A STUDY OF THE MESSAGE PASSING INTERFACE FOR BUILDING EFFICIENT PARALLEL ALGORITHMS

Rajesh Pydipati
Miguel Arnedo
Abstract

The present document explains the main goals that have been achieved in the Parallel Computer Architecture project. The project was based on Message Passing Interface (MPI), which is widely used in the parallel computing industry for communication within a wide range of parallel architectures. The project laid special emphasis in the study of performance of various algorithms when parallelized and run on multiple processors.
Objective:

The objective of our project has been to obtain an in-depth understanding of the Message Passing Interface (MPI), which is widely used in the field of parallel computing. The emphasis has been to show how this approach is more efficient than the traditional sequential programming for a single processor. As part of our project we have gained a thorough understanding of this interface and studied various ways and means by which we can effectively reduce the execution time of various algorithms by parallelizing them.

We have learned how to program, compile programs and execute them under MPI. We implemented several small programs to test the initial knowledge gained from the material. In order to evaluate the communication performance of several important MPI functions a benchmark was chosen and run on the alpha-cluster in HCS lab.

In addition, we have had a chance to look at various applications which can significantly benefit from MPI for building more efficient and computationally efficient algorithms. In order to limit the scope of our project which had to be done in a reasonable amount of time we chose to concentrate on a single application and study the effect of using MPI on its speedup with regard to execution time. The chosen application was ray tracing, which is a global illumination based rendering method. A parallel ray-tracing library named TACHYON was used and many sample scenes were constructed, both on a single processor and a group of processors and ray tracing times were recorded.

Finally, by doing this project we have explored the usefulness of MPI and to measure how parallel processing gives a better performance than the traditional sequential programming on a single processor.

To be more specific our focus has been:

- To get familiar with the basics of the MPI by programming different algorithms.
- MPI advanced programming. Collective operations and Virtual topologies have been included in the project.
- To obtain better performance of different algorithms, from simple to quite complex ones by using MPI. Some of them have been programmed, and a few of them have been collected and modified.
- To observe the communication performance of various MPI functions.
- To observe effectiveness of MPI for a chosen application of reasonable complexity.
- To obtain conclusions about our study of performance, based on the results obtained.
MPI Basics:

MPI stands for “Message Passing Interface”. It is a standard library of function calls (in C) or subroutines (in FORTRAN) that you insert into source code to perform data communication between processes. MPI allows the coordination of a program running as multiple processes in a distributed memory environment, yet is flexible enough to also be used in a shared memory system. MPI programs can be used and compiled on a wide variety of parallel computers such as the IBM SP2, the Silicon Graphics Origin 2000, or a cluster of workstations (homogenous or heterogeneous) over a network.

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

The standardization of the MPI library is one of its most powerful features. What it means is the parallel programmer can write code containing MPI subroutine and function calls that will work on ‘any’ machine on which the MPI library is installed without having to make changes in the this code. This standard is intended for use by all those who want to write portable message-passing programs in Fortran 77 and C. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines.

The standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- Bindings for Fortran 77 and C
- Environmental Management and inquiry
- Profiling interface

The attractiveness of the message-passing paradigm at least partially stems from its wide portability. Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these. In addition, shared-memory implementations are possible. The paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement this standard on a great variety of machines, including those ``machines" consisting of collections of other machines, parallel or not, connected by a communication network.

The interface is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of SPMD. Although no explicit support for threads is provided, the interface has been
designed so as not to prejudice their use. With this version of MPI no support is provided for dynamic spawning of tasks.

MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of MPI will be provided on such machines. At the same time, implementations of MPI on top of standard UNIX interprocessor communication protocols will provide portability to workstation clusters and heterogeneous networks of workstations. Several proprietary, native implementations of MPI, and a public domain, portable implementation of MPI are in progress at the time of this writing. MPI should not be used when we can achieve sufficient performance and portability using a data-parallel (e.g., High-Performance Fortran) or shared-memory approach (e.g., OpenMP, or proprietary directive-based paradigms) or when we don't need parallelism at all. The goal of the Message Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows.

- Design an application-programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran 77 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc., and provides extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendors’ platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread-safety.

Results:

We tested the MPI on a variety of small programs and a computationally intensive application of computer graphics area called Ray Tracing. We did not have any background in graphics but the suitability of this particular application for parallelization made us try this method to see how parallel processing affects the execution time.

Ray Tracing is a global illumination based rendering method. It traces rays of light from the eye back through the image plane into the scene. Then the rays are tested against all objects in the scene to determine if they intersect any objects. If the ray misses all objects, then that pixel is shaded the background color. Ray tracing handles shadows, multiple specular reflections, and texture mapping in a very easy straightforward manner.

