

# Efficiency of Functional Languages in Client-Server Applications

\*Dr. Maurice Eggen  
Dr. Gerald Pitts  
Alexander Starche  
Department of Computer Science  
Trinity University  
San Antonio, Texas

Dr. Roger Eggen  
Computing and Information Sciences  
University of North Florida  
Jacksonville, Florida

**Abstract:** *Functional languages offer considerable benefit to the programmer/researcher. Simple syntax, ease of programming, and availability make functional languages a desirable choice. There are many tools available to the researcher when developing client-server applications. But how do these environments compare? This paper will consider a functional language, Scheme, and compare its capabilities to procedural languages. We use C-MPI as a baseline for comparison purposes.*

**Keywords:** sockets, threads, distributed processing, parallel processing, client-server.

## 1 Introduction

Software engineers, computer professionals and researchers are seeing a remarkable increase in performance as well as decreasing cost of hardware in the computing machines available to them. This new hardware presents unique challenges to the computing professional to develop software systems that adequately take advantage of this new hardware. Much of the programming is directed toward the programming of distributed applications and parallel systems. Multi-core machines are soon going to be standard. Moreover, it is straight-forward for the computer engineer to interconnect multiple off-the-shelf machines and gain a respectable parallel system. Thus, parallel machines will be ubiquitous.

The problem becomes one for the software engineer to develop software systems which will harness the tremendous computing power now available. Since

much of the programming effort is directed toward distributed and parallel systems, many of the applications are client-server oriented applications.

This paper will discuss one aspect of the programming process, that of programming client-server systems. These applications can be represented by a master-slave parallel application, where the client serves as the master, and the servers act as slaves in the parallel system.

There are two issues associated with such systems, efficiency and ease of programming. This paper advocates the use of functional programming languages for the ease of programming such systems, while observing that the performance of such systems is nearly comparable to other programming tools.

## 2 Hardware

The hardware for these experiments consisted of a laboratory of networked

workstations. Each workstation consisted of a hyperthreaded CPU at 3.00GHz. Each of the machines contained one gigabyte of main memory and 80 gigabyte external memory. Each ran with a cache size of 2048 KB. The machines were all interconnected by gigabit fast Ethernet.

### 3 Software

The operating system for each of the aforementioned machines was Linux version 2.6.12-1.1398\_FC4smp. The MPI version was LAM MPI while the scheme software was MzScheme from the PLT group.

#### 3.1 MPI

LAM/MPI is a high-quality open-source implementation of the [Message Passing Interface](#) specification, including all of MPI-1.2 and much of MPI-2. Intended for production as well as research use, LAM/MPI includes a rich set of features for system administrators, parallel programmers, application users, and parallel computing researchers. [1]

#### 3.2 MzScheme

The Programming Language Team (PLT) consists of numerous people distributed across several different universities in the USA

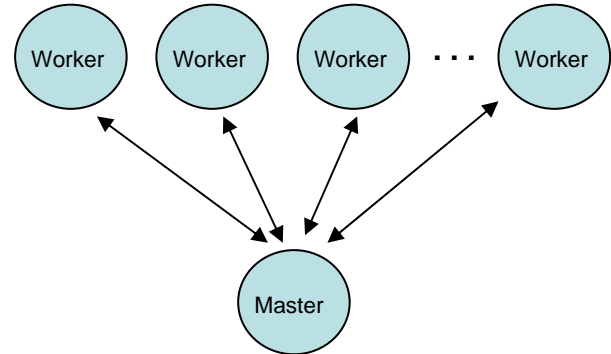
- Brown University, Providence, RI
- Northeastern University, Boston, MA
- University of Chicago, Chicago, IL
- University of Utah, Salt Lake City, UT

MzScheme is an implementation of the Scheme programming language for Windows (95 and up), Mac OS X, and UNIX. MzScheme is R<sup>5</sup>RS-compliant,

including the full numerical tower. It also provides threads (on all platforms), exceptions, modules, class-based objects, Unicode, regular-expression matching, TCP/IP, and more. [2].

### 4 Parallel and Distributed Processing

Each of the environments was programmed using a master-slave model. The master generates the data to be processed, and passes out the work to each of the workers. Figure one below displays the distributed configuration.



**Figure 1. Master-Slave Configuration**

Every time a distributed cluster of workstations cooperates in the solution to a distributed parallel application, a messaging system is involved. Here we are assuming that we do not have shared memory. While the details may vary with different programming environments, fundamentally all systems must perform the same tasks in a similar manner. We assume the parallel application is divided into a collection of processes, with the following requirements:

- the code in a process is inherently sequential
- a process must be able to send and receive messages

- the send operation should be synchronous
- the receive operation should be asynchronous
- a process should have the ability to perform dynamic task creation and allocation
- messaging should be able to be created and destroyed dynamically
- parallel processes should be able to run in their own thread of control

Moreover, when processes communicate certain things must happen, whether under the control of the programmer or handled implicitly by the programming environment. When process one communicates with process two, for example, the following must occur:

- process one must prepare a send buffer
- process one must pack the information to be sent into the send buffer
- process one initiates a send and sends the buffer
- process two receives the buffer
- process two unpacks the information from the buffer and performs the appropriate tasks
- process two must then prepare a send buffer, pack the new information into the buffer, and initiate a send.

