Mono versus .Net: A Comparative Study of Performance for Distributed Processing

Gregory J Blajian
*Roger Eggen
Computer and Information Sciences
University of North Florida
Jacksonville, FL USA
Phone 904.620.1326
Fax 904.620.2988
ree@unf.edu

Maurice Eggen
Gerald Pitts
Department of Computer Science
Trinity University
San Antonio, TX USA
Phone 210.999.7487
Fax 210.999.7477
meggen@cs.trinity.edu

Abstract: *Microsoft has released .NET, a platform dependent standard for the C# programming language. Sponsored by Ximian/Novell, Mono, the open source development platform based on the .NET framework, has been developed to be a platform independent version of the C# programming environment. While .NET is platform dependent, Mono allows developers to build Linux and cross-platform applications. Mono's .NET implementation is based on the ECMA standards for C#. This paper examines both of these programming environments with the goal of evaluating the performance characteristics of each. Testing is done with various algorithms. We also assess the trade-offs associated with using a cross-platform versus a platform dependent implementation.*

Keywords: distributed processing, algorithm efficiency, native code

1 Introduction

This research compares the relative efficiency of Mono, [1] a cross-platform implementation of C# with Microsoft's .Net environment. [2] Microsoft introduced .Net and the Common Language Runtime, CLR, as a runtime environment similar to Sun's Java Runtime Environment, JRE. [3] CLR is an environment that accepts an intermediate language, MSIL, which is interpreted at execution time similar to the byte code used by JRE.

While both languages use an intermediate language, there are some distinct differences between Microsoft's CLR and Sun's Java Virtual Machine, JVM. In Microsoft's implementation, the MSIL is actually compiled to managed native code on the first execution so that subsequent executions are at native code speed. At a higher level, many languages have compilers that target MSIL for execution in the CLR whereas the JRE is essentially a Java only environment. This paper compares the cross-platform CLR implementation, Mono, with Microsoft's CLR as applied to parallel distributed computing.

2 Fundamentals

As with any application performance study, a reasonable set of performance criteria must be established in the beginning. This involves the selection of hardware platform, the operating systems to be used and the performance measuring code requirements. The hardware selection was simply based on reasonable availability. The research facility is a cluster of six Dell personal computers. [4] Each personal computer had an Intel Pentium III 550MHz processor, the same motherboards, BIOS's, the same built in Network Interface Cards, and either 128MB of RAM or 256MB of RAM. The study was not meant to test memory management or memory performance so the differences in available memory would does not impact the results in any significant or measurable way.

The next choice was that of operating systems. The choice of Microsoft Windows ™ XP Professional [5] with Service Pack 2 seemed the appropriate choice for one of the two sets of tests since it is the incumbent operating system on which the first implementation of the CLR runs and for which the CLR was designed. The expectation was that the Microsoft implementation of the CLR should easily outperform any CLR implemented by a competitor. The choice of alternative operating systems was Fedora Core 4 Test 3. [6] Arguably, this choice may not have been the best of those Linux operating systems available, but the choice seemed quite reasonable given the strong ties to Redhat's previous commercial offerings. [7] The choice of a linux operating system was not expected to impose any penalties on the study due to the performance characteristics of the operating system. After all, the operating systems would be run on the identical hardware and were expected to perform in a similar manner regardless of the operating system installed.

A variety of algorithms exhibiting varied performance characteristics were chosen to challenge the distributed capabilities of each environment. Using data sets of varying size and algorithms requiring different execution times tests communication as well as runtime efficiency. A bubble sort algorithm, $O(n^2)$, requires more processor time and correspondingly less communication time. The second algorithm was chosen to more evenly align compute and data transfer times. A heap sort algorithm running in $O(n \log_2 n)$ was used. The last algorithm is a simple exhaustive search algorithm, O(n) that would place more burden on the network performance than on processors performance.

