An Analysis of Object Orientated Methodologies in a Parallel Computing Environment

Travis Frisinger
Computer Science Department
University of Wisconsin-Eau Claire
Eau Claire, WI 54702
frisintm@uwec.edu

Abstract

By examining several of the different methods for Object Oriented (OO) programming in a parallel environment, we are able to make a recommendation as to which is the most efficient method. We are also able to recommend whether procedural programming is more efficient than OO programming in a parallel environment. There are three object methodologies that will be explored, which are: MPICH, OOMPI and CHARM++. These three methodologies will be compared based on how much memory, CPU time and bandwidth each uses. These factors are also used to compare procedural programming to OO programming in a parallel environment.

Introduction

The choice of methodology for parallel program design and implementation can have a profound impact on the application. There are many factors that contribute to a parallel application's performance. The use of Message Passing Interface (MPI) is perhaps one of the most important decisions that can be made. The process does not stop there; one must chose which implementation of MPI to use and whether or not to use a procedural or OO programming approach. By examining three parallel methodologies, MPICH, OOMPI, and CHARM++ we will make a determination as to which is the most efficient. We will also be able to determine if an OO implementation or procedural implementation is better suited to parallel application design. To solve the above issues the following factors will be examined in relation to each methodology: CPU time, memory and bandwidth used by each. To illustrate this difference a simple test application that is fairly easy to port for each language and methodology was created. The effects of scaling the processes beyond the number of nodes will also be examined to see how each deals with this type of stress.

MPICH C

MPICH is one of the most popular implementations of the MPI standard in use today. It currently supports bindings for FORTRAN, C and C++ [1]. This MPI implementation was used for both the procedural and OO application.

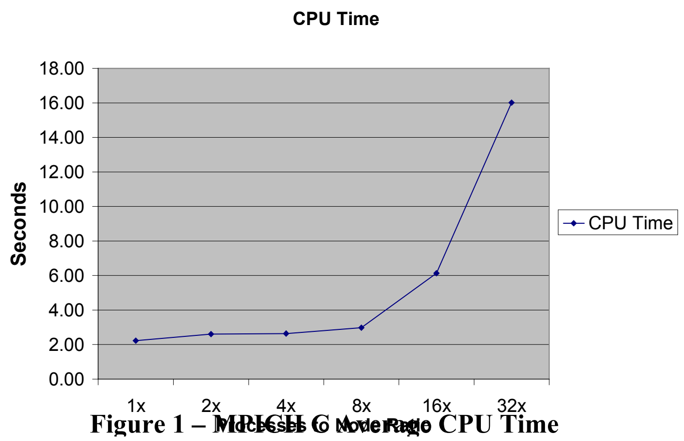

Figure 1 – MPICH C Average CPU Time

Figure 1 illustrates the average CPU time for the test application, a prime number finding application, as we scale the number of processes well beyond the number of nodes in the cluster. The CPU time for the C application up to about the eight processes to node ratio is about two seconds. After that point the CPU time rises sharply. This may be due to the fact that the network has become saturated with packets; further investigation is needed to verify this fact. If this is the case it will help explain how MPICH deals with network issues – by preventing the more finite resource from becoming scarce and causing an even greater increase in the CPU runtime.

Figure 2 illustrates the average system memory usage for each node of the cluster while executing the test application. System memory usage remains fairly constant around 250,000 Kbytes for all process to node ratios. One would believe that as the processes to node ratio increases that the amount of memory would increase.

