Basic Concepts of Parallel Programming

Ritu Arora, Texas Advanced Computing Center (TACC)

Abstract: This chapter presents a brief overview of some of the parallel programming concepts that should be understood before using IPT. These are: data parallelism, task parallelism, data distribution and collection, synchronization, and loop/data dependency.

Acknowledgement: *The content presented in this chapter was supported through the National Science Foundation Award #1642396: "SI2-SSE: An Interactive Parallelization Tool".* **Carlos Redondo,** from University of Texas at Austin, reviewed/edited the content.

Parallel programming involves the divide-and-conquer approach to reducing time-to-results. Some parallel programming concepts are explained in this chapter.

2.1. Data Parallelism

When the dataset on which the computations are desired is divided and distributed across the nodes (or servers) running the application instances in parallel (as shown in Figure 1), the application is said to exhibit data parallelism.

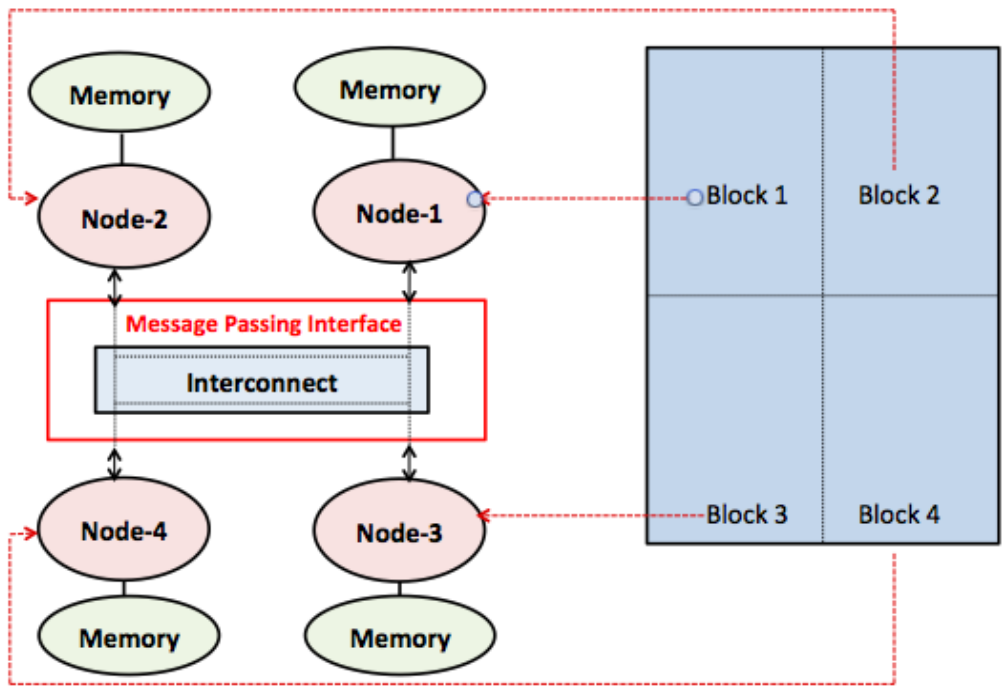

Figure 1. Each node (or server) running an instance of the application works with separate data block

With data parallelism, each processing element participating in a parallel computation performs the same task on different chunks of the data (e.g, for-loop iterations).

2.2. Task Parallelism

When the independent application tasks are distributed across the different processing elements and executed in parallel, then it is called task parallelism. Example, when one processing element performs sorting and the other processing element performs searching, we can say that the processing elements exhibit task parallelism.

2.3. Data distribution and collection

As parallel programming typically involves breaking-down large and complex computations into smaller pieces and running these pieces simultaneously on multiple cores or processors, each core or processor only has access to its local machine memory. There are mainly three ways in which the memory architecture for parallel computing may be set up: shared memory, distributed memory, and hybrid distributed-shared memory. Depending on which memory architecture is being used, the program will be decomposed and mapped to the separate processing elements, which will then perform the computation, and collect the data into a single result.

2.4. Synchronization

In executing a parallel program, it may be necessary to control access to certain shared variables or segments of code in order to prevent some common issues - such as dead-locks and race-conditions - due to thread conflicts. Synchronization typically involves either barriers, or locks, which a task may acquire or release. When a certain task needs to write to a shared variable, it acquires the lock for writing to that variable, and any other task needing to write to that variable must wait until that task has released the lock. When all tasks are executing a segment of code with a barrier, all tasks must wait after execution until every task has completed that segment of code.

2.5. Data/Loop Dependency

A data dependency results from multiple uses of the same location in storage by different tasks, and can inhibit parallel processing if the value at the location isn't synchronized or communicated between tasks. Loop dependency is a type of data dependency that results when data from a previous loop iteration is required for the computation of the following loop iteration. Each successive loop iteration is dependent on the last, and thus parallelism is inhibited.