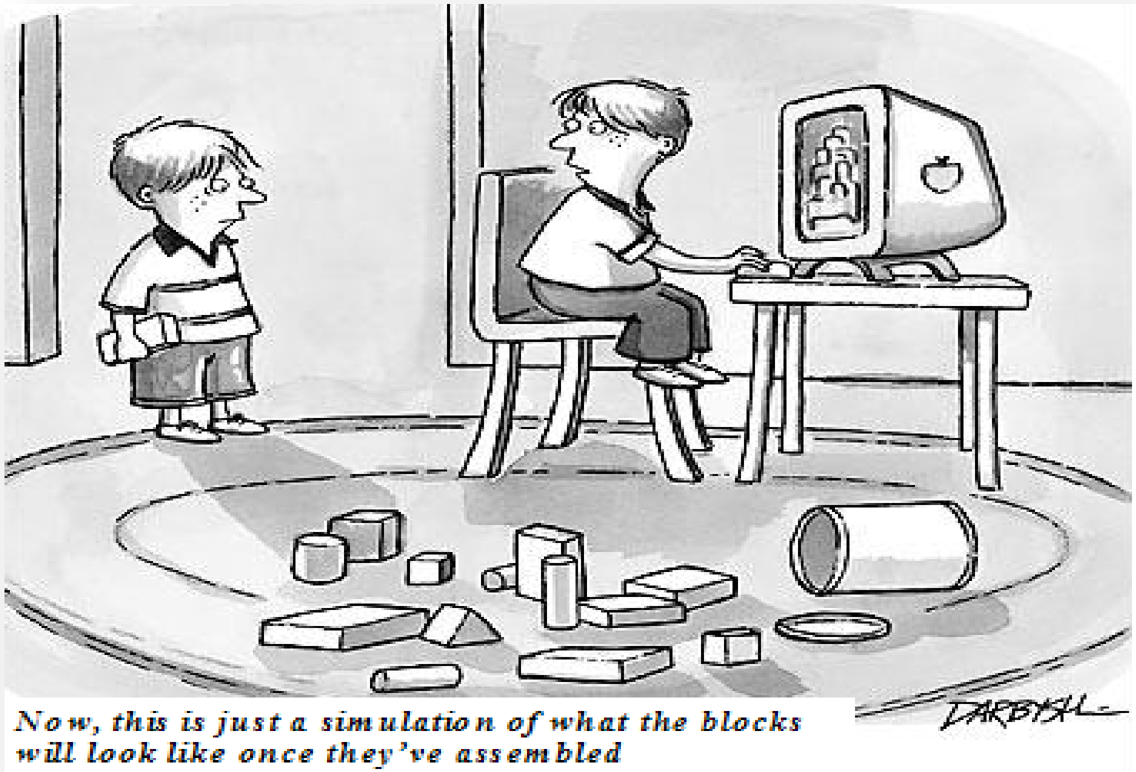
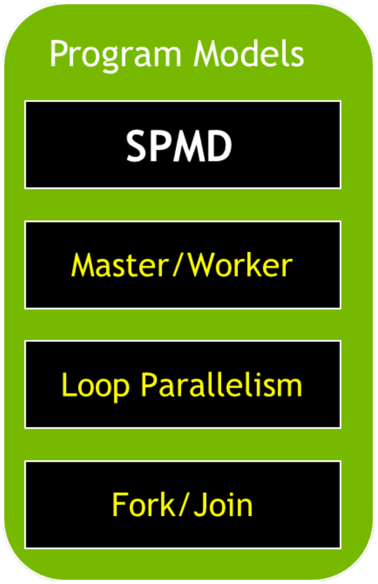
ECE569
Module 48