3 $O(n^2)$

If an $O(n^2)$ algorithm receives double the amount of data then execution time increases four times, thus the notation $O(n^2)$. In a distributed environment, $O(n^2)$ algorithm will take more execution time when the problem size, n, grows larger; however, if the problem is split either to execute in separate threads on a symmetric multiprocessor (SMP) or to execute as a thread on separate machines, the problem no longer grows in a strictly $O(n^2)$ manner. For instance, let us assume a problem of size n is run in a single thread on a singe machine; the execution time will take $O(n^2)$. Now assume the same problem set is divided by k processors, where each subset is distributed, and the same algorithm is applied to each subset. One can easily see that the time is now $k * (n/k)^2$ plus the time to distribute and to reconstitute the full resulting set. For the purposes of the study the resulting set is reconstituted using an ordered merge sort in O(n) time. At this point it becomes difficult to determine how much performance benefit is gained simply by decreasing the size of each calculated element and how much is due to the CLR environment in which it is running. However, both environments are organized in exactly the same manner so that the relative performance characteristics are meaningful.

4 $O(n \log_2 n)$

Interestingly, an $O(n \log_2 n)$ algorithm does not benefit nearly as much from creating subsets of the problem set for a distributed environment. In fact, the contribution of the merge sort to reconstitute a distributed $O(n \log_2 n)$ sort can be significant in the overall processing time required for even reasonably sized problem sets. This must be kept in mind when viewing the results, but it does not invalidate the original assumption that the amount of time required to move the data between machines would be a reasonably close approximation of the time required to perform the heap sort on each subset of the overall problem set. $O(n \log_2 n)$ to sort dominates O(n) to merge.

5 O(n)

The exhaustive search algorithm, O(n), has a somewhat interesting problem when in a distributed environment. The problem set is only required to move one way and the result set is expected to be significantly smaller than the outbound set. Like the $O(n \log_2 n)$ algorithm, this does not impact the results or invalidate the initial assumption that more time would is required to marshal and un-marshal the problem set across the network than is required to perform the algorithm.

6 Results

The machines used in this study were configured in the fashion indicated in Figure 6.1. Those objects marked S indicate the Servers, the one marked C is the Client machine and the one marked Sw is the 10/100Mbps switch. All the NICs in the machines were 100Mbps and operated at this

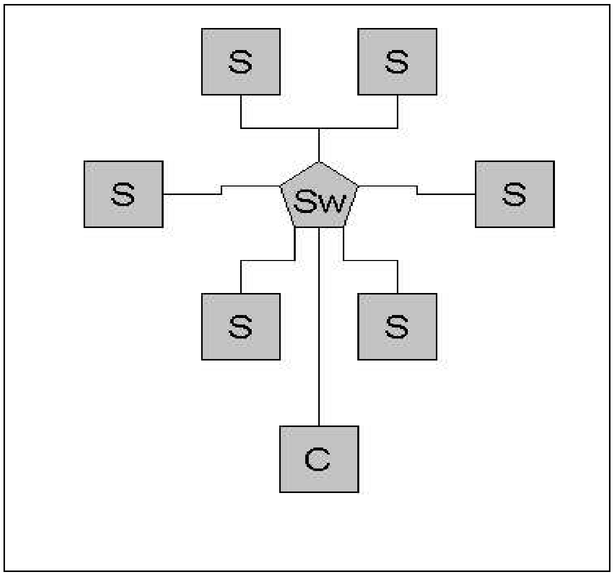

Figure 6.1

speed for both sets of tests. The client machine shared the same hardware characteristics as the servers.

6.1 The Mono Results

The first sets of tests to be conducted were the Linux tests using Mono. Fedora Core 4 Test 3 operating system was on each server and the client in the default package mode for a personal desktop. Next, the Mono CLR environment, version 1.1.7.1, was installed on all seven machines. Figure 6.1.2 shows the results obtained running the n^2 algorithm in the Mono CLR on the Fedora Core 4 Test 3 machines.

One may note that even as the problem set size gets to the reasonably large value of 131072 integers, all of the problem sizes exhibit negligible increases in time required for 6 or more threads. It is only when the larger array sizes must be sorted with 3 or fewer threads that the expected n^2 behavior becomes clearly evident.