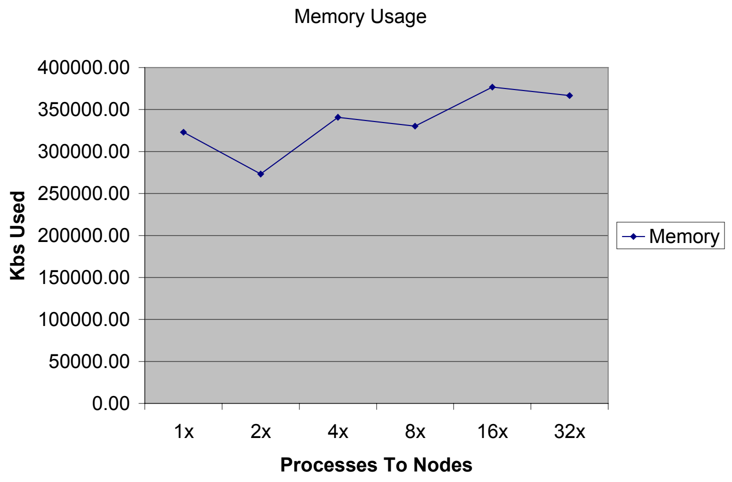

Figure 2 – MPICH C Average Memory Usage

Figure 3 illustrates the average bandwidth used by the cluster. A very interesting result of this is the fact that as the ratio of processes to nodes increases the amount of bandwidth decreases at an almost linear rate. One possible reason for the fact that the bandwidth usage drops as the ratio of processes to nodes increases is the fact that MPICH may deal with network saturation by decreasing the volume of the currently executing process. This causes a rise in runtime and a decrease in bandwidth usage and might be an attempt to balance the use of available resources to the system.

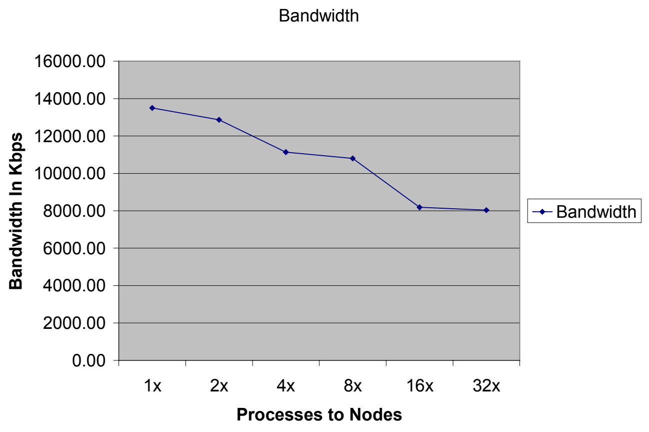

Figure 3 – MPICH C Average Bandwidth Usage

MPICH C++

MPICH also supports C++ bindings for its version of MPI. One of the questions posed is whether or not procedural applications or OO applications function best in a parallel environment. MIPCH was chosen to evaluate this question because of its support for both types of bindings.

Figure 4 illustrates the average CPU time for the test application using MPICH. This graph illustrates the same behavior that the MPICH C application illustrates. A fairly constant runtime of about 2 seconds appears up until the ratio of processes to nodes reaches an eight to one ratio. The runtime for the OO application is slightly less then that of the procedural C application. This comes as a something of a surprise, as one would believe that the OO application would take a larger amount of time to run, due to the increase in overhead caused by using objects. One possible reason for the slight decease in runtime may be due to the use of reference passing instead of direct object passing. More exploration of the MPICH C++ bindings would be needed to confirm this hypothesis.

