Parallel computing concepts

NCAR's production supercomputers are clusters of symmetric multiprocessor (SMP) nodes. While these systems can run single processes on a single processor, high-performance computing requires having many processors work *in parallel* so compute-intensive programs can run to completion in a reasonable amount of wall-clock time. Some clusters used for data analysis and visualization have both CPUs and GPUs.

Unless you have hands-on experience with multiprocessor cluster systems, you may need to learn some new techniques before you can run parallel programs efficiently and economically.

This introduction to parallel computing concepts will help prepare you to run your programs successfully on our systems.

Page contents

Parallel computing overview

In cluster system architecture, groups of processors (36 cores per node in the case of Cheyenne and as many as 128 on Derecho compute nodes) are organized into hundreds or thousands of nodes, within which the CPUs communicate via shared memory. Nodes are interconnected with a communication fabric that is organized as a network. Parallel programs use groups of CPUs on one or more nodes.

To exploit the power of cluster computers, parallel programs must direct multiple processors to solve different parts of a computation simultaneously. To be efficient, a parallel program must be designed for a specific system architecture. It also must be tailored to run on systems that differ in the number of CPUs connected by shared memory, the number of memory cache levels, how those caches are distributed between CPUs, and the characteristics of the communication mechanism for message passing.

You also need to understand how to use each computer's system software and the services that help you run your code on that platform. Your ability to work productively on these complex computing platforms is greatly enhanced by system-specific services such as compilers that offer multiple levels of code optimization, batch job schedulers, system resource managers for parallel jobs, and optimized libraries.

To become a proficient user of these powerful computing systems, you need to understand all three of these aspects of parallel computing and how they relate to each other:

- System hardware architecture
- User code tailored to computer hardware
- System software and services that enable users to modify their code for various platforms

There are many reasons to become a proficient user of these systems and make fair use of their resources. One is that you make efficient use of the computer time that you receive with your allocation. What's more, when your code runs efficiently, you reduce your job turnaround time (wall-clock time) and your job has less exposure to execution delays on these heavily subscribed computers. Reducing your job's wall-clock time makes more of your time available for other activities, and better use of the system also benefits the entire user community.

Programming issues

For complex parallel programs, modern cluster computers require two basic things from programmers. First, they need to design their codes for the system architecture. Then, at runtime, programmers must control the software agent that manages task scheduling and system resources. You control this task scheduling and system resource configuration by setting environment parameters at runtime.

A parallel job consists of multiple streams of program instructions executing simultaneously. Another name for an instruction stream is "thread." A process may have one or more threads. All threads associated with a process must run on the same node because they communicate using the main memory they share on that node (shared memory). Multiple processes can run on one node or multiple nodes. When multiple processes that are part of one application run on multiple nodes, they must communicate via a network (message passing). Since network speeds are slower than memory fetches and disk accesses, programmers need to design their codes to minimize data transfers across the network.

Parallel jobs do not always involve multiple processes. A single process with multiple threads of execution also is considered a parallel job. Processes running threads can be combined with other processes that may also be using threads. When parallel programs use both threads and message passing, those are called "hybrid" jobs.

Interprocess communication often is the main performance problem for jobs running on cluster systems. Library calls are available to help users perform efficient communication between processes running on a node and/or processes running on multiple nodes. These libraries often are provided by the computer vendor or by third-party software vendors. The MPI library, which is described below, is a good example.

Cluster systems organize memory into a hierarchy of levels of proximity to the CPUs. Typically there is a large, slow main memory with one or more levels of very small but very fast cache memory between the CPU and the main memory.

Users must design their programs to use the data cached closest to each CPU as much as possible before requiring time-consuming data movement away from the CPU. Programmers must devote considerable effort to managing the flow of data into and out of the hierarchical memories as their jobs are running.

You can maximize CPU performance and the speed at which your code runs by designing your programs to use the differently sized memory caches efficiently. You can also modify your code so the compiler and its optimization routines can improve it for a specific architecture.

The following sections describe the basic concepts that will help you understand these issues.

Parallel programming paradigms

Parallel programming paradigms involve two issues:

- Efficient use of CPUs on one process
- Communication between nodes to support interdependent parallel processes running on different nodes and exchanging mutually dependent data

A parallel program usually consists of a set of processes that share data with each other by communicating through shared memory over a network interconnect fabric.

Parallel programs that direct multiple CPUs to communicate with each other via shared memory typically use the **OpenMP** interface. The independent operations running on multiple CPUs within a node are called threads.

Parallel programs that direct CPUs on different nodes to share data must use message passing over the network. These programs use the **Message Passing Interface** (MPI).

Finally, programs that use carefully coded hybrid processes can be capable of both high performance and high efficiency. These hybrid programs use both OpenMP and MPI.

Two parallel programming paradigms: threads and message passing

As stated above, there are two ways to achieve parallelism in computing. One is to use multiple CPUs on a node to execute parts of a process. For example, you can divide a loop into four smaller loops and run them simultaneously on separate CPUs. This is called **threading**; each CPU processes a thread.

The other paradigm is to divide a computation into multiple processes. This causes each of the processes to depend on the same data. This interdependence requires processes to pass messages to each other over a communication medium. When processes on different nodes exchange data with each other, it is called **message passing**.

Programmers must define parallel threads explicitly. They also must direct the process intercommunication for message passing. They do both of these by determining where and how to insert the system-supplied library function calls in their code.

For threading or message passing to be effective, programmers must place these function calls in the code in logically appropriate locations. These locations must be determined not only by code function but also by considering CPU restrictions (such as CPU-cache coherence restrictions that are specific to the system architecture).

Both threading and message passing paradigms divide up the data domain (called domain decomposition). Threads use the OpenMP interface within a node and message passing uses the MPI interface for multiple nodes. The programmer decomposes the data domain into parts that individual CPUs can manage. Then each CPU that processes its domain communicates its computational results with its neighboring domains.

When you develop a new code for a cluster system, you can reduce your troubleshooting and debugging effort by starting with a single process that runs on a single CPU. Then time your runs as you adjust your code for increased efficiency. For timing your runs, it is best to use a performance tool that shows the slowest portions of your code. This allows you to focus on basic performance issues. When the program is running efficiently, you can divide loops into parts that can run simultaneously. When your multithreaded code runs properly, you can add processes and implement message passing. The intent of this approach is to deal with inevitable difficulties step by step as your code becomes more complex.

Hierarchical memory management and caching

Commodity component memories are composed of main memory and one or more levels of memory caches. The caches are provided to minimize the time delay for data being moved from memory to the CPU. As data moves from main memory to a CPU, the access speed (time between the fetch instruction and the processing of the data) increases, but the capacity for holding the data decreases because each cache closer to the CPU is smaller than the previous cache.

Programmers must maximize the use of data in the Level 1 cache (closest to the CPU) where it can be processed most efficiently. The second and often third cache levels and the main memory hold increasingly more data but with increasingly more delay before it can be processed.

Efficient management of data flowing through memory caches into and out of the CPU can produce a significant improvement in code performance. Therefore, programmers seeking to optimize their code need to devote serious time and consideration to the logic of moving the data stream through the caches to the CPUs and back.