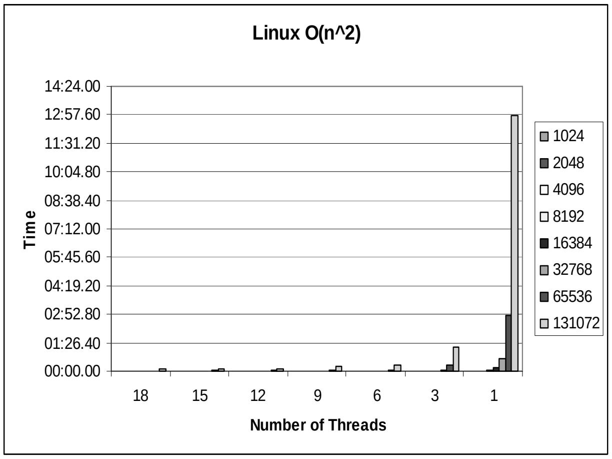

Figure 6.1.2

Figure 6.1.3 contains the results for the same environment using the O(n $\log_2$ n) algorithm.

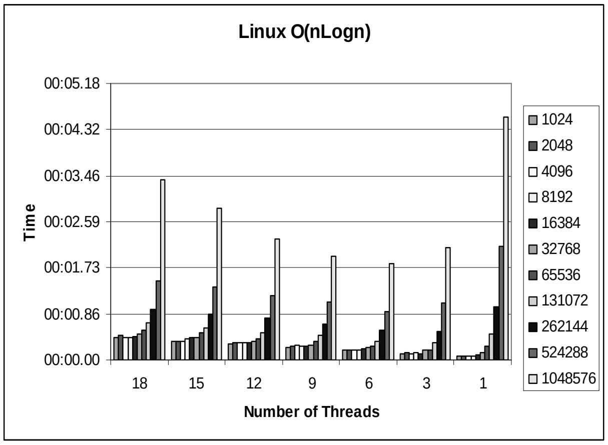

Figure 6.1.3

The O(n $\log_2$ n) graph shows a somewhat expected behavior as a result of the merge sort combined with the network and algorithm processing requirements for each machine. Note that the times required actually decrease as the number of threads approaches the number of physical machines and that for the smaller array sizes even beyond that point. The smaller array sizes actually suffer a significant amount in time lost due to the splitting and reconstitution of the array. However, as the array size approaches a significant value the benefit of the distributed model begins to manifest itself.

Figure 6.1.4 shows the results for the same environment using the O(n) algorithm. It is certainly not to be unexpected that the O(n)

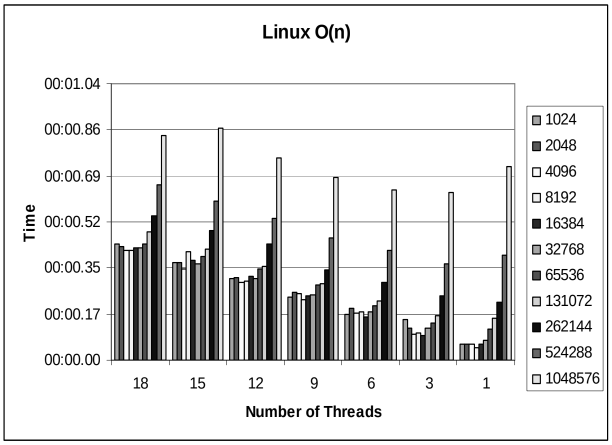

Figure 6.1.4

algorithm will display near linear behavior in a negative direction since one would not expect to gain any benefit from either splitting an array up for O(n) processing via threads or via threads in combination with a distributed system. That being said, some benefit may be gained if the array size is large enough that the time required to process each subset of the array begins to outweigh the cost of transferring the array across the network. This would seem to be born out for array sizes 524288 and more so for array size 1048576.

6.2 The .Net Results

As one would expect the results for the windows set of tests were expected to be significantly better than the results indicated in the Mono CLR environment and the results of the .Net CLR tests certainly seemed to bear that out at first glance. Unfortunately, things are not always what they seem. The results from the O(n^2) algorithm on the Windows operating system using the .Net CLR are illustrated in Figure 6.2.1.

