Lecture 10: Inter-Process Communication in Distributed Memory

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture

```
void homer() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_full);
        printf("Homer ate Cookie-%d\n", cookiejar->cookie);
        sem_post(&cookiejar->jar_empty);
    }
    cleanup_and_exit();
}
```

```
void marge() {
    for(int i=0; i<5; i++) {
        sem_wait(&cookiejar->jar_empty);
        printf("Marge bake Cookie-%d\n", ++cookiejar->cookie);
        sem_post(&cookiejar->jar_full);
    }
    cleanup_and_exit();
}
```

Homer Process (CPU 0)		Sem (empty)	Sem (full)	Marge Process (CPU 1)	
Action	**Process State**	**Sem Value**	**Sem Value**	**Action**	**Process State**
Forked	Ready	1	0	Forked	Ready
…….				…….	

Today's Class

- IPC in distributed memory
  - Message Passing Interface
- Quiz-2

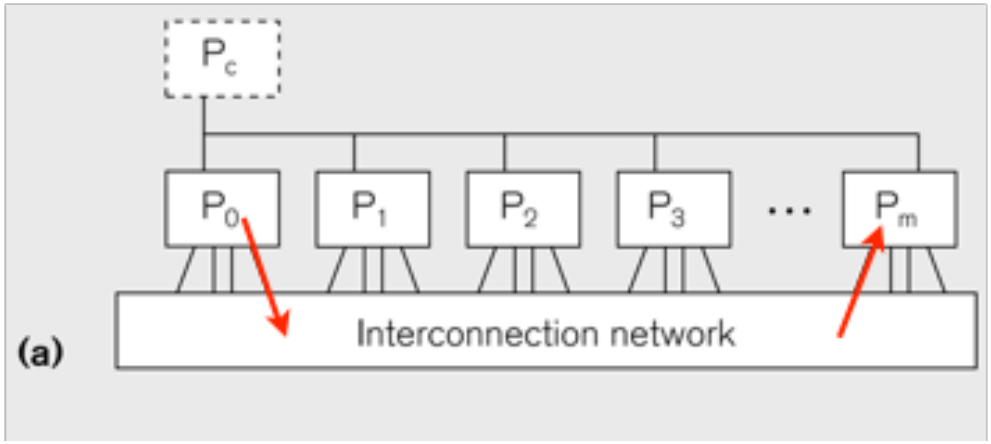
Distributed Memory

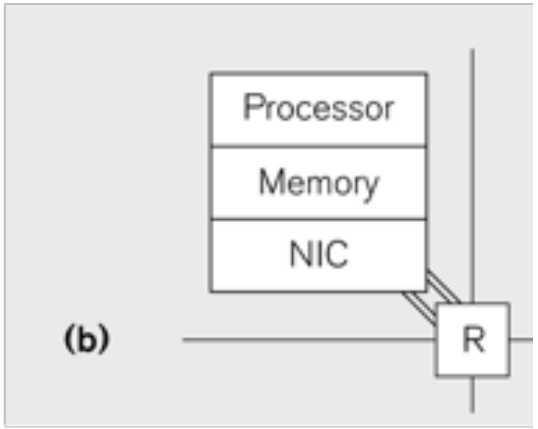
Picture source: https://cs.boisestate.edu/~amit/research/beowulf/cluster-small.jpg

- Several machines (e.g., workstations) connected together in a cluster with some interconnect (e.g., ethernet)
  - Easy to scale computing resources. E.g., ten desktops each having a 10-core processor would result in 100-core processing power in a cluster (a.k.a. Beowulf cluster)
  - Supercomputers are similar but with high-end processors and interconnects
  - Grid computing is also similar, but the machines are not inside a single room but can be in different countries/location connected with internet (it is a type of distributed computing instead of cluster computing)
- One big computation can be divided equally into N number of sub-computations that can run on N different processes
  - 1x1 mapping between a process and a core in the cluster