The need for high quality rendering of three dimensional geometry and vector fields has grown tremendously in recent years. As a result, much work has been done in the design of algorithms and systems to render photo realistic images of these objects on computers. Many of the algorithms used to
generate high quality three-dimensional images require a lot of processing time to execute. Ray Tracing is so time-consuming because of the intersection calculations. Since each ray must be checked against all objects, for a naive ray tracer (with no speedup techniques) the time is proportional to the number of rays X the number of objects in the scene. Each intersection requires from a few (5-7) to many (15-20) floating point (fp) operations. Thus for a scene with 100 objects and computed with a spatial resolution of 512 x 512, assuming 10 fp operations per object test there are about 250,000 X 100 X 10 = 250,000,000 fps. This is just for the primary rays (from the eye through the image plane) with no anti-aliasing. Clearly there are computational problems with this. There are several approaches to speeding up computations:

1. Use faster machines
2. Use specialized hardware, especially parallel processors.
3. Speed up computations by using more efficient algorithms
4. Reduce the number of ray-object computations

To date, many researchers have created high efficiency graphics algorithms on sequential computers. Through this research, modern graphics algorithms now achieve peak efficiency and performance on sequential computers in use today. Although many algorithms currently in use are already highly tuned, it is possible to further reduce execution time through the use of many processors in parallel. By parallelizing the rendering process, execution time can be reduced by more than two orders of magnitude, given appropriate computational resources.

Ray tracing efficiency schemes can drastically improve the execution time of ray tracing software, but they have limitations and may perform poorly under adverse conditions. When algorithmic efficiency schemes have reached their limits and are no longer able to provide necessary increases in performance, the only remaining option is to apply more processing power to the problem. The judicious application of parallel processing techniques to ray tracing can dramatically increase performance while retaining its elegance. Ray tracing is especially well suited for parallelization. Each pixel in the ray-traced image can be calculated independently of the rest of the pixels in the image. This property is known as data parallelism. Ray tracing is often placed in the category of algorithms that are sometimes referred to as embarrassingly parallel. Although naive ray tracing systems are trivial to implement in parallel, there are significant challenges involved in implementing an efficient photo realistic parallel ray tracing system. Many of the algorithmic efficiency schemes developed to increase ray tracing performance work against the scalability of a parallel ray tracing system. Load balance, and constraints placed on the accessibility of data in shared and/or distributed memory present problems that must be overcome in a high performance ray tracing system.

To achieve our objective of rendering 3D images we used the parallel ray-tracing library called TACHYON, which is used on distributed memory parallel computers, shared memory computers, and clusters of workstations. Tachyon supports MPI for distributed memory parallel computers, threads for shared memory machines, and can support both simultaneously for clusters of shared memory machines. It uses MPI for message passing. We constructed various images by specifying some preset parameters in the input file. First the image was rendered on a single computer and time for ray tracing is measured. Later, the same image was rendered on a variable number of processors and execution time is measured. This process is repeated for several scenes ranging from simple to complex involving thousands of objects. Some interesting conclusions are drawn and analysis is done on the obtained results.

The table below shows the number of objects involved in each of the constructed scenes.
The table below shows the number of objects in different scenes:

<table>
<thead>
<tr>
<th>Scene</th>
<th>Number of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teapot</td>
<td>2330</td>
</tr>
<tr>
<td>820spheres</td>
<td>823</td>
</tr>
<tr>
<td>2balls</td>
<td>6</td>
</tr>
<tr>
<td>Balls</td>
<td>7385</td>
</tr>
<tr>
<td>Room</td>
<td>28</td>
</tr>
<tr>
<td>Island</td>
<td>50565</td>
</tr>
<tr>
<td>Lattice</td>
<td>2679</td>
</tr>
<tr>
<td>Model3</td>
<td>2318</td>
</tr>
<tr>
<td>Model6</td>
<td>51206</td>
</tr>
</tbody>
</table>

The graph between the run time and number of processors is as shown below.

The graph shows that with the use of MPI the execution times for rendering images falls as number of processors is increased. Also worthwhile to observe is the difference in the execution times of various scenes based on number of objects contained. The scene with large number of objects takes more time to be ray traced rather than the one with less number of objects. For images with huge number of objects Parallelization of the ray-tracing algorithm is very efficient as it significantly reduces the execution time.
Conclusion:

After completion of this project, we got a reasonable idea of how parallelization of programs is done. We got familiar with MPI and tested its efficacy on a sample application.

Acknowledgement:

We express our appreciation to Mr. John E. Stone who is the developer of Tachyon, the parallel ray tracing library that we used in our experiments.