear system with $m$ nonzero entries:

- *Full Matrix Storage*: $n^2$ elements. For $n = 10,000$ a double precision matrix takes approx $763$ Mb.
- *Matlab-like* $(i, j, A_{ij})$ *Storage*: $3m$ elements. For $n = 10,000$ and $m = 50,000$ a double precision matrix takes approximately $\frac{1}{2}$ Mb.

An example problem would be solving an elliptic PDE on a 2D mesh. Using a 2D mesh and simple finite differences, one would want an approximation to the solution of a PDE at each point in the mesh. For a $1000 \times 1000$ mesh, which is quite small in practical problems, we have a resulting $1,000,000 \times 1,000,000$ matrix. Having $5,000,000$ nonzeros would result in approximately $76$ Mb for $(i, j, A_{ij})$ storage.

# CSR & CSC

For $(i, j, A_{ij})$, generally all the $i = 1$ values come first, then $i = 2$, and so in. Instead of saving each $i$ value, we save how many entries come before we transition from row $i$ to row $i + 1$. This is the idea behind compressed sparse row (column) format.

Consider storing the following matrix:

$$\begin{pmatrix} 6 & 1 & 0 & 1 & 0 & 0 & 2 & 0 \\ 1 & 6 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 6 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 6 & 0 & 3 & 0 & 0 \\ 5 & 0 & 0 & 1 & 6 & 0 & 0 & 1 \\ 2 & 0 & 0 & 3 & 0 & 6 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \end{pmatrix}.$$

We keep

$$\begin{aligned} a &= [6\,1\,1\,2\,1\,6\,1\,1\,1\,1\,6\,1\,1\,6\,3\,5\,1\,6\,1\,2\,3\,6\,1\,6\,1\,6] \\ J &= [1\,2\,4\,7\,1\,2\,4\,6\,1\,2\,3\,4\,7\,4\,6\,1\,4\,5\,8\,1\,4\,6\,1\,7\,7\,8] \\ I &= [1\,5\,9\,14\,16\,20\,23\,25\,27] \end{aligned}$$

which is only $2m + n + 1$ entries.

# CSR & CSC

Storage of $n \times n$ linear system with $m$ nonzero entries:

- *Full Matrix Storage*: $n^2$ elements. For $n = 1,000,000$ a double precision matrix takes approx $7$ Tb.
- *Matlab-like* $(i, j, A_{ij})$ *Storage*: $3m$ elements. For $n = 1,000,000$ and $m = 5,000,000$ a double precision matrix takes approximately $76$ Mb.
- *CSR & CSC Storage*: $2m + n + 1$ elements. For $n = 1,000,000$ and $m = 5,000,000$ a double precision matrix takes approximately $61$ Mb.

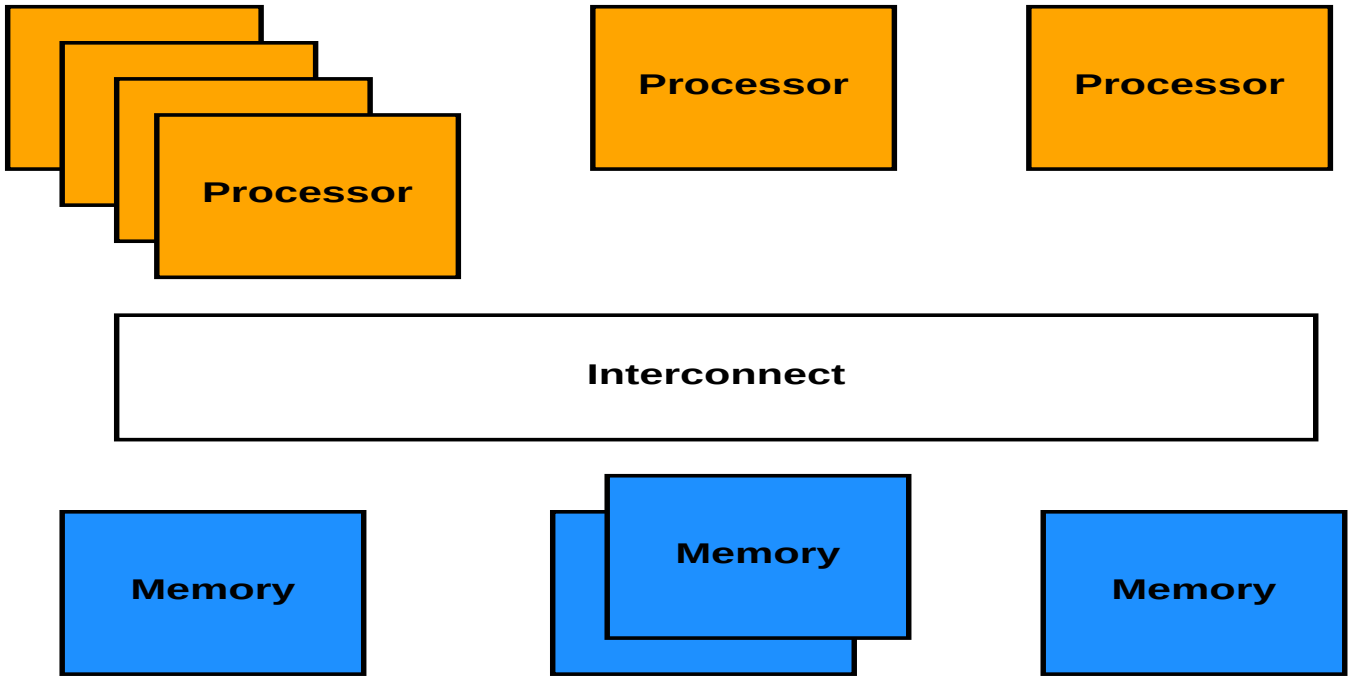
Issues:

- CSR does not save a huge amount of space, but in applications $> 2D$, matrix sizes are well into the trillions. This could be a savings in a few terabytes.
- CSR does not take advantage of symmetry, or repeated patterns.
- Matrix-vector, matrix-matrix multiplication routines should be rewritten to use CSR.
- In a practice, a full matrix is stored only if completely necessary. Often, we only need to know how matrices act, rather than every element of a matrix. For example, to use a Conjugate Gradient routine one only needs to know how to compute $A\mathbf{x}$ for any $\mathbf{x}$.

# Parallel Computing Architectures

Some problems simply cannot be done on a single computer or single processor. For these problems, we must use some special techniques. First, let us discuss different type of parallel computing environments.

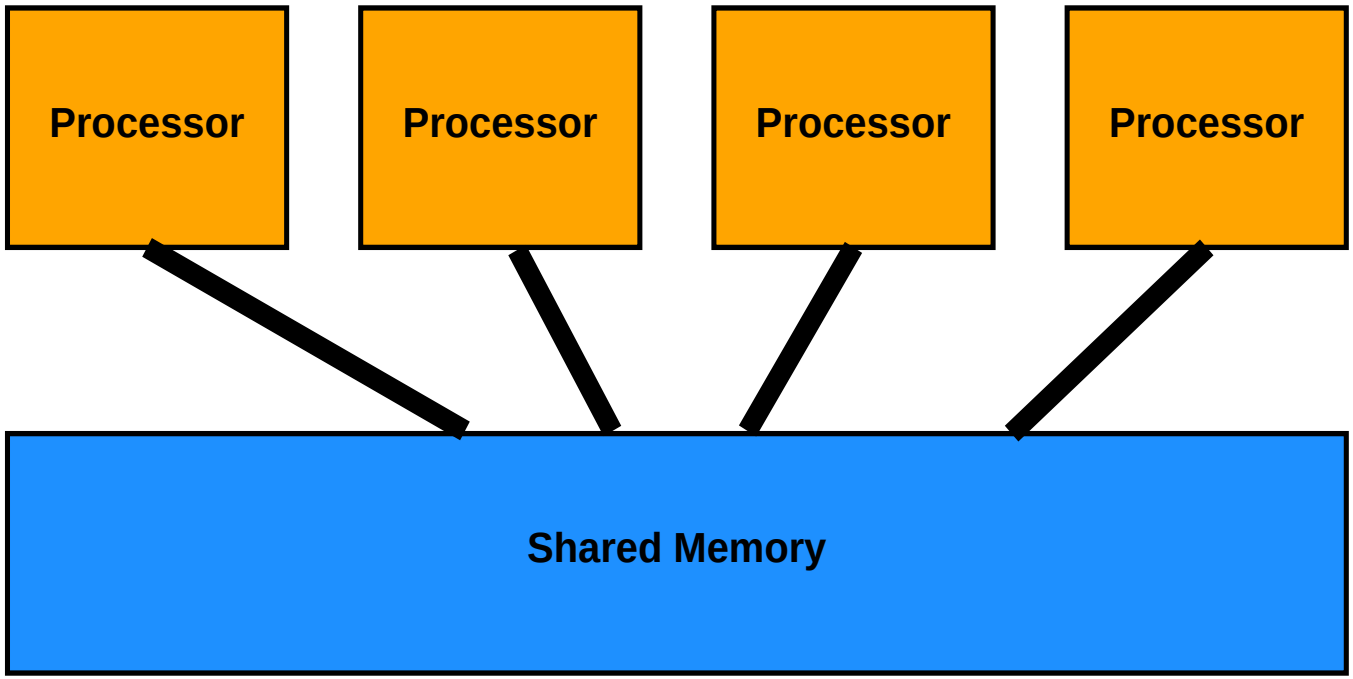
In a general environment, we have a set of processors, a set of memory banks, and some way to connect between them. The way this is set up defines different types of parallel machines