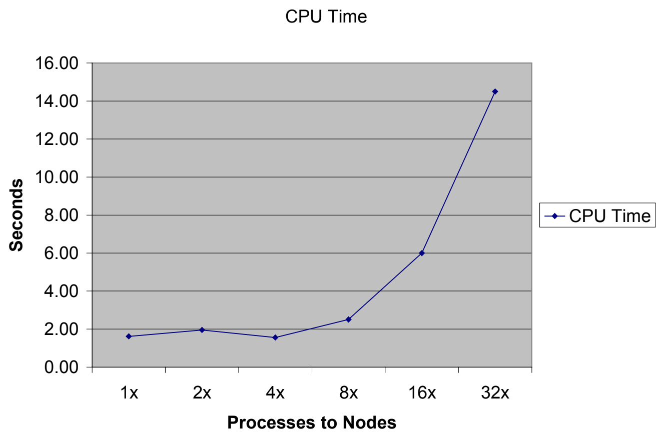

Figure 4 – MPICH C++ Average CPU Time

Figure 5 illustrates the average amount of memory used by the MPICH C++ application. This graph differs from the MPICH C application's graph by the fact that memory usage increases as the ratio between processes to nodes increases. This is expected behavior since objects are used instead of primitives. One would expect the amount of memory used to increase as the amount of processes increased, because objects consume more memory than primitives. The average amount of memory used by the MPICH C++ application is around 400,000 Kbytes or almost double the amount used by the procedural MPICH C application.

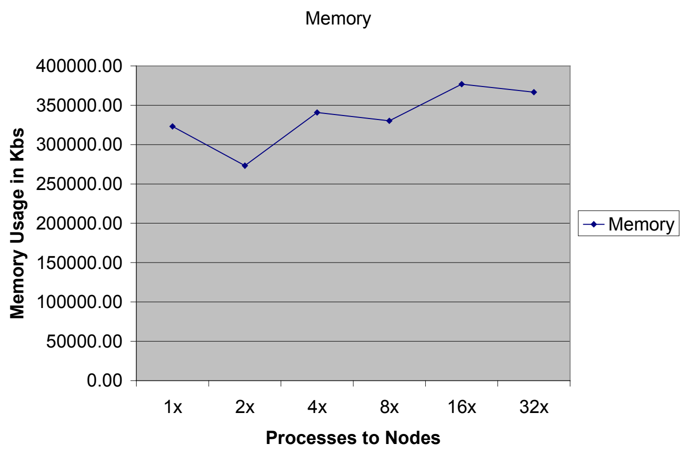

Figure 5 – MPICH C++ Average Memory Usage

Figure 6 illustrates the average amount of bandwidth used by the MPICH C++ application. The bandwidth decreases as the ratio of processes to nodes increases. The point at which it starts to take the sharpest dip is at the eight processes to node ratio. This is also where the CPU runtimes start to shoot up. Once again the increase in CPU time and decrease in bandwidth might be a type of resource conservation; an attempt to conserve the most finite resource, network bandwidth. This is at the cost of a more bountiful resource CPU time allowing the most efficient use of system resources. If too many processes were executing then network bandwidth would become saturated causing a large hit on overall application performance.

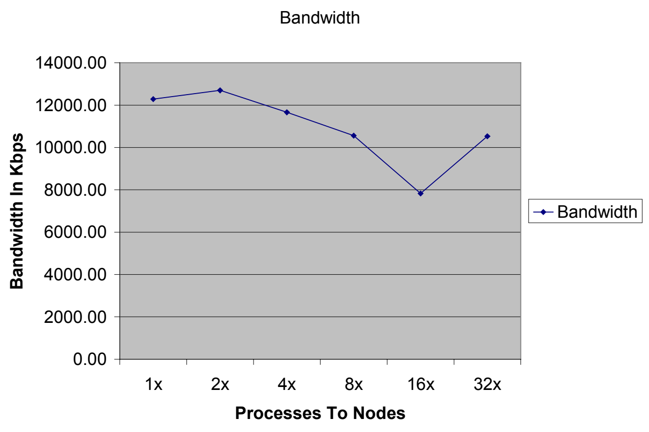

Figure 6 – MPICH C++ Average Bandwidth Usage

OOMPI

OOMPI was formed when the MPI consortium was considering the creation of C++ bindings for the MPI standard [2]. They wanted to test the feasibility of creating native C++ bindings vs. the creation of C++ bindings on top of C bindings. The current OOMPI implementation is a thin client on top of MPICH C bindings [3]. Due to this fact one would expect the overhead, even though it is small, to possibly affect the performance of OOMPI in comparison to the other methodologies.