Organization of a Distributed Memory Multiprocessor

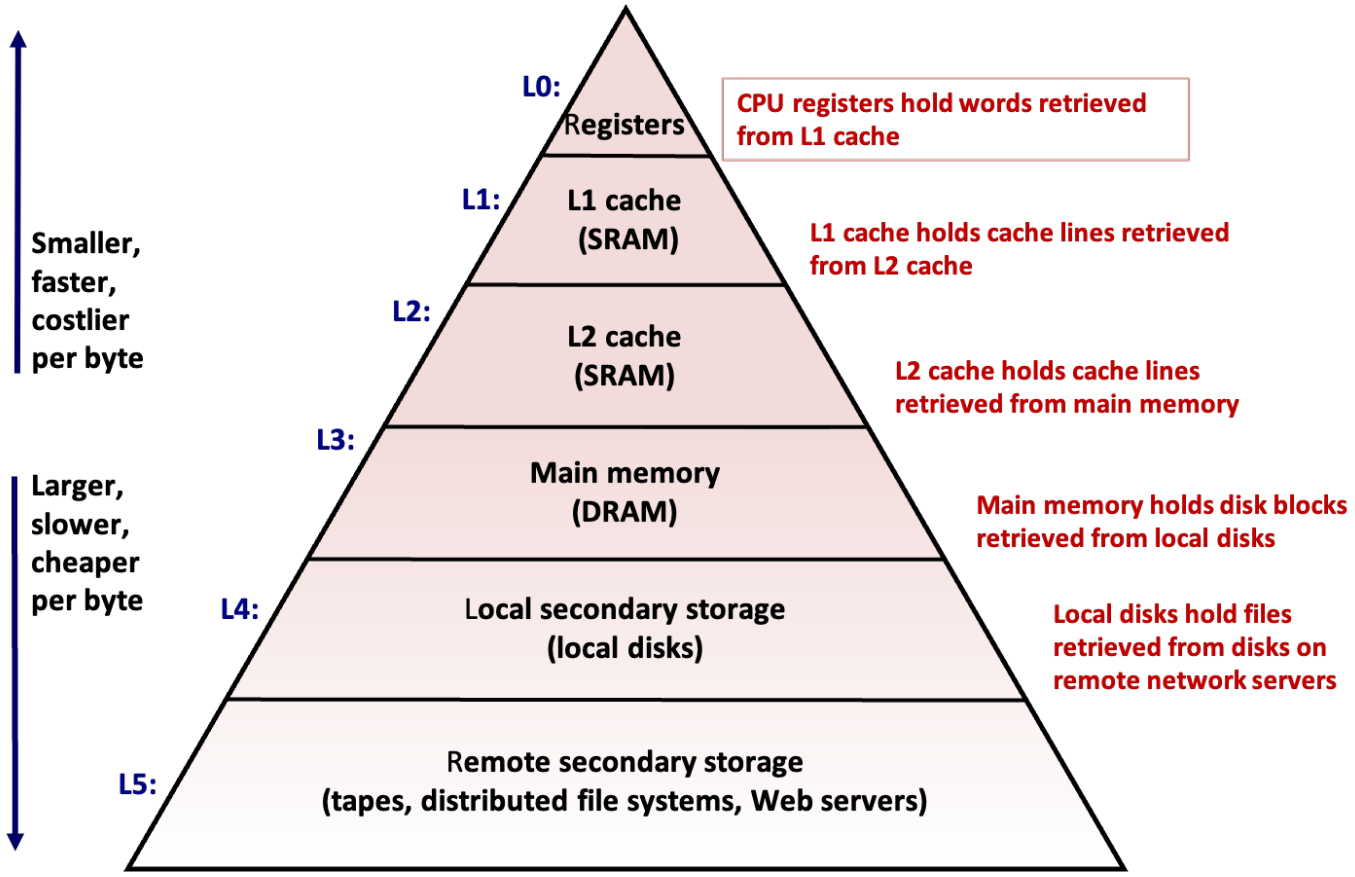
==> Processors communicate by sending messages via an interconnect





- Figure-a
  - Host processor (Pc) connected to a cluster of processor nodes (P0 … Pm)
  - Processors P0 … Pm communicate via an interconnection network
- Figure-b
  - Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router (R) in the interconnect

