Message Passing Fundamentals

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Topics

- The topics to be discussed in this chapter are
  - The basics of parallel computer architectures.
  - The difference between domain and functional decomposition.
  - The difference between data parallel and message passing models.
  - A brief survey of important parallel programming issues.

Parallel Architectures

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Parallel Architectures

- Parallel computers have two basic architectures: **distributed memory** and **shared memory**.
  - **Distributed memory** parallel computers are essentially a collection of serial computers (nodes) working together to solve a problem. Each node has rapid access to its own local memory and access to the memory of other nodes via some sort of communications network, usually a proprietary high-speed communications network. Data are exchanged between nodes as messages over the network.
  - In a **shared memory** computer, multiple processor units share access to a global memory space via a high-speed memory bus. This global memory space allows the processors to efficiently exchange or share access to data. Typically, the number of processors used in shared memory architectures is limited to only a handful (2 - 16) of processors. This is because the amount of data that can be processed is limited by the bandwidth of the memory bus connecting the processors.

Parallel Architectures

- The latest generation of parallel computers now uses a mixed shared/distributed memory architecture. Each node consists of a group of 2 to 16 processors connected via local, shared memory and the multiprocessor nodes are, in turn, connected via a high-speed communications fabric.

Problem Decomposition

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Problem Decomposition

- Roughly speaking, there are two kinds of decompositions.
  - Domain decomposition
  - Functional decomposition

Domain Decomposition

- In domain decomposition or "data parallelism", data are divided into pieces of approximately the same size and then mapped to different processors.
- Each processor then works only on the portion of the data that is assigned to it. Of course, the processes may need to communicate periodically in order to exchange data.

Domain Decomposition

- Data parallelism provides the advantage of maintaining a single flow of control. A data parallel algorithm consists of a sequence of elementary instructions applied to the data: an instruction is initiated only if the previous instruction is ended. Single-Program-Multiple-Data (SPMD) follows this model where the code is identical on all processors.
- Such strategies are commonly employed in finite differencing algorithms where processors can operate independently on large portions of data, communicating only the much smaller shared border data at each iteration.