The fact that the current OOMPI implementation is a thin client over C bindings appears to have no effect on the efficiency of the OOMPI C++ application. The CPU runtimes are almost identical to that of MPICH, the MPI implementation that OOMPI is running on top of. This graph like all of those preceding it shows a sharp increase in runtime at the eight processes to a single node ratio. This is expected behavior for OOMPI due to the fact that the C bindings for MPICH cause the same effect and the overhead involved in using MPICH under OOMPI does not seem to affect the performance of OOMPI.

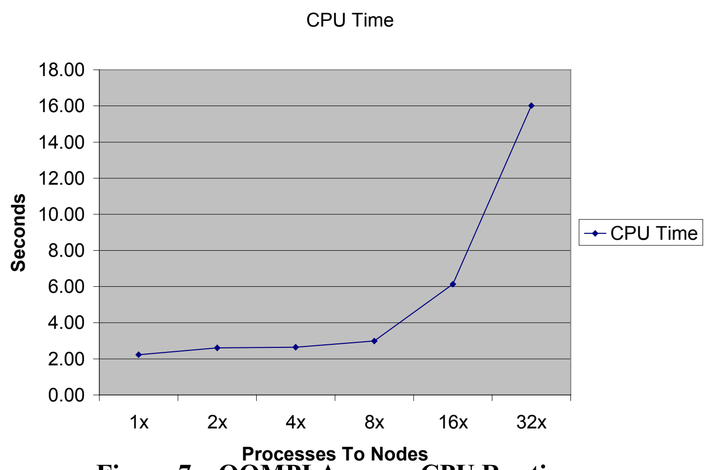

Figure 7 – OOMPI Average CPU Runtime

The average amount of memory usage for OOMPI follows a distribution similar to that of the MPICH C bindings. The average amount of memory used by OOMPI is around 350,000Kbs or about 100,000Kbs more than the MPICH C application. This increase is to be expected since we are using objects and not just primitives. The interesting fact here is that the MPICH C++ application uses almost twice the amount of memory than the MPICH C application. It can be suggested that the creators of OOMPI have optimized the C++ bindings in OOMPI to make the most efficient use of the MPI C bindings, which OMPI relies on.

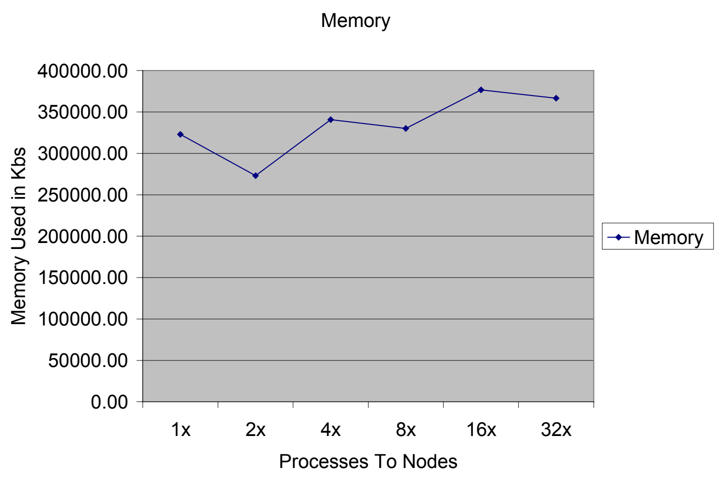

Figure 8 – OOMPI Average Memory Usage

The bandwidth usage for OOMPI follows the same pattern that all the other OO methodologies have exhibited. A decline around at the ratio of eight processes to nodes, only in this case the amount of bandwidth used increases sharply at the ratio of 32 processes to nodes. The behavior shown in Figure 9 may be a result of OOMPI attempting to balance the difference between network bandwidth and CPU time. OOMPI appears to have a more developed algorithm for this behavior. This is the first graph that does not continually fall off after the ratio of eight processes per node. Instead OOMPI attempts to find a middle ground for network bandwidth and CPU runtime usage. OOMPI finishes about 10% quicker at the 32 processes to nodes ratio then the MPICH C application did implying a slight performance increase. This might be due to the increased use of network bandwidth causing a more efficient use of the available resources in the system