# Shared Memory

Examples of shared memory computers would be your multicore desktop, some SGI Power Challenge servers, and some Cray machines. Properties often include:

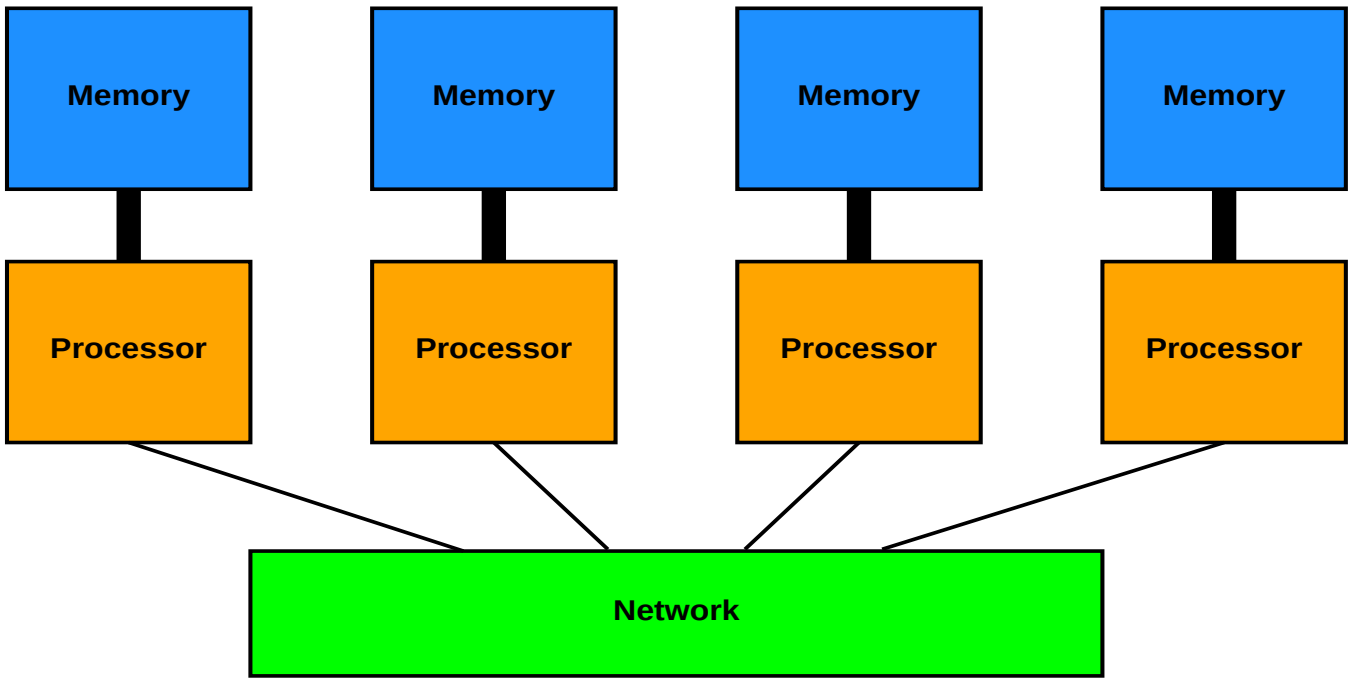
- A single memory bank, which each processor has relatively fast access to.
- Few processors or cores. Usually, at most, $1000$ or so processors.
- Very expensive. Fast interconnects between processor are extremely expensive.