Distributed Memory



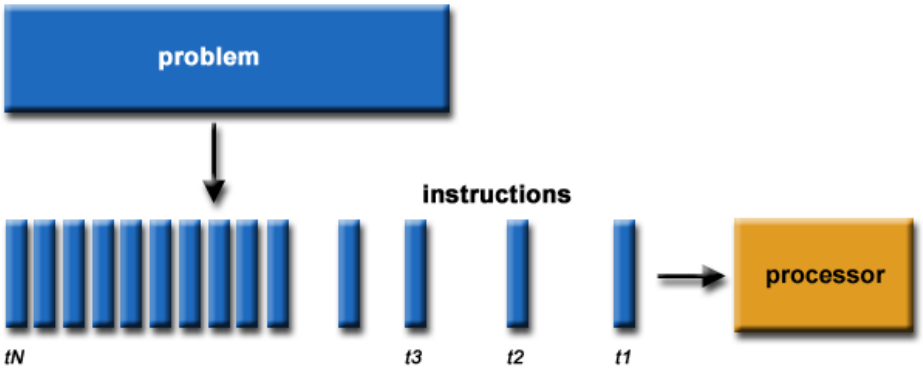
Picture source: http://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/lectures/09-memory-hierarchy.pptx

- Distributed memory access latency slowest in the memory hierarchy
- Each process has access to the memory on the local system as well as the memory on the remote machines
- Processes communicate with each other using inter-process communication

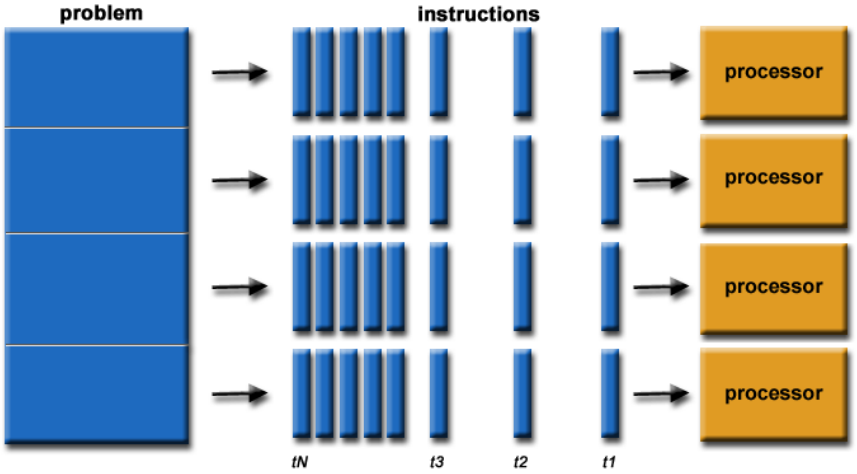
IPC in Distributed Memory

- Processes communicate in distributed memory by exchanging messages with each other
  - For example, client-server model that uses sockets based communication
- In this course we will only **introduce** a simple to use runtime library for exchanging messages between processes
  - Message Passing Interface (MPI) that provides a set of standard and portable APIs for **parallel computing** in distributed memory (as well as between the processes running in a shared memory)