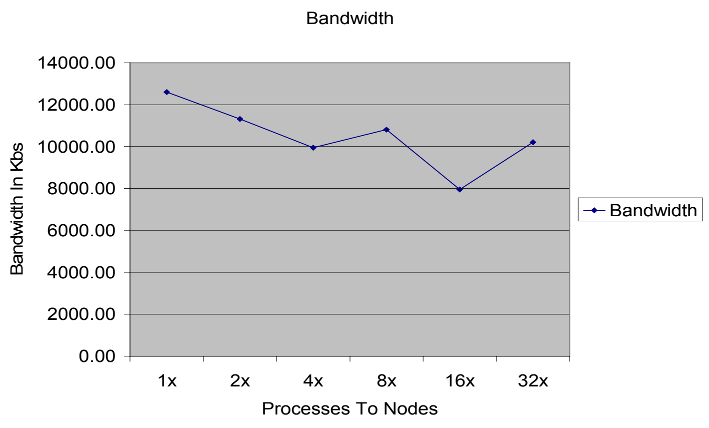

Figure 9 – OOMPI Average Bandwidth Usage

CHARM++

Charm++ is a hybrid of C++ and the MPI standard. CHARM++ was created at the University of Illinois – Urbana-Champaign in an attempt to increase programmer efficiency [4]. One would believe that this marriage of the two would yield a very well rounded product. Unfortunately this does not appear to be the case.

Figure 10 illustrates the average amount of time the test application took to execute using the CHARM++ methodology. One of the largest surprises here is the fact that the quickest run time was around 20 seconds at the one to one ratio of processes to nodes. This is slower then the 32 process to one node ratio of all other methodologies.

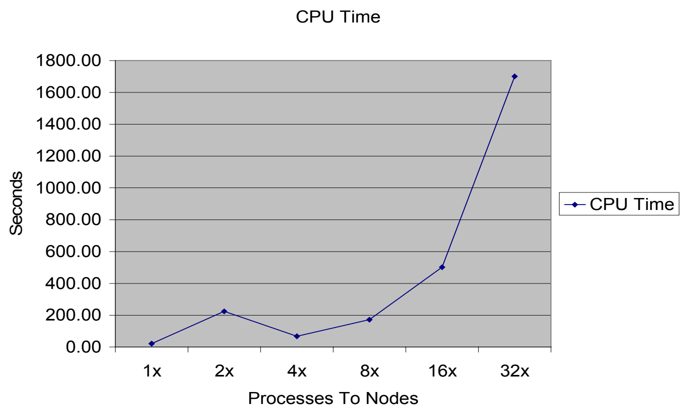

Figure 10 – CHARM++ Average CPU Runtime

One reason for the poor performance maybe due to the fact that CHARM++ is run on top of an implementation of MPI causing a large amount of overhead in creating the communication between the layers. Further investigation into the underpinnings of CHARM++ is needed to determine a more accurate reason. One way to test the existence of overhead would be to compile and test the application where CHARM++ was configured to use pure TCP/IP rather then the MPI standard if the results improved greatly this would imply that MPI was to blame.

The memory usage for CHARM++ was slightly more than that of either of the other two OO trials. This might be due to the fact that CHARM++ is a layer on top of the MPI standard rather than a thin client. It appears to be more robust methodology; this might be due to the fact that the C++ language has been modified to merge both C++ and MPI into one implementation.