# Distributed Memory

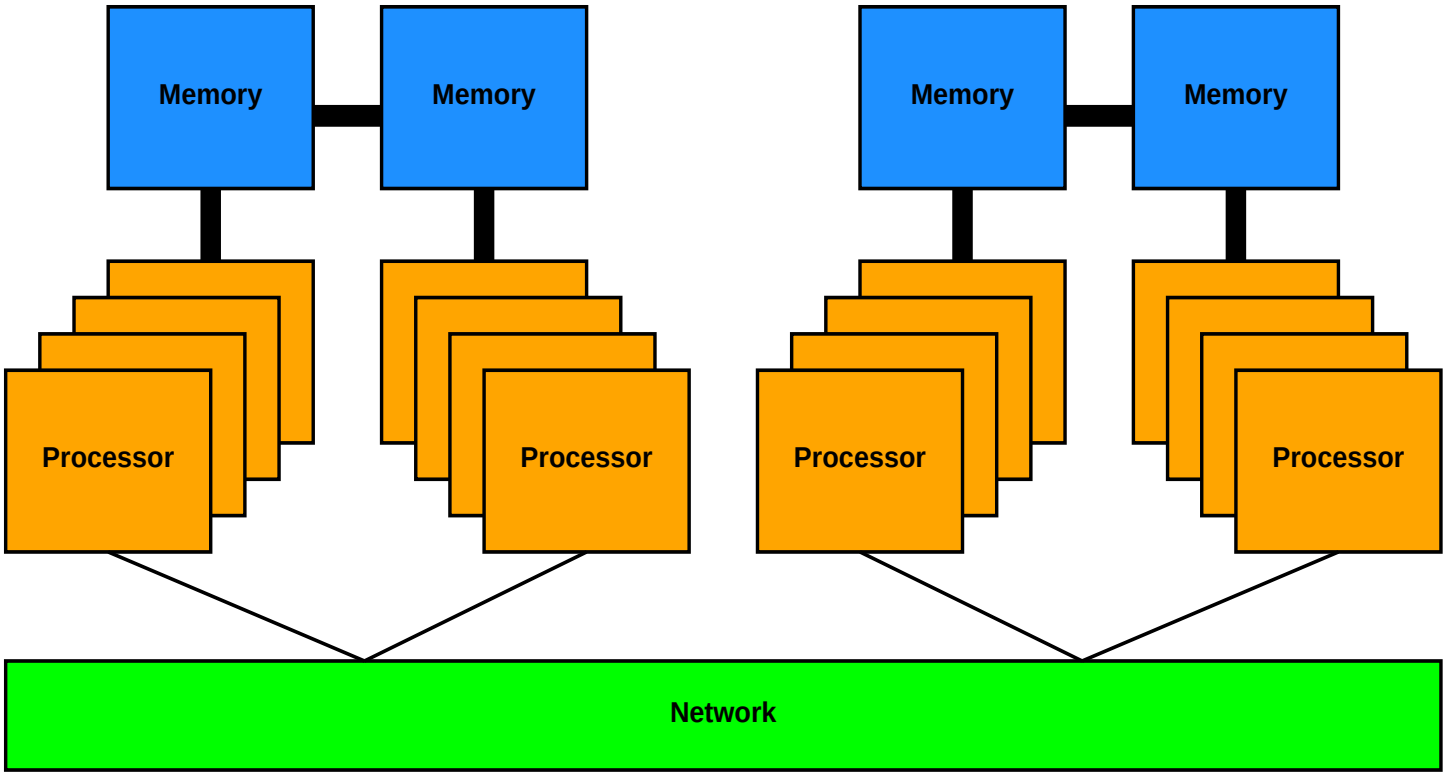
Examples of distributed memory computers would be a traditional Beowulf cluster. Properties often include:

- Each processor has it's own local memory bank.
- Relatively slow interconnect between processors.
- Inexpensive. In it's simplest form, this can simply be PCs linked together over a network.
- Can be many processors (IBM *Roadrunner* has $19,440$ processors, with $129,600$ cores).



# Realistic and Modern Designs

Nearly all powerful clusters are a combination of shared and distributed memory. Essentially, think of this as series of shared memory machines linked to each other in a distributed network.



# Parallelization of Linear Systems

Let us consider a matrix problem $A\mathbf{x} = \mathbf{b}$ where $A$ is a tridiagonal matrix, with $-2$ on the diagonal and $1$ on the upper and lower diagonals. Suppose we are given a vector $\mathbf{y}$ and wish to compute $A\mathbf{y} = \mathbf{z}$. (Note: In a practical application, we'd never store this matrix, since it's matrix-vector result can be easily computed without keeping each entry of $A$).

We must compute:

$$\begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} -2y_1 + y_2 \\ y_1 - 2y_2 + y_3 \\ \vdots \\ y_{n-2} - 2y_{n-1} + y_n \\ y_{n-1} - 2y_n \end{pmatrix}$$

Suppose that we would like to compute the result using two processors, by letting the first processor compute the first half of $\mathbf{z}$ and the second processor the second half.

# Linear Systems on Shared Memory

On a shared memory machine, each processor has direct access to each part of $A$ and $\mathbf{y}$. We simply let each processor do the required work. We denote the work done on the first processor in red, and the second in green.

$$\begin{pmatrix} -2 & 1 & 0 & \dots & 0 \\ 1 & -2 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 1 & -2 & 1 \\ 0 & \dots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} -2y_1 + y_2 \\ y_1 - 2y_2 + y_3 \\ \vdots \\ y_{n-2} - 2y_{n-1} + y_n \\ y_{n-1} - 2y_n \end{pmatrix}$$

We can easily extend this idea to many more processors. Note that we can split the matrix any way desired, as long as we balance work to reach processor evenly.