Introduction to Parallel Computing

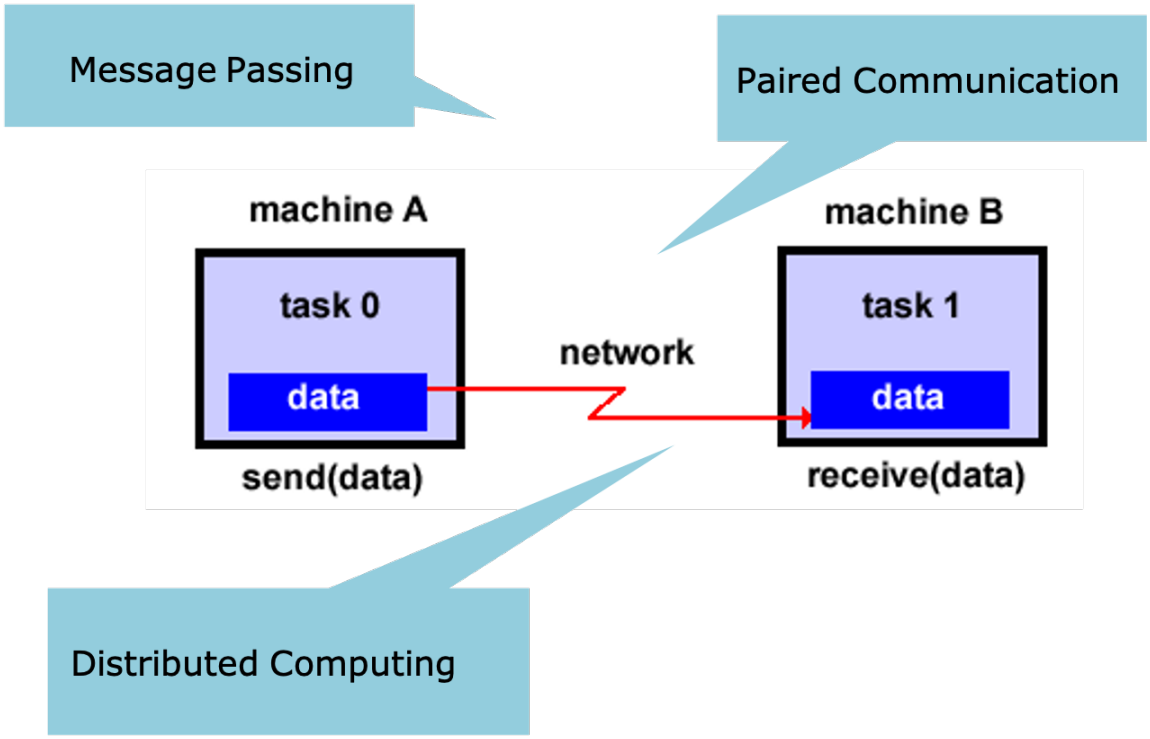


Serial (sequential) computing



Parallel computing

Message Passing Interface (MPI)



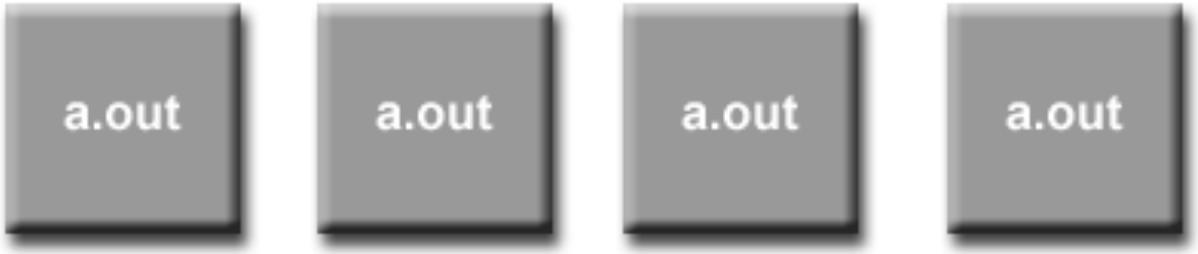
- The logical view of a machine supporting the message-passing paradigm consists of *p processes*, each with its own exclusive address space, that are capable of executing on different nodes in a distributed-memory multiprocessor
  1. Each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed.
  2. 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.
- In this loosely synchronous model, processes synchronize infrequently to perform interactions. Between these interactions, they execute completely asynchronously.

Single Program Multiple Data (SPMD)

```
void array_sum() {
    int A[SIZE]; //initialized
    int sum=0;
    if(fork() == 0) {
        printf("I am the child process\n");
        for(int i=SIZE/2; i<SIZE; i++) sum += A[i];
        exit(0);
    }
    printf("I am the parent process\n");
    for(int i=0; i<SIZE/2; i++) sum += A[i];
    wait(NULL);
    return 0;
}
```