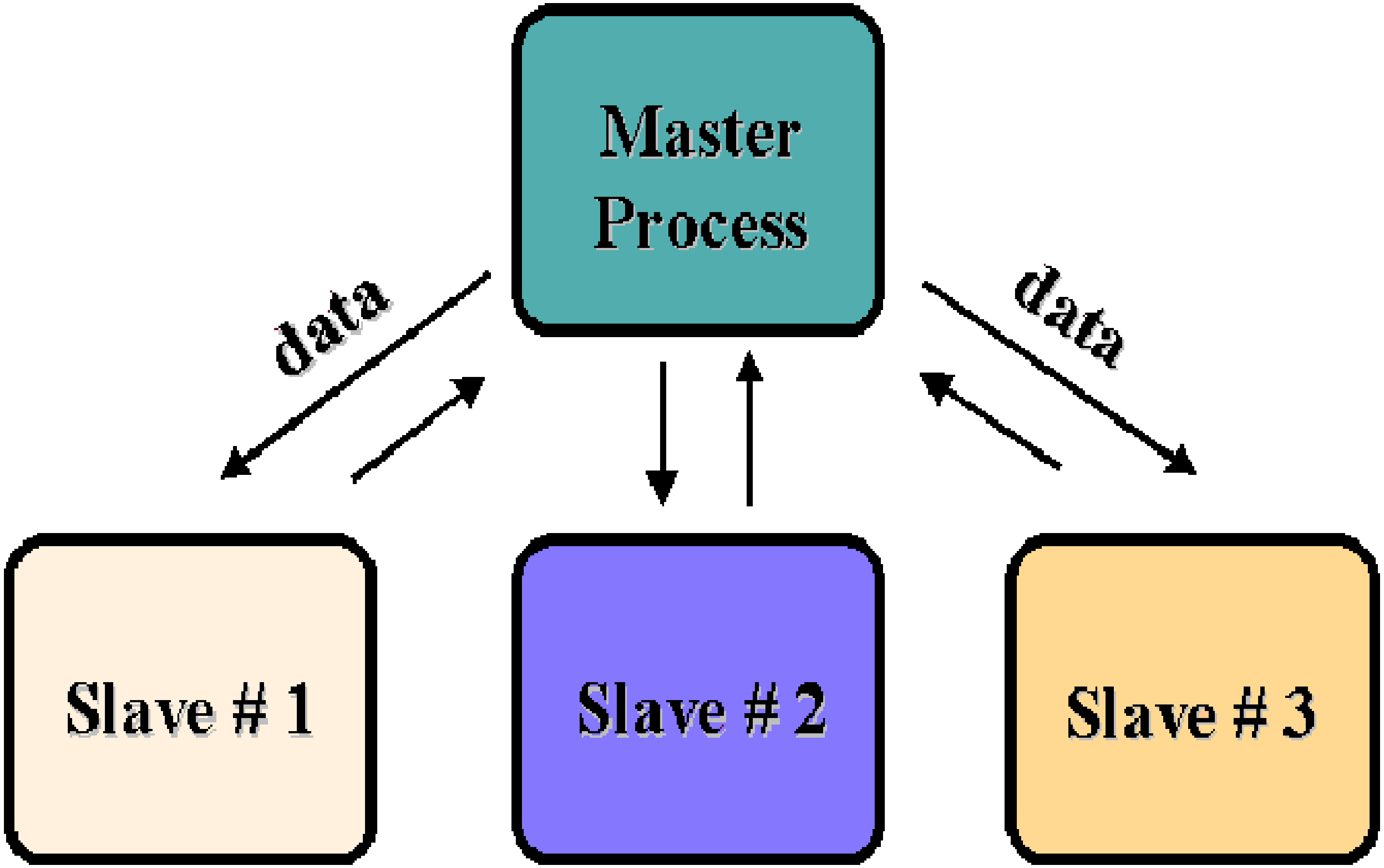
Functional Decomposition

- Frequently, the domain decomposition strategy turns out **not** to be the most efficient algorithm for a parallel program. This is the case when the pieces of data assigned to the different processes require greatly different lengths of time to process. The performance of the code is then limited by the speed of the slowest process. The remaining idle processes do no useful work. In this case, functional decomposition or "task parallelism" makes more sense than domain decomposition. In task parallelism, the problem is decomposed into a large number of smaller tasks and then, the tasks are assigned to the processors as they become available. Processors that finish quickly are simply assigned more work.

Functional Decomposition

- Task parallelism is implemented in a client-server paradigm. The tasks are allocated to a group of slave processes by a master process that may also perform some of the tasks.
- The client-server paradigm can be implemented at virtually any level in a program.
  - For example, if you simply wish to run a program with multiple inputs, a parallel client-server implementation might just run multiple copies of the code serially with the server assigning the different inputs to each client process. As each processor finishes its task, it is assigned a new input.
  - Alternately, task parallelism can be implemented at a deeper level within the code.

Functional Decomposition



Data Parallel and Message Passing Models

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Data Parallel and Message Passing Models

- There have been two approaches to writing parallel programs. They are
  - use of a directives-based data-parallel language, and
  - explicit message passing via library calls from standard programming languages.

Data Parallel and Message Passing Models

- In a **directives-based data-parallel language**
  - Such as High Performance Fortran (HPF) or OpenMP
  - Serial code is made parallel by adding directives (which appear as comments in the serial code) that tell the compiler how to distribute data and work across the processors.
  - The details of how data distribution, computation, and communications are to be done are left to the compiler.
  - Usually implemented on shared memory architectures because the global memory space greatly simplifies the writing of compilers.
- In the **message passing approach**
  - It is left up to the programmer to explicitly divide data and work across the processors as well as manage the communications among them.
  - This approach is very flexible.

Parallel Programming Issues

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Parallel Programming Issues

- The main goal of writing a parallel program is to get better performance over the serial version. Several issues that you need to consider:
  - Load balancing
  - Minimizing communication
  - Overlapping communication and computation

Load Balancing

- **Load balancing** is the task of equally dividing work among the available processes.
- This can be easy to do when the same operations are being performed by all the processes (on different pieces of data).
- When there are large variations in processing time, you may be required to adopt a different method for solving the problem.

Minimizing Communication

- Total execution time is a major concern in parallel programming because it is an essential component for comparing and improving all programs.
- Three components make up execution time:
  - Computation time
  - Idle time
  - Communication time

Minimizing Communication

- **Computation time** is the time spent performing computations on the data.
- **Idle time** is the time a process spends waiting for data from other processors.
- Finally, **communication time** is the time it takes for processes to send and receive messages.
  - The cost of communication in the execution time can be measured in terms of latency and bandwidth.
  - **Latency** is the time it takes to set up the envelope for communication, where **bandwidth** is the actual speed of transmission, or bits per unit time.
  - Serial programs do not use inter-process communication. Therefore, you must minimize this use of time to get the best performance improvements.

Overlapping Communication and Computation

- There are several ways to minimize idle time within processes, and one example is overlapping communication and computation. This involves occupying a process with one or more new tasks while it waits for communication to finish so it can proceed on another task.
- Careful use of nonblocking communication and data unspecific computation make this possible. It is very difficult in practice to interleave communication with computation.

END

Reference:
http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

Reference: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/