# Linear Systems on Distributed Memory

On a distributed memory machine, different parts of the matrix and vectors are stored locally on different memory/processor locations. For example, we could multiply using the coloring below, where entries in cyan are needed by both processors (we ignore zeros):

$$\begin{pmatrix} -2 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 & 0 & 0 \\ 0 & \dots & 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & \dots & 0 & 1 & -2 & -1 & 0 & 0 \\ 0 & \dots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & \dots & 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n/2-1} \\ y_{n/2} \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} =$$

$$\begin{pmatrix} -2y_1 + y_2 \\ \vdots \\ y_{n/2-2} - 2y_{n/2-1} + y_{n/2} \\ y_{n/2-1} - 2y_{n/2} + y_{n/2+1} \\ \vdots \\ y_{n-1} - 2y_n \end{pmatrix}$$

# Linear Systems on Distributed Memory

Issues in using distributed memory systems:

- BOTH CPUs need $y_{n/2}$ and $y_{n/2-1}$.
- Usually, CPU $1$ would store $y_1$ to $y_{n/2-1}$ while CPU $2$ stores the rest.
- This means for CPU $1$ to use $y_{n/2}$, the value needs to be PASSED from CPU $2$ to CPU $1$.
- Intead of splitting the matrix as above, what happens if each odd row is stored on CPU $1$ and each even row on $2$?
- It is harder to write good code for distributed memory machines, but the same code would run very well on shared memory as well.

Application programming interfaces (APIs) for parallel machines:

- For distributed memory machines, Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the most used.
- For shared memory machines, there are many choices:
  - OpenMP: Often works with MPI in practical machines.
  - Intel Threading Building Blocks (TBB): Relatively new.
  - POSIX Threads (Pthreads): usually C only.

# Parallelization in Matlab

Many Matlab functions are already built to use multicore processors :

- *» maxNumComThreads(n)* sets the number of cores to use.
- Enabling parallel computations usually comes through function options:

*options = optimset('UseParallel','always');*

There is also a *Parallel Computing Toolbox* allowing you to parallelize your own codes:

- Use *matlabpool open n* to enable $n$ machines/cores (if there are $n$ available).
- Change *for* to a *parfor* to indicate a loop can be done in any order.
- Use *matlabpool close* to indicate parallel code is done.
- Arrays can be split for distributed clusters using the *codistributed* command:

  *»A = rand(80, 1000);  D = codistributed(A, 'convert');*

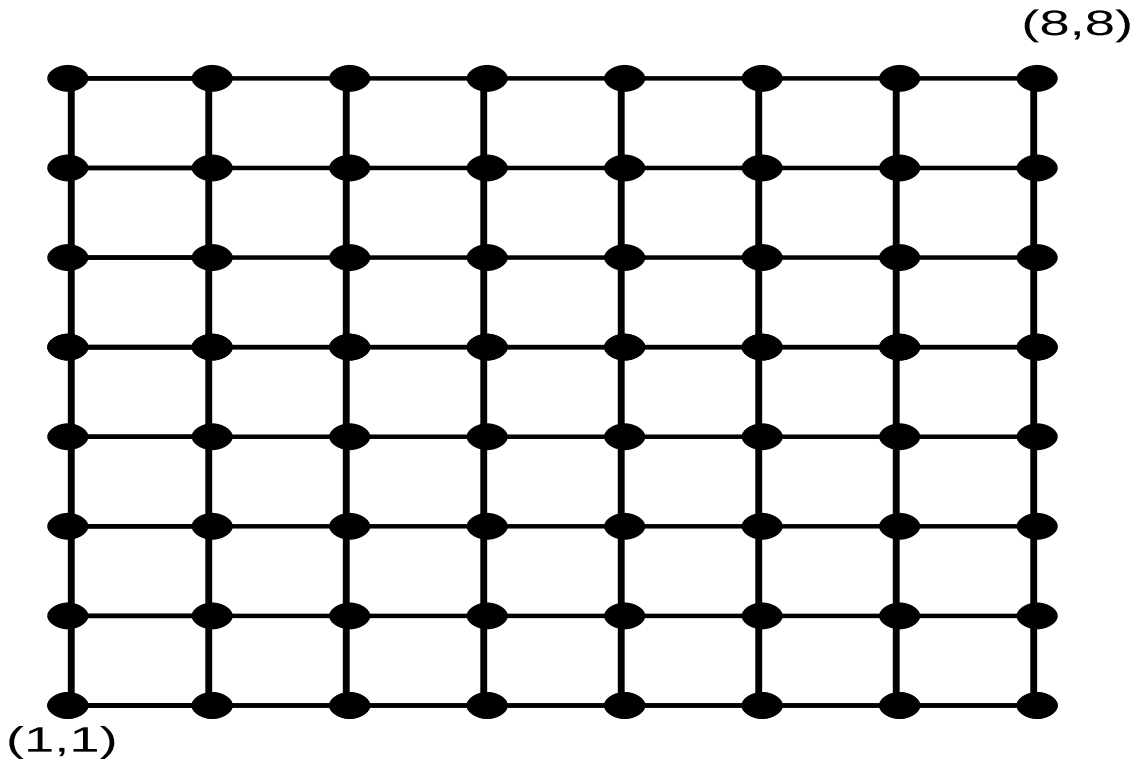
  This distributes the array to each process in the Matlab pool , as $D$. This requires everyone to already have $A$ stored.
- The above does not save on memory. To do that, we use something like:

  *»L = magic(5) + labindex;  A = codistributed(L, codistributor());*

# Mesh Partitioning

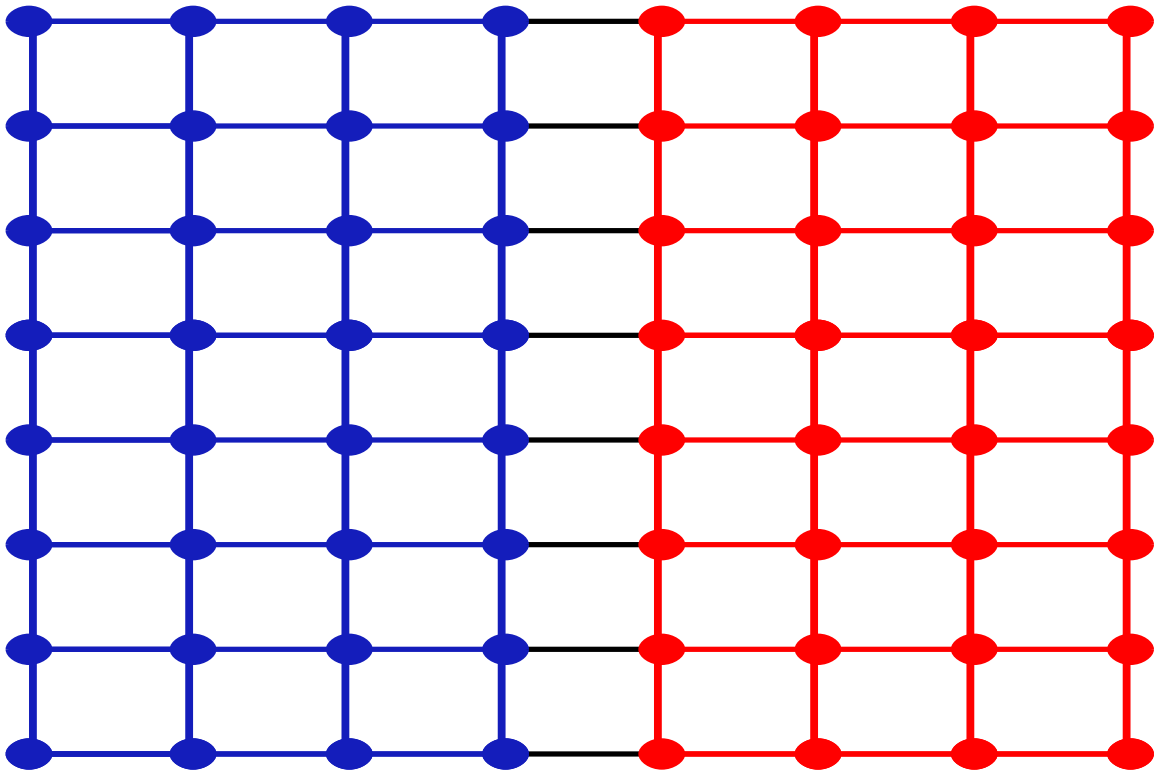
Often a nonzero entry $A_{ij}$ implies a geometric relationship between $i$ and $j$. This can be used to distribute loads evenly to CPUs. This process is often called *mesh partitioning*.

