"1st International Symposium on Computing in Informatics and Mathematics (ISCIM 2011)"

in Collabaration between EPOKA University and "Aleksandër Moisiu" University of Durrës

on June 2-4 2011, Tirana-Durres, ALBANIA.

MPI and Pthreads in parallel programming

Sidita DULI[1]

[1]*Department of Mathematics and Informatics, University "Luigj Gurakuqi", Shkoder*
Email : siditaduli@yahoo.com

ABSTRACT

By programming in parallel, large problem is divided in smaller ones, which are solved concurrently. Two of techniques that makes this possible are *Message* Passing Interface (MPI) and POSIX threads (Pthreads). In this article is highlighted the difference between these two different implementation of parallelism in a software by showing their main characteristics and the advantages of each of them.

INTRODUCTION

The goal of processing in parallel consists of taking a large task and divides it in smaller tasks that can be processed in the same time simultaneously. In this way, the whole task is completed faster than if it would be executed in one and big task.

Parallel computing requires this architecture:

1. The hardware of the computer that will execute the task should be designed in the way to work with multiple processors and to enable the communication between the processors.
2. The operating system of the computer should enable the management of many processors.
3. The software should be capable of breaking large tasks into multiple smaller tasks which can be performed in parallel.

The advantages that offers the parallel processing is increasing the power of processing, making a higher throughput and better performance for the same price. Most of these advantages can benefit those applications that can break larger tasks into smaller parallel tasks and that can manage the synchronization between those tasks. On the other hand, the performance should be higher to justify the overhead of parallelism

BASIC FEATURES OF MESSAGE PASSING PROGRAMS

Message passing programs consist of multiple instances of a serial program that communicate by library calls. These calls may be roughly divided into four classes:

1. Calls used to initialize, manage, and finally terminate communications.
2. Calls used to communicate between pairs of processors.
3. Calls that perform communications operations among groups of processors.
4. Calls used to create arbitrary data types.

The first class of calls consists of calls for starting communications, identifying the number of processors being used, creating subgroups of processors, and identifying which processor is running a particular instance of a program.

The second class of calls, called point-to-point communications operations, consists of different types of send and receive operations.

The third class of calls is the collective operations that provide synchronization or certain types of well-defined communications operations among groups of processes and calls that perform communication/calculation operations.

The final class of calls provides flexibility in dealing with complicated data structures. The following sections of this chapter will focus primarily on the calls from the second and third classes: point-to-point communications and collective operations.

There are two key attributes that characterize the message-passing programming paradigm. The first is that it assumes a partitioned address space and the second is that it supports only explicit parallelisation. The machine should have p processes.[1] Each of these processes has its own address space. There are two implications of a partitioned address space:

- Each data must belong to one of the partitions address space, which is each data must be explicitly partitioned and place. In this case the programming becomes more difficult, but the performance is higher because the processor can access its local data much faster than non-local data on such architectures.

-All interactions, operations like read and write, require cooperation of both processes; the one that has the data and that which wants to access the data. This requirement adds the complexity for some reasons. The participation of both processes happens even in the case where the process that has the data has nothing to do with the events at the requesting process. In this way the program becomes complex. An advantage of this type of programming is that it can be efficiently implemented on a wide variety of architectures.

The message-passing requires that all the process of parallelism is explicitly coded by the programmer. He must analyse the algorithm, must define which part can be performed in parallel. This is why the programmer must be a highly qualified one.

MESSAGE PASSING MODEL

MPI is a standard implementation of the "message passing" model of parallel computing. [2]

-A parallel computation consists of a number of **processes**, each working on some local data. Based on the MPP architecture, each process has its own local variables, and there is no mechanism for any process to **directly** access the memory of another. The only way to share the data between processes is by using the message passing, that is, by explicitly sending and receiving data between processes.

In a more general view, the model involves **processes**, which need not, in principle, be running on different **processors**.