- MPI

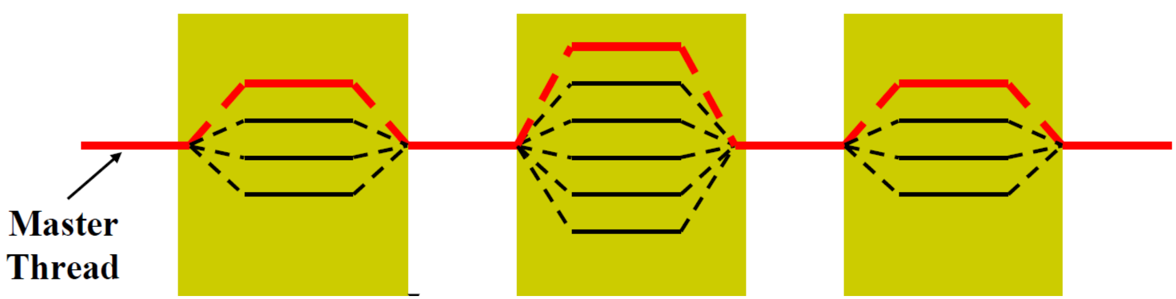
Coding Styles

- **SPMD (Single Program, Multiple Data)**
  - All PE’s (Processor Elements) execute the same program in parallel, but has its own data
  - CUDA Grid model (also OpenCL, MPI)
  - SIMD is a special case – WARP used for efficiency
- **Master/Worker**
  - A Master thread sets up a pool of worker threads
  - Workers execute concurrently, removing tasks until done
- **Loop Parallelism (OpenMP)**
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence
- **Fork/Join**
  - Most general, generic way of creation of threads



OpenMP Programming Model

- Master thread spawns a team of threads as needed
  – Managed transparently on your behalf
  – relies on low-level thread fork/join methodology to implement parallelism
  – The developer is spared the details
- Leveraging OpenMP in an existing code:
  – Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
- **Not Scalable**



SPMD Program

- **Dominant coding style of scalable computing**
  - MPI code is mostly developed in SPMD style
  - Many OpenMP code is also in SPMD
  - Particularly suitable for algorithms based on task parallelism and geometric decomposition.

SPMD is by far the most commonly used pattern for structuring massively parallel programs.

SPMD Program Phases

- **Initialize**
  - Establish localized data structure and communication
- **Obtain a unique identifier**
  - Each thread acquires a unique identifier, typically range from 0 to N-1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.
- **Distribute Data**
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads
- **Run the core computation**
- **Finalize**
  - Reconcile global data structure,

Data Sharing

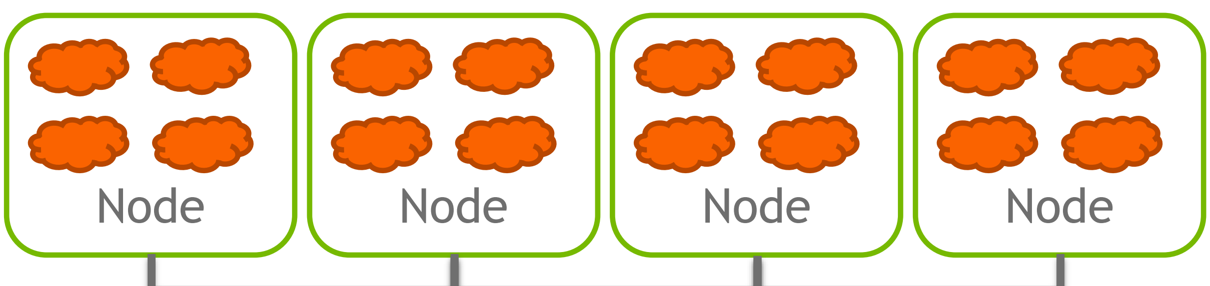
- **Data sharing can be a double-edged sword**
  - Excessive data sharing drastically reduces advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- **Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data**
  - Efficient use of on-chip, shared storage and datapaths
- **Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires more synchronization**

MPI

- A distributed memory model
  - processes exchange information by messaging
- API communication functions
  - Seamless interconnect network
- Processes address each other using logical numbers
- No cache coherence and no need for special cache coherency hardware
- Software development: more difficult to write programs: keep track of memory usage