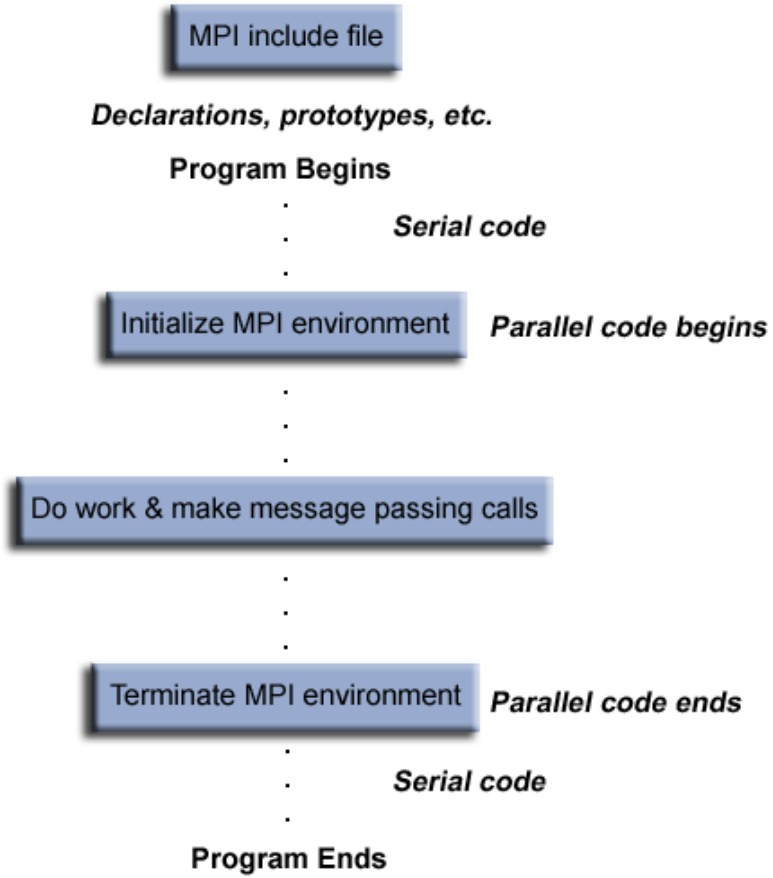
- **S**ingle **P**rogram executed by both the parent and child processes but with different halves of the same array (**M**ultiple **D**ata)
  - Parent calculate the sum of first half of the arrays whereas the child calculate the sum of the second half
- The problem(s)
  - How the child and parent would communicate their partial results to get the total sum?
  - What if we have a huge array size that could be faster to compute with multiple processes, some of which would be running on a shared memory whereas the rest on distributed memory?
    - Imagine the difficulty in writing such a program

SPMD Using the MPI



- Run the same program on P processing elements (PEs)
- Use the “rank” … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code

General MPI Program Structure



- Write one program that will execute on each process
- Each of the N processes will have a unique rank (0 – N-1)
- Processes will communicate via messages for non-local data accesses

The Hello World Program in MPI (1/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);









    MPI_Finalize();
    return 0;
}
```

- MPI_Init and MPI_Finalize APIs are used for initializing and cleanup the MPI environment (runtime)

The Hello World Program in MPI (2/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Finalize();
    return 0;
}
```

- Compile the program as "**mpicc program.c**" (first install MPI using **sudo apt-get install mpich**)
- Run the program as "**mpirun -np 4 ./a.out**" where 4 is the total number of MPI processes to be created
- MPI_COMM_WORLD is the **name** of the group that contains the processes created by the user using mpirun command (e.g., 4 above)
  - Total number of processes in the group (MPI_COMM_WORLD) can be enquired using the MPI_Comm_size API

The Hello World Program in MPI (3/3)

```
// the header file containing MPI APIs
#include <mpi.h>
int main(int argc, char **argv) {
    // Initialize the MPI runtime
    MPI_Init(&argc, &argv);
    int rank, nprocs;
    // Get the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    // Get the rank of this process in MPI_COMM_WORLD
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("My rank is %d in world of size %d\n", rank, nprocs);
    // Terminate the MPI runtime
    MPI_Finalize();
    return 0;
}
```

There is no user data being exchanged in this program. We will see message exchange in the next slides. Each process will simply print their rank and total number of processes invoked by the user

- Each of the processes will get a unique rank (id) in the group MPI_COMM_WORLD that can be enquired using the MPI_Comm_rank API
- MPI rank can be used to decide what will execute on which process

Parallel Array Sum Using MPI (1/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);









    MPI_Finalize();
}
```

- These four APIs are must in any parallel program being written using MPI
- There are more than 120 APIs in MPI, but we will only restrict our discussion to only six APIs

Parallel Array Sum Using MPI (2/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];









    MPI_Finalize();
}
```

- The array initialization is happening for each of the MPI process. Hence, each process has the exactly same array, but a separate copy in their own address space
  - In reality, the array could be read from disk in parallel by each processes (chunk size)
- For simplicity, we are assuming the array can be divided into equal sized chunk depending on the total number of MPI processes
- Each MPI process calculate the sum of its own chunk
- The variable “my_sum” is local to each process

Parallel Array Sum Using MPI (3/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0