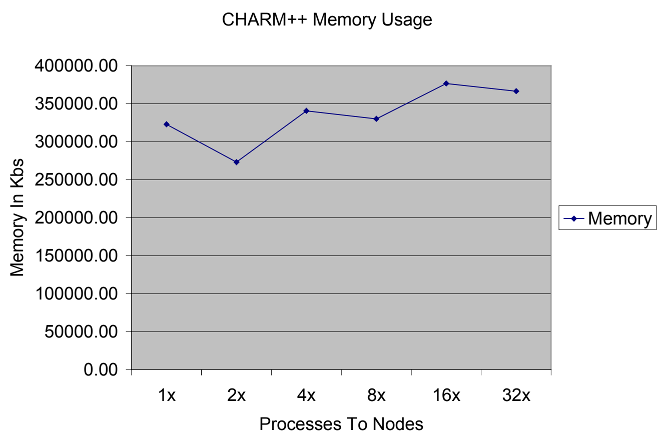

Figure 11 – CHARM++ Average Memory Usage

One very interesting observation about CHARM++'s bandwidth usage is that is continues to trail off after the two process to one node ratio. One side note is that the two to one level is also the location of the largest increase of runtime from the previous level. One possible reason for this apparent inefficient use of network bandwidth might be due to a poor balancing strategy in regard to CPU time and network bandwidth. CHARM++ may not have the ability to fully optimize the algorithm used to do balancing of system resources.

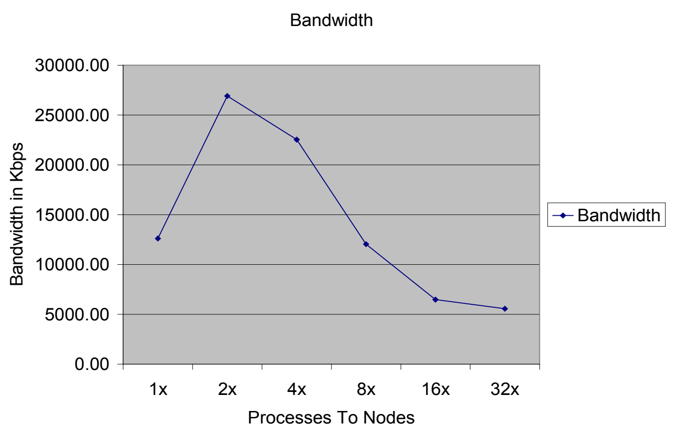

Figure 12 – CHARM++ Average Bandwidth Usage

Summary

With the exception of CHARM++, the OO methodologies appear to outperform the MPICH C bindings. The CPU runtimes of the OO application are as good if not better then those of a procedural application. They do degrade in memory and bandwidth usage. However, the added cost of memory and network bandwidth does not affect the scaling of performance.

Based on the research data it would appear that OO is a better choice for parallel application design. However, CHARM++'s data shows signs of being a very poor choice for developing parallel applications. More research is needed to determine if this is really the case or just an accident due to a small data set. In the choice between OO methodologies either MPICH C++ or OOMPI both appear to work as efficiently as the other. There is very little difference between them, with the exception that OOMPI appears to handle network bandwidth allocation better. Given the lack of object overhead in a procedural language it would seem that a procedural language would run more efficiently. As far as allocation of system resources is concerned procedural C wins out requiring far less memory (anywhere from about thirty percent to half as much then the OO methodologies.)

The surprising fact about this research is that the OO methodologies are more efficient with respect to CPU time, showing solid performance results as the process to node ratio increases.

References

[1] *MPICH - A Portable MPI Implementation*, http://www-unix.mcs.anl.gov/mpi/mpich/
[2] Peter S. Pacheco, *Parallel Programming with MPI,* Morgan Kaufmann (1997)
[3] Indiana University, *OOMPI*, http://www.osl.iu.edu/research/oompi/
[4] University of Illinois at Urbana-Champaign, *Parallel Programming Laboratory*, http://charm.cs.uiuc.edu