MPI – Execution Model

- Many processes distributed in a cluster
- Each process computes part of the output
- Processes communicate with each other
- Processes can synchronize when collaborating on a large task
  - What differentiates processes is their rank: “branching based on the process rank”
- Very similar to GPU computing, where one thread did work based on its thread index

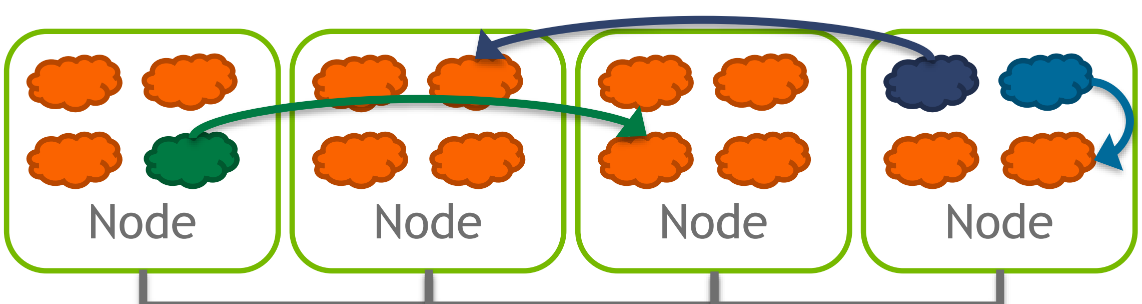


MPI Initialization, Info and Sync

- int MPI_Init(int *argc, char ***argv)
  - Initialize MPI
- int MPI_Comm_rank (MPI_Comm comm, int *rank)
  - Rank of the calling process in group of comm
- int MPI_Comm_size (MPI_Comm comm, int *size)
  - Number of processes in the group of comm
- int MPI_Abort (MPI_Comm comm)
  - Terminate MPI communication with an error flag
- int MPI_Finalize ( )
  - Ending an MPI application, close all resources

MPI Point to Point Communication: Sending Data

- **int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
  - Buf: Initial address of send buffer (choice)
  - Count: Number of elements in send buffer (nonnegative integer)
  - Datatype: Datatype of each send buffer element (handle)
  - Dest: Rank of destination (integer)
  - Tag: Message tag (integer)
  - Comm: Communicator (handle)



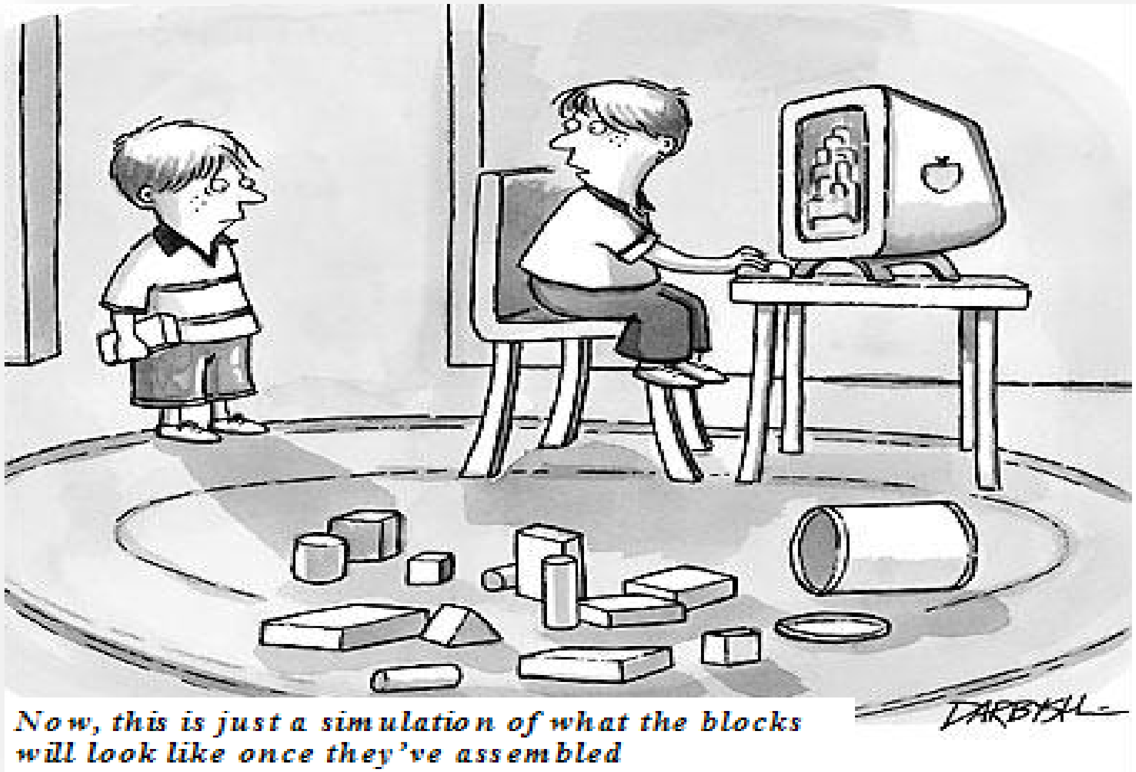
MPI Point to Point Communication: Receiving Data

- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - Buf: Initial address of receive buffer (choice)
  - Count: Maximum number of elements in receive buffer (integer)
  - Datatype:Datatype of each receive buffer element(handle)
  - Source: Rank of source (integer)
  - Tag: Message tag (integer) – MPI_ANY_TAG
  - Comm: Communicator (handle)
  - Status: Status object (Status)

Next

- **MPI Example**

ECE569
Module 49



- MPI Example: Vector Addition

MPI Example

```c
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
int my_rank; /* rank of process */
int p; /* number of processes */
int source; /* rank of sender */
int dest; /* rank of receiver */
int tag = 0; /* tag for messages */
char message[100]; /* storage for message */
MPI_Status status; /* return status for receive */

MPI_Init(&argc, &argv); // Start up MPI
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes
```

MPI Example

```c
 if (my_rank != 0) {
   /* Create message */
   sprintf(message, "Greetings from process %d!", my_rank);
   dest = 0;
   /* Use strlen+1 so that '\0' gets transmitted */
   MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
                                                  MPI_COMM_WORLD);
 }
 else { /* my_rank == 0 */
      for (source = 1; source < p; source++) {
         MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD,
         &status);
         printf("%s\n", message);
      }
 }
 MPI_Finalize(); // Shut down MPI
 return 0;
} /* main */
```

MPI Example – Vector Addition (main)

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
int vector_size=1024*1024*1024;
int np; /* number of processes */
int pid; /* rank */

MPI_Init(&argc, &argv); // Start up MPI
MPI_Comm_rank(MPI_COMM_WORLD, &pid); // Find out process rank
MPI_Comm_size(MPI_COMM_WORLD, &np); // Find out number of processes


If (np < 3 ) {
    if (pid == 0) printf (“need 3 or more processes\n”);
    MPI_Abort(MPI_COMM_WORLD,1);
    return 1;
}
```

MPI Example – Vector Addition (main)

```
if (pid < np-1)
  compute_node(vector_size/(np-1));
else
  data_server(vector_size)
MPI_Finalize(); // Shut down MPI
return 0;
} /* main */
```

MPI Example Vector Addition: Server(1)

```
void data_server(unsigned int vector_size) {
   int np, num_nodes = np - 1;
   unsigned int num_bytes  = vector_size * sizeof(float);
   float *input_a = 0, *input_b = 0, *output = 0;

   /* Set MPI Communication Size */
   MPI_Comm_size(MPI_COMM_WORLD, &np);