    }
    MPI_Finalize();
}
```

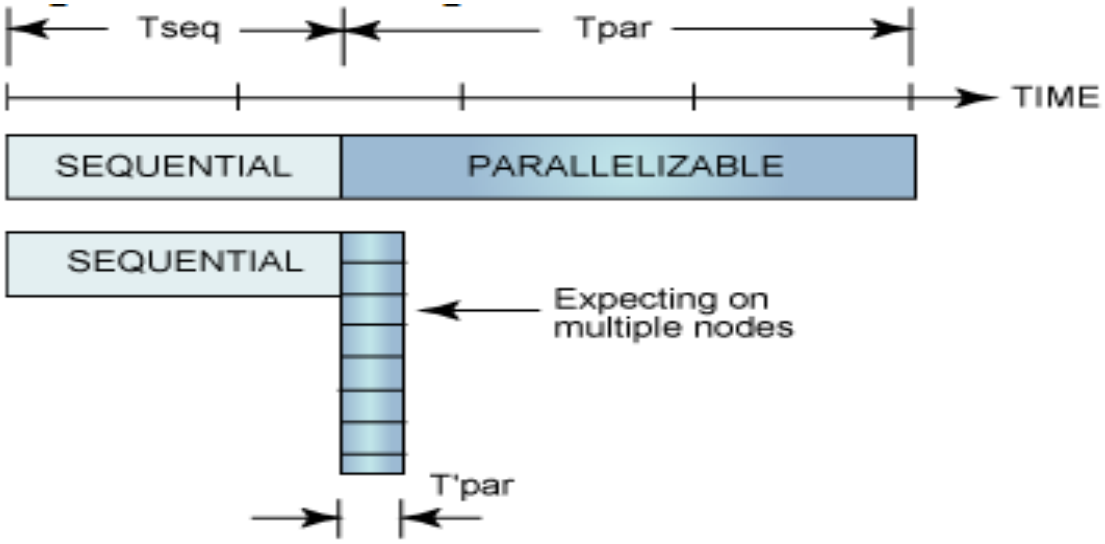
- Non-root rank use MPI_Send API to send the value of their local variable my_sum
- Parameters to MPI_Send are
  1. Address of the variable (or array) to be send
  2. Total count of data being sent (it will be the size of array when being sent)
  3. Platform independent name of the data type that MPI can understand (MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR, etc.)
  4. Rank of the receiver process (0 in this case)
  5. Tag (you can ignore it for now and simply pass 0 at all places)
  6. The group name MPI_COMM_WORLD

Parallel Array Sum Using MPI (4/4)

```
int main(int argc, char **argv) {
    int rank=0, nproc=4;
    MPI_Init(&argc, &argv);
    // 1. Get to know your world
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    int array[SIZE]; // initialized and assume (SIZE % nproc = 0)
    // 2. calculate local sum
    int my_sum = 0, chunk = SIZE/nproc;
    for (int i=rank*chunk; i<(chunk+1)*rank; i++) my_sum += array[i];
    // 3. All non-root processes send result to root processes (rank=0)
    if(rank > 0) {
        MPI_Send(&my_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else { // executed only at rank=0
        int total_sum = my_sum, tmp;
        for(int src=1; src<nproc; src++) {
          MPI_Recv(&tmp, 1, MPI_INT, src, 0, MPI_COMM_WORLD, NULL);
          total_sum += tmp;
        }
    }
    MPI_Finalize();
}
```

- Root rank will use (nproc-1) number of MPI_Recv API to receive the message being sent by (nproc-1) number of processes
- Parameters to MPI_Recv are exactly same as in MPI_Recv with few minor changes
  - The first parameter is the buffer used to store the message receive from the destination. It must be of the same size and datatype as in the sender side
  - The fourth parameter is the rank of the sender
  - You can ignore the last parameter and simply pass it as NULL

Amdahl's Law



- If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on
  - $T_{Total} = T_{seq} + T_{par}$
  - With infinite processors (or cores), $T_{par} = 0$ (theoretically) implying $T_{Total} = T_{seq}$

Reading Material

- Tutorial on MPI by LLNL
  - https://hpc-tutorials.llnl.gov/mpi/

Next Lecture

- Process scheduling
- Assignment 2 will be released tomorrow!