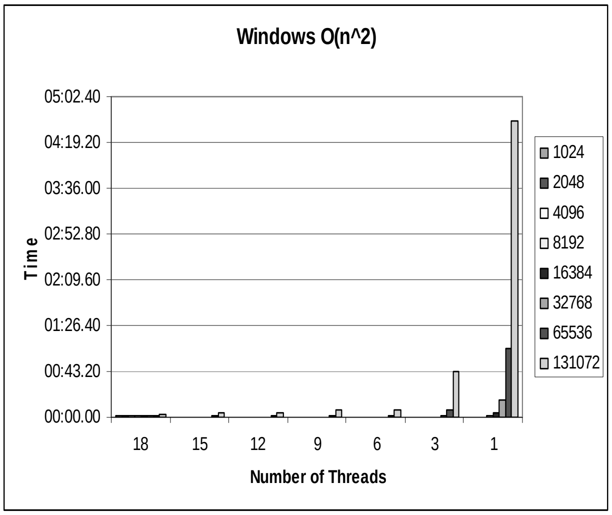

Figure 6.2.1

Figure 6.2.2 contains the results of the .Net CLR on Windows for the O(n log$_2$ n) algorithm.

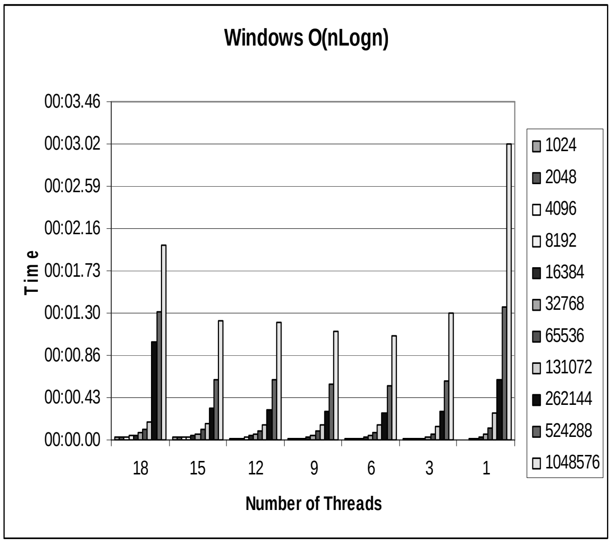

Figure 6.2.2

And lastly, Figure 6.2.3 contains the results of the O(n) algorithm.

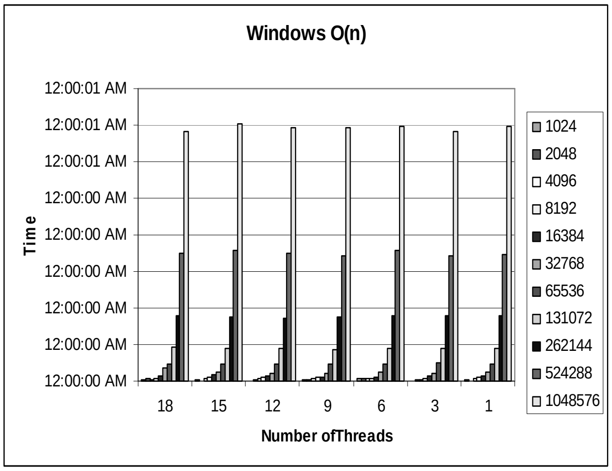

Figure 6.2.3

You may note that the results look quite a bit different when compared to the Linux results using the Mono CLR. The O(n^2) algorithm is pretty similar, but a look at the actual numbers unveils a significant difference. At this point a suspicion may be dawning that things are not as they should be for this comparison. After all the times required for the O(n) algorithm on Windows are completely linear whereas they were not on the Linux platform. The O(n $\log_2$ n) results while they are reasonably similar are again quite a bit faster. This leads to some speculation that there may be other factors at work. It is also important to note that there was more difference in the results than these graphs indicate. The Linux machines all demonstrated a much quicker propensity to go to the hard drive cache/swap space as the problem size grew which led to wild fluctuations in the results. These results were discarded as not pertinent since it was not intended for the study to validate the efficiency of the memory management, but the effect could not be entirely ignored either.

