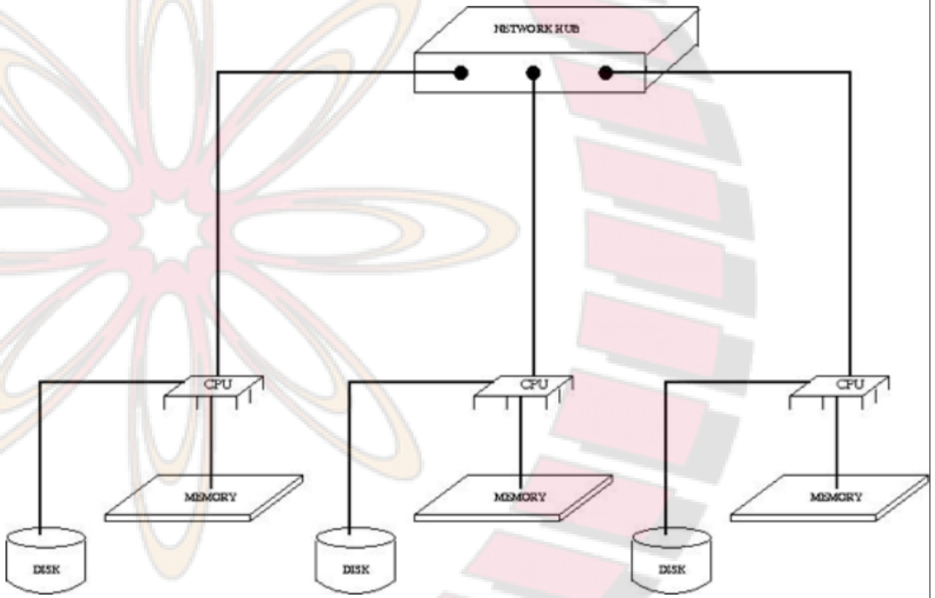
Distributed – Memory Machines

- Each node in the computer has a locally addressable memory space
- The computers are connected together via some high-speed network
  - Infiniband, Myrinet, Giganet, etc..

- Pros
  - Really large machines
  - Size limited only by gross physical considerations:
    - Room size
    - Cable lengths (10's of meters)
    - Power/cooling capacity
    - Money!
  - Cheaper to build and run
- Cons
  - Harder to program

    Data Locality



Sending a Message

Sending a message is a simple procedure. In our case the routine will look like this in C (the standard man pages are in C, so you should get used to seeing this format):

```
MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD)
```

&numbertosend	a pointer to whatever we wish to send. In this case it is simply an integer. It could be anything from a character string to a column of an array or a structure. It is even possible to pack several different data types in one message.
1	the number of items we wish to send. If we were sending a vector of 10 int's, we would point to the first one in the above parameter and set this to the size of the array.
MPI_INT	the type of object we are sending. Possible values are: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LING, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED Most of these are obvious in use. MPI_BYTE will send raw bytes (on a heterogeneous workstation cluster this will suppress any data conversion). MPI_PACKED can be used to pack multiple data types in one message, but it does require a few additional routines we won't go into (those of you familiar with PVM will recognize this).
0	Destination of the message. In this case PE 0.
10	Message tag. All messages have a tag attached to them that can be useful for sorting messages. For example, one could give high priority control messages a different tag then data messages. When receiving, the program would check for messages that use the the control tag first. We just picked 10 at random.
MPI_COMM_WORLD	We don't really care about any subsets of PEs here. So, we just chose this "default".

Receiving a Message

Receiving a message is equally simple and very symmetric (hint: cut and paste is your friend here). In our case it will look like:

```
MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD, &status)
```

&numbertoreceive	A pointer to the variable that will receive the item. In our case it is simply an integer that has has some undefined value until now.
1	Number of items to receive. Just 1 here.
MPI_INT	Datatype. Better be an int, since that's what we sent.
MPI_ANY_SOURCE	The node to receive from. We could use 1 here since the message is coming from there, but we'll illustrate the "wild card" method of receiving a message from anywhere.
MPI_ANY_TAG	We could use a value of 10 here to filter out any other messages (there aren't any) but, again, this was a convenient place to show how to receive any tag.
MPI_COMM_WORLD	Just using default set of all PEs.
&status	A structure that receive the status data which includes the source and tag of the message.

Send and Receive C Code

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv){

  int my_PE_num, numbertoreceive, numbertosend=42;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

if (my_PE_num==0){
  MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  printf("Number received is: %d\n", numbertoreceive);
}
else MPI_Send( &numbertosend, 1, MPI_INT,    0, 10, MPI_COMM_WORLD);

MPI_Finalize(); }
```

Steady State Heat Conduction

Phenomenon is modelled using Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \nabla^2 T = 0$$

which can be discretised on a grid as,