   /* Allocate input data */
   input_a = (float *)malloc(num_bytes);
   input_b = (float *)malloc(num_bytes);
   output = (float *)malloc(num_bytes);
   if(input_a == NULL || input_b == NULL || output == NULL) {
     printf("Server couldn't allocate memory\n");
     MPI_Abort( MPI_COMM_WORLD, 1 );
   }
   /* Initialize input data */
   random_data(input_a, vector_size , 1, 10);
   random_data(input_b, vector_size , 1, 10);
```

MPI Example Vector Addition: Server(2)

```
/* Send data to compute nodes */
float *ptr_a = input_a;
float *ptr_b = input_b;

for(int process=0; process < num_nodes; process++) {
    MPI_Send(ptr_a, vector_size / num_nodes, MPI_FLOAT,
        process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_a += vector_size / num_nodes;

    MPI_Send(ptr_b, vector_size / num_nodes, MPI_FLOAT,
        process, DATA_DISTRIBUTE, MPI_COMM_WORLD);
    ptr_b += vector_size / num_nodes;
}

/* Wait for nodes to compute */
MPI_Barrier(MPI_COMM_WORLD);
```

MPI Example Vector Addition: Server(3)

```
    /* Serve node ready to receive data from compute nodes */
    /* set up MPI_Status flag for error check
   /* Collect output data */
    MPI_Status status;
   for(int process = 0; process < num_nodes; process++) {
      MPI_Recv(output + process * vector_size / num_nodes,
        vector_size / num_comp_nodes, MPI_FLOAT, process,
       DATA_COLLECT, MPI_COMM_WORLD, &status );
   }

   /* Store output data */
   print_output(output,vector_size);

   /* Release resources */
   free(input_a);
   free(input_b);
   free(output);
}
```

MPI Example Vector Addition: Compute(1)

```
void compute_node(unsigned int vector_size ) {
int np;
unsigned int num_bytes = vector_size * sizeof(float);
float *input_a, *input_b, *output;
MPI_Status status;

MPI_Comm_size(MPI_COMM_WORLD, &np);
int server_process = np - 1;

/* Alloc host memory */
input_a = (float *)malloc(num_bytes);
input_b = (float *)malloc(num_bytes);
output = (float *)malloc(num_bytes);