Consider an $8 \times 8$ grid where each point is an entry in $\mathbf{x}$. This results in a $64 \times 64$ matrix, $A$. Assume that row $i$ contains an element in the $j^{th}$ column iff $i$ and $j$ are connected in the mesh:



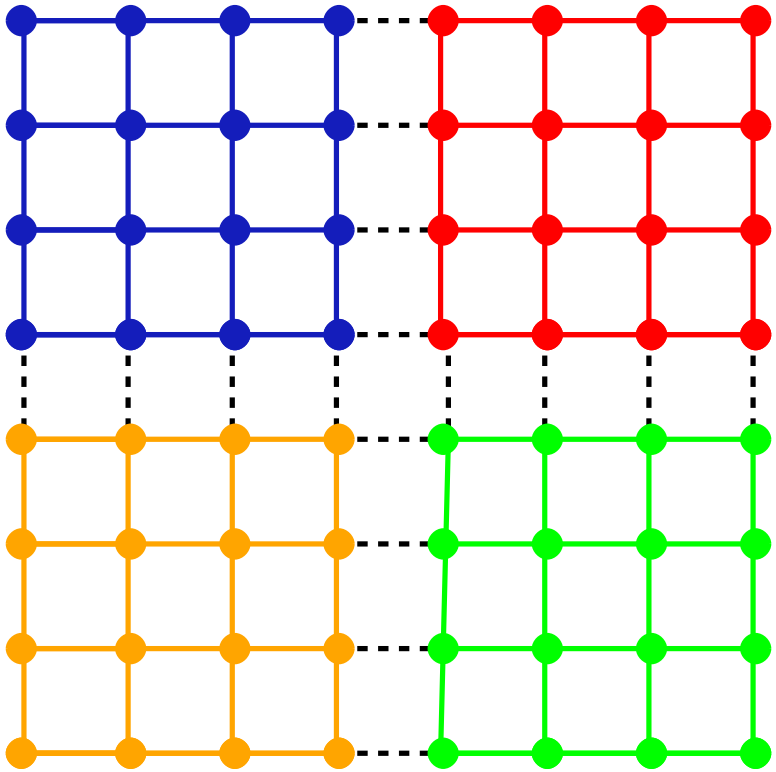
# Mesh Partitioning - 2 Processors

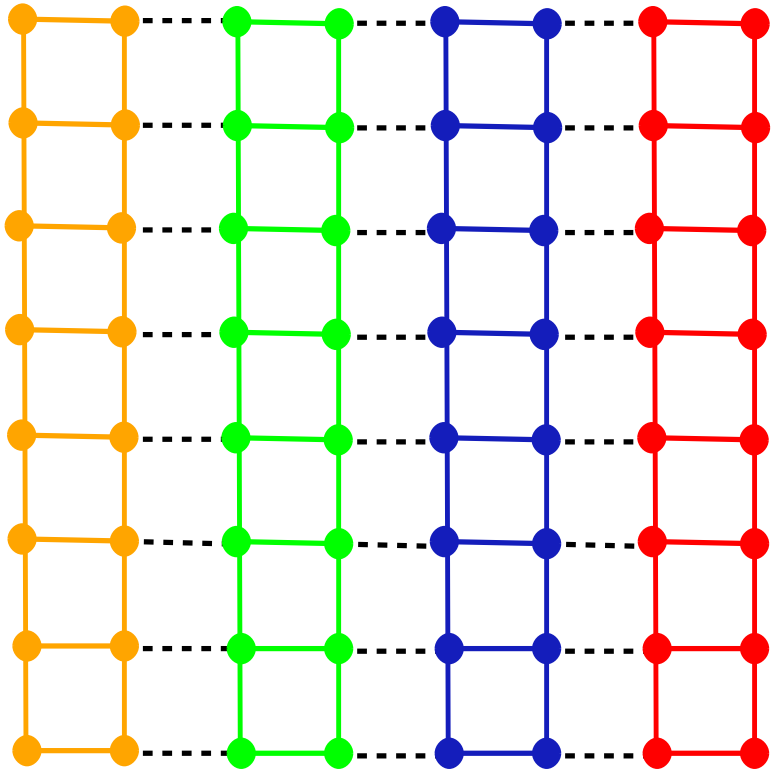
There are many ways to decompose an $8 \times 8$ mesh into two parts with an equal amount of unknowns. After a tiny bit of thinking, it is clear that splitting through the middle horizontally or vertically will produce the configuration requiring the smallest amount of communication:



# Mesh Partitioning - 4 Processors

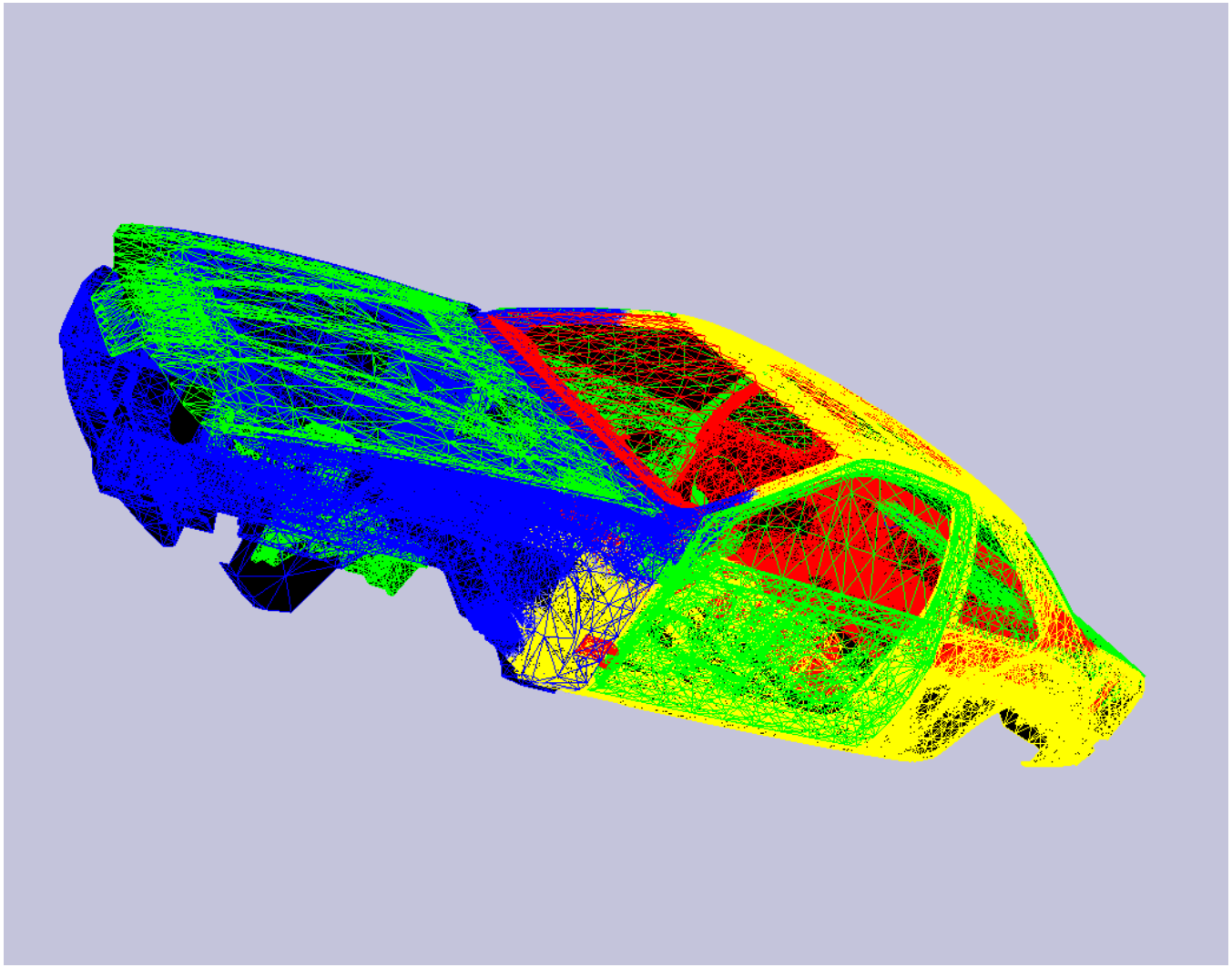
For four processors , things get slightly more complicated. Each of the following are possibilities. Either situation may be optimal, depending on the processor configuration:





# Mesh Partitioning - General Case

Partitioning a general mesh so that the load given to each processor is approximately the same, while still minimizing communication is an extremely difficult task and an active area of research.



# Future Directions

A fairly recent development in parallel computing is the use of GPUs.

- GPUs were developed to run advanced graphics and video games. They have a huge number of processing threads.
- GPUs are hundreds to thousands of times faster at certain calculations than a CPU, but struggle with general computing.
- Offloading certain calculations to a GPU can greatly speed up your computations.
- Special coding environments (OpenCL and CUDA) must be used, and codes need to written very carefully.
- An Intel Core i7 965 Extreme provides approximately $60$ GFLOPS and costs $\$1000$.
- An AMD/ATI Firestream 9270 SP provides $270$ GFLOPS and costs $\$1250$.

Bottom line:

- Multicore processor are here to stay, and you will have to learn how to use them properly.
- If you are interesting in computing, make sure to take a basic parallel programming class.
- A *faster* algorithm won't necessarily be better computationally, since it may be very difficult to parallelize.