The main reason why this model is useful is because that it is extremely general. All types of parallel computation can be cast in the message passing form. Other features are:

-can be implemented on many types of platforms, like shared-memory multiprocessors, networks of workstations or single-processor machines.

-allows more control over data location and flow within a parallel application than in, for example, the shared memory model. By using it, programs can reach higher performance using explicit message passing. And of course, performance is a good reason why message-passing will never disappear from the parallel programming world.

The model which is the simplest for the programmer is of course the sequential model. To the programmer seems very normal to think he is programming for a single processor, which has a memory. The message-passing is the model for executing the program in the parallel, so when he will program, he will think of some processors, each of them with its own memory. The program will be run in each processor. But they need to communicate and to exchange data between them. This message-passing model takes the name from this technique of communication, from the way they send data to each other by sending e message to the other processor. This means that this is the only way of communication, and not by accessing directly somehow the other processor's memory. Otherwise the message would be not used. The message passing is just e message sent to another processor, and in programming techniques just a subroutine is called.

Programs written in a message-passing style can run on distributed or shared memory multi-processors, networks of workstations, or even uni-processor systems. Message-passing is popular, not because it is particularly easy, but because it is so *general*.

The message passing model is defined as:

1. Set of processes having only local memory
2. Processes communicate by sending and receiving messages
3. The transfer of data between processes requires cooperative operations to be performed by each process (a send operation must have a matching receive)

One other model is data parallelism, where the parallelism is enabled by partitioning the data. Other models are the one is mentioned above, the shared memory, the remote memory operation, where a process can access the memory of another process even without its permission. Threads are also an alternative model where a single process has multiple executions paths. Some hybrid versions are available with two of these models.

The major goal of MPI is to increase the possibility of implementing it in different platforms. The expectation is for a degree of portability comparable to that given by programming languages such as FORTRAN. The same MPI code can be executed in all types of machines where its library is available.

As mentioned above, the message-passing way to exchange data is mostly used in the distributed architecture. But the same code in MPI can be used as well in shared-memory architecture. It can run on a network of workstations or in a single workstation in which are working several processes. Being used in a wide variety of computers, it gives a high degree of flexibility in code development, debugging, and in choosing a platform for production runs.

MPI is not used only in homogeneous systems. It is also compatible in a collection of processors part of a system with different architectures. This is because the MPI implementation provides a virtual computing model where the architectural model is

hidden. In this case, the user will not think if the message will be sent in a particular architecture, or which part of the system will be the receiver processor. MPI does all the data conversion, the changes needed for a different architecture, will choose the correct protocol to send the data, etc.

MPI of course allow the implementations in a homogeneous system. In the case when the user wants to implement it in a system with different architectures, he must use a MPI system designed to support heterogeneity.

A sequential algorithm can be used in any architecture that supports the specific sequential paradigm. But this is not enough for programmers. They want that this algorithm should be portable. The same is true for message-passing programs and forms the motivation behind MPI. MPI provides source-code portability of message-passing programs written in C or FORTRAN across a variety of architectures. As for the sequential case, this has many benefits, including

• protecting investment in a program

• allowing development of the code on one architecture (e.g. a network of workstations)

Before running it on the target machine (e.g. fast specialist parallel hardware)

The concept of processes communicating by sending messages to one another has been understood some years ago but only recently it is only it has been developed the message-passing systems which allow source-code portability.

MPI was the first effort to produce a message-passing interface standard across the whole parallel processing community.

A good point of using the message-passing is its wide portability. The programs using MPI libraries may run on distributed-memory multicomputer, shared-memory multiprocessors, networks of workstations, and combinations of all of these. MPI is implemented on a great variety of machines, including those "machines" consisting of collections of other machines, parallel or not, connected by a communication network.

The architecture might be MIMD (Multiple Instruction stream, Multiple Data stream), where each process follows a distinct execution path through the same code, or even executes a different code. It might be also the SPMD (Single Process, Multiple Data), where all processes follow the same execution path through the same program.