6.3 Native Code

Since the results seemed to indicate that the .Net CLR outperformed the Mono CLR written and supported by operating system vendor Novell [8] by such a wide margin the suspicion led to the native code testing of each operating system. Windows was tested using a bare O(n^2) bubble sort algorithm written in C/C++ and compiled using Microsoft Visual Studio 6 in release mode. The Fedora Core 4 Test 3 operating system was tested using the exact same code compiled using GCC, version 3.4.4, without debugging. The results are indicated in Figure 6.3.1.

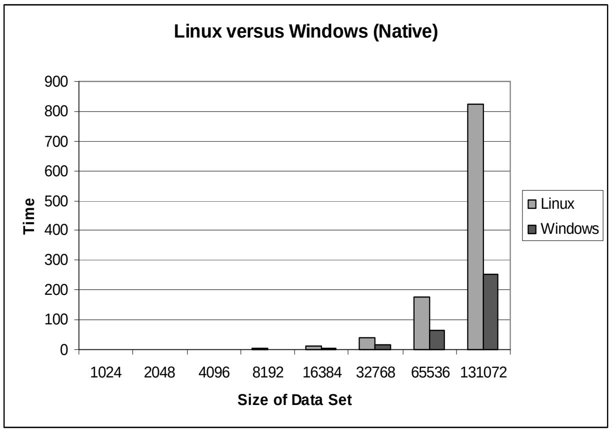

Figure 6.3.1

Please note the vast difference indicated in the two operating systems even utilizing native code. At 131072 integers in the array the Fedora Core 4 Test 3 operating system performed more than 3 times slower than the exact same code in Windows.

6.4 Combined Results

The following three figures indicate the direct comparison between .Net and Mono. In the figures, N = .Net and M = Mono. Figure 6.4.1 contains the O(n^2) results, while figure 6.4.2 contains the O(n $\log_2$ n) results and finally figure 6.4.3 contains the O(n) results. The chart indicating the direct comparison N = .Net and M = Mono.

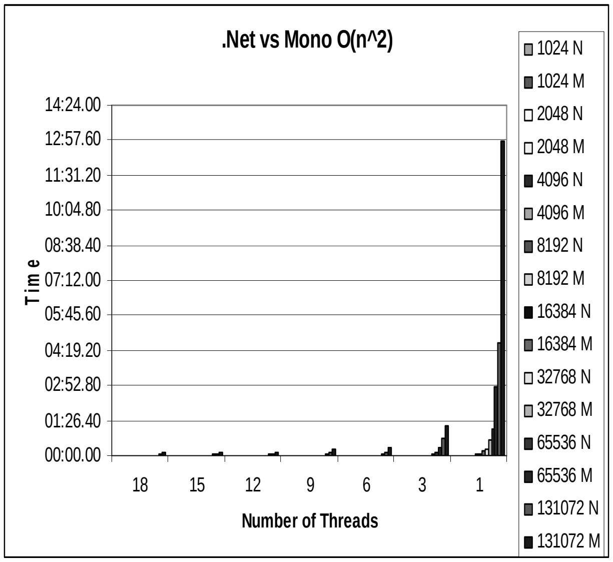

Figure 6.4.1.