/* Get the input data from server process */
MPI_Recv(input_a, vector_size, MPI_FLOAT,
    server_process,DATA_DISTRIBUTE,MPI_COMM_WORLD,
    &status);
MPI_Recv(input_b, vector_size, MPI_FLOAT,
    server_process,DATA_DISTRIBUTE, MPI_COMM_WORLD,
    &status);
```

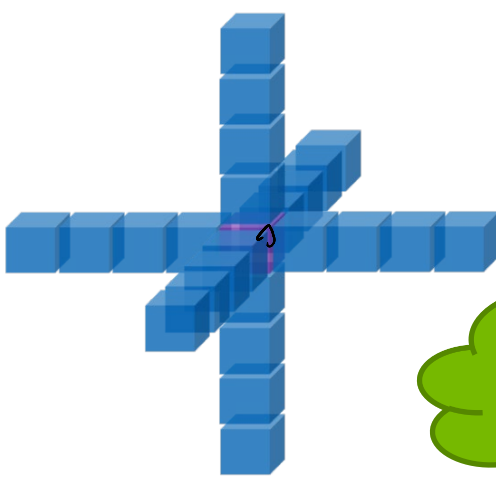
```
if (pid < np-1)
  compute_node(vector_size/(np-1));
else
  data_server(vector_size)
MPI_Finalize(); // Shut down MPI
return 0;
} /* main */
```

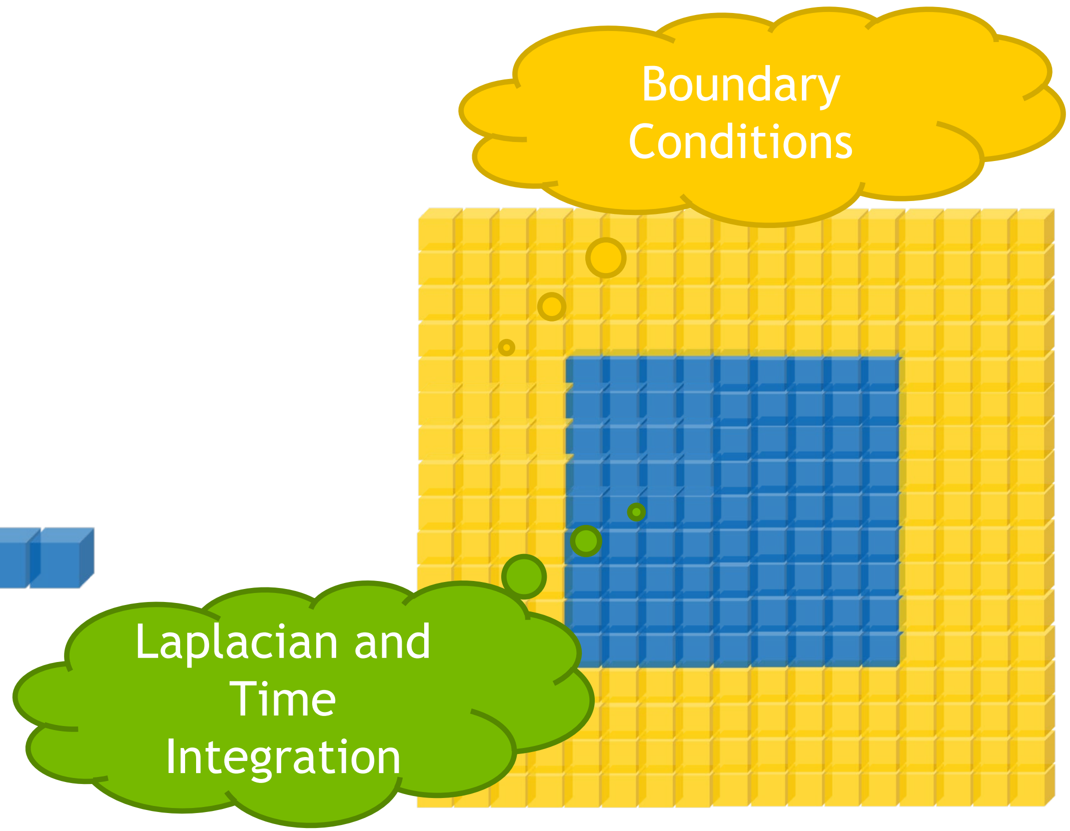
MPI Example Vector Addition: Compute(2)

```
    /* Compute the partial vector addition */
    for(int i = 0; i < vector_size; ++i) {
        output[i] = input_a[i] + input_b[i];
    }
    /* Replace serial execution with CUDA kernel  */
    /* same as your vector addition main function */
    /* and kernel code                            */
    /* Report to barrier after computation is done*/
    MPI_Barrier(MPI_COMM_WORLD);

    /* Send the output */
    MPI_Send(output, vector_size, MPI_FLOAT,
        server_process, DATA_COLLECT, MPI_COMM_WORLD);

    /* Release memory */
    free(input_a);
    free(input_b);
    free(output);
}
```

Next: Wave Propagation Example

- **Approximate Laplacian using finite differences**
- 3D Stencil, 4 points in each direction (x,y,z)
  - (i−4,j,k), (i−3,j,k), (i−2,j,k), (i−1,j,k), (i+1,j,k), (i+2,j,k), (i+3,j,k), (i+4,j,k), (i,j−4,k), (i,j−3,k), (i,j−2,k), (i,j−1,k), (i,j+1,k), (i,j+2,k), (i,j+3,k), (i,j+4,k), (i,j,k−4), (i,j,k−3), (i,j,k−2), (i,j,k−1), (i,j,k+1), (i,j,k+2), (i,j,k+3) and (i,j,k+4).





Next: Stencil Domain Decomposition

- **Volumes are split into tiles (along the Z-axis)**
  - 3D-Stencil introduces data dependencies