PTHREAD

Pthreads are a set of types and procedure calls in C. They are implemented in a *pthread.h* file, and a thread library. [3]

The primary motivation for using threads is to realize potential program performance gains. Comparing to processes, creating a thread requires fewer operations, and to manage a thread requires fewer system resources.

Another advantage is related to the software portability. Applications that use threads can be developed on serial machines and run on parallel machines without changing anything. This portability is very significant advantage of threaded APIs.

A feature of threaded applications is the hiding of the latency. One of the major overheads in programs is the access of latency for memory access, I/O and communication. Multiple threads executed in the same processor so the latency is hidden.

In the case when a thread is waiting to an input from the user, other threads can use the CPU for other operations. So the latency of the user does not reflect in the overall work of the process.

A programmer must do scheduling in the thread-level the threaded application. He must express concurrency between threads for the same shared address in order that it is minimised the overheads of remote interaction and idling. In a structured application the task of balancing the work of the threads with the processor is not so difficult. But in cases when the application is unstructured, the balancing process may be difficult for the programmer.

The last advantage that has programming with threads because that they are easier to write than corresponding programs using message passing APIs. Achieving identical levels of performance for the two programs may require additional effort, however. With widespread acceptance of the POSIX thread API, development tools for POSIX threads are more widely available and stable.

The advantages of using threaded applications comparing the non-threaded applications are:

- Overlapping CPU and I/O devices: processing in the same time long I/O operations and CPU that is performing other threads.
- Scheduling based in the priority: inside a process, the prior task can interrupt the lower priority task and can be performed first.

POSIX modified the concept of a process. Now a process is an execution unit which has its own resources during the execution. With this concept, a process is defined as an address space with one or more threads. A thread can be executed from which-ever processor that is available, in a multiprocessor system. In this way is can be reached a parallel execution of a multi-threaded process. Threads exist within a process, and they share the resources of the process. Threads are light-weight compared to a process and hence can be scheduled more efficiently than processes.

The programmer should know the parallel algorithm of the application, which should be the parallel threads that will be executed simultaneously. In the same time, the programmer should think about the probable race conditions, which should not exist between parallel threads when they share same data. The programmer has to decide how the threads will be scheduled by setting also their priority. All these issues about the thread management should be designed before. That's why the model of threads is necessary for a programmer.

There are several programming models for threads. On UNIX platforms, the predominant model is IEEE POSIX threads (Pthreads) model; the international standard: ISO/IEC 9945-1 incorporates this standard. The Pthreads programming model is very comprehensive. It provides a set of Application Programming Interfaces (APIs) which are typically supported by a user level library. Applications written to these APIs obtain source level portability on all conforming platforms. POSIX defines bindings for C language; as of this writing, there is no Pthreads standard with FORTRAN bindings and thus the most expedient approach to using Pthreads with FORTRAN is to have wrappers in C for calls to the Pthreads library. IBM provides an API for multi-threaded

programming in FORTRAN, which is not a POSIX standard but bears a strong resemblance to the Pthreads standard.

CONCLUSION

Comparing these two ways of parallelism, I can conclude that:

-MPI should be used when it's needed a portable parallel code. Using MPI it is archived high performance in parallel programming, e.g. when writing parallel libraries. It handles a problem that involves irregular or dynamic data relationships that do not fit well into the "data-parallel" model.

-MPI is not really needed in such cases when any other pre-existing library of parallel routines can be used. They might be using MPI codes too. And of course, MPI is better not to be used when it's not needed parallelism at all!

- In a hybrid version, which uses both MPI and Pthreads in SMP architectures, the role of Pthreads is to increase the performance. In such cases, MPI is used for the communications between the nodes.

REFERENCES

[1] Introduction to parallel computing (2003) A.Grama, A.Gupta, G.Karypis, V.Kumar

[2] MPI The complete reference (1996) , M.Snir, S.Otto, S.H.Lederman, D.Walker, J.Dongarra

[3] B. Lewis, D.J. Berg, (2008), Pthreads Primer.