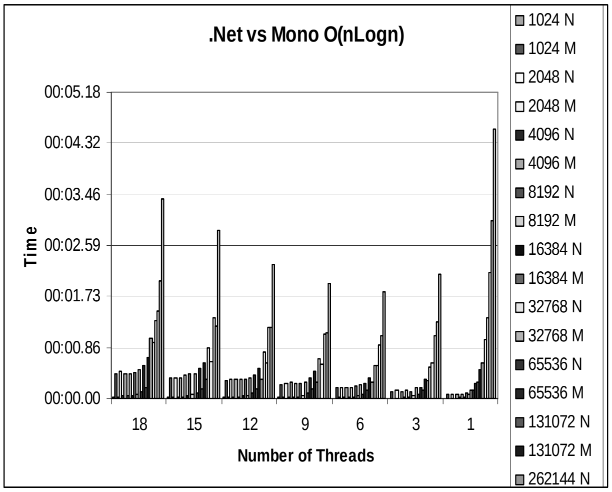

Figure 6.4.2

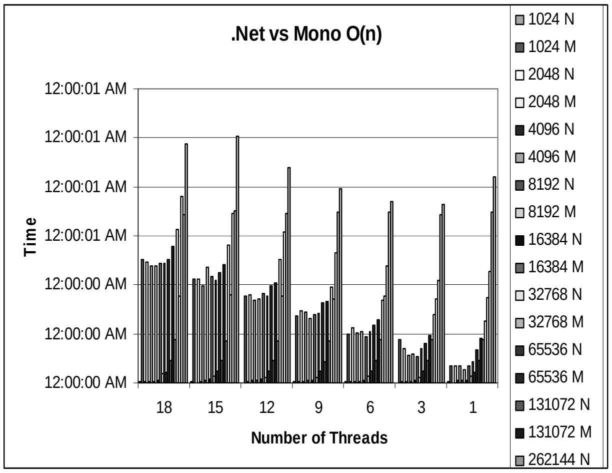

Figure 6.4.3

7 Conclusions

So where does this leave the results as a function of the study? Clearly the use of the Fedora Core 4 Test 3 operating system severely handicapped the Mono CLR as a test environment. It would seem, however, that not all is as bad as it may at first appear. Looking at the time required to run the O(n^2) algorithm in the Mono CLR with 131072 integers in the array, 774.64s, and comparing that to the length of time it took to run the same number of integers using native code, 825.14s, it would seem that the Mono CLR is indeed MORE efficient than even native code. Though the operating system handicapped the results such that truly direct comparisons seem unreasonable to take, there is certainly no reason, given the results of Mono versus Native code, to discount Mono as a viable alternative to .Net. In fact, if one was considering such a move they may get even more performance than they may otherwise expect.

If one is familiar with Java and RMI, the Java equivalent of .Net/Mono distributed processing, it is noteworthy to point out that the Mono/.Net was easier to implement than the same functionality in Java.

8 Future Directions

It is easy to argue that these results more than justify further study. Of the free Linux operating systems available, one could try Debian, Ubuntu, Knoppix, FreeBSD, or another not listed here, possibly even one or more commercial variants of linux, such as Suse/Novell or Redhat. One should consider the direct comparison of native code speeds before pursuing CLR.

Another possible direction in this would be to compare both the Mono CLR versus .Net CLR on the Windows operating system. Mono will run on Windows, by doing so will eliminate operating system impact.

A third possible study would entail a comparison of the Mono CLR with Java. Both could be run on a Linux platform and neither would necessarily gain an operating system advantage. It may even be reasonable to expect that one could compare the two in the same clustered environment as this study did.

9 References

[1] *What is Mono?* (31 Jan 2006). Retrieved February 16, 2006, from http://www.mono-project.com/Main_Page

[2] .NET Highlights. (December 27, 2005). Retrieved February 20, 2006 from http://www.microsoft.com/net/default.mspx

[3] *Java.sun.com: The Source for Java Developers.* (2006). Retrieved February 24, 2006 from http://java.sun.com/

[4] *Dell Products and Services.* (2006). Retrieved February 1, 2006 from http://www.dell.com/

[5] *Windows XP Editions.* (226). Retrieved February 1, 2006 from http://www.microsoft.com/windowsxp/default.mspx

[6] *Fedora*. (March 20, 2006). Retrieved February 21, 2006 from http://fedora.redhat.com/

[7] *Red Hat*. (March 20, 2006). Retrieved February 21, 2006 from http://redhat.com/

[8] *Novell: Software for the Open Enterprise.* (2006). Retrieved February 19, 2006 from http://www.novell.com/