The above must be repeated for each send-receive operation that is performed. In many cases the programming environment handles the grisly details of the above operations, but in some cases, such as the programming of native sockets, the programmer must take responsibility for the operations. Regardless of who or what handles the details, the point is that it must be done. In the C-MPI environment much of the work described above is handled by the

programming environment, but not all. In the MzScheme environment, as we shall see requires the programmer to handle some of the details of the message passing.

## 5 The Experiment

To evaluate software systems for performance and programming efficiency it is important that the experiment performed in some sense represents a real world application. Thus, the authors of this paper have decided to study sorting. Since sorting is ubiquitous in many applications in computer science it represents a real programming application. The programs were written in a consistent manner, as much as possible, utilizing the features of each of the programming environments. To challenge the programming hardware, an order  $O(n^2)$  algorithm as well as an order  $O(n \log_2(n))$  algorithm were chosen. The insertion sort is readily programmable in any language, and since the insertion sort has average behavior  $O(n^2)$  it, along with the quicksort which is  $O(n \log_2(n))$  provided a suitable test suite. The algorithms were tested on data sets of several sizes, and were programmed utilizing several choices of distributed configurations.

In our experiment, the master process (the client) acquires a random array. The array is subdivided into a number of subarrays equal to the number of threads (processes) available to do the processing. Each thread simultaneously sends its data to a waiting server, which receives the data, sorts it, and sends it back to the master process. The master process then merges the sorted arrays to produce a single sorted array. Since array merge can be accomplished in  $O(n)$  time it should not affect the results. It could be measured both ways, with and without the merge. The results of the study are below.

## 6 Results

Many functional languages, including scheme, are natural for parallel and distributed processing. Since the language is list based, operations on lists are natural. Moreover, if an operation is to be performed on each of the elements of a list, there is no reason that the operations can not all be performed simultaneously. In our sorting problem described above, it is natural to think of an array as a list of subarrays, each of the subarray elements of the list to be sorted simultaneously. There

are many other operations which can be envisioned in a similar way. .

We sorted varying array sizes using one thread, two threads, four threads, eight threads and 16 threads. We sorted data sets of size  $2^{10}$ ,  $2^{12}$ ,  $2^{14}$ ,  $2^{16}$ ,  $2^{18}$ ,  $2^{20}$ . While the machines used for the experiment were in a multi-user environment, times were taken at 3:00am when traffic on the machines was minimal. Moreover, the timings were taken several times, and any anomalous readings were discarded and average readings were kept. Where times were so large as to be of little use, the results were not kept. The results are summarized in the tables below.

**Table 1: C-MPI  $O(n^2)$  Timings**

	Number of Threads				
	1	2	4	8	16
1024	0.001951	0.001583	0.001285	0.001045	0.003953
4096	0.030620	0.009992	0.004152	0.003863	0.005762
16384	0.491108	0.127155	0.039502	0.016371	0.009935
65536	7.859865	2.896729	0.520549	0.362670	0.268322
262144	127.707181	61.661990	14.109480	2.126443	0.779449
1048576	2084.721181	1032.409042	253.515024	60.988970	13.504822

**Table 2: C-MPI  $O(n \log_2(n))$  Timings**

	Number of Threads				
	1	2	4	8	16
1024	0.000204	0.001145	0.001043	0.002782	0.004007
4096	0.000785	0.002664	0.002494	0.003686	0.005444
16384	0.003227	0.008073	0.009316	0.009500	0.008588
65536	0.017259	0.030543	0.036278	0.038091	0.274978
262144	0.066520	0.119782	0.142485	0.162853	0.329270
1048576	0.276221	0.497360	0.592908	0.651315	0.687051
4194304	1.199954	2.007406	2.444802	2.623541	2.739232
33554432	10.760394	19.350707	19.613282	21.171894	21.992592

**Table 3: Threaded Scheme  $O(n^2)$  Timings**

	Number of Threads				
	1	2	4	8	16
1024	0.148755	0.062851	0.024057	0.010760	0.006796
4096	2.359424	0.636509	0.188424	0.072518	0.057016
16384	37.327755	9.344478	2.487939	0.846342	0.262002
65536	601.128855	149.927041	37.873995	12.483140	3.264589
262144	> 20 min	> 20 min	> 20 min	193.030645	72.966614

**Table 4: Threaded Scheme  $O(n \log_2(n))$  Timings**

	Number of Threads				
	1	2	4	8	16
1024	0.009180	0.017728	0.008474	0.006742	0.006412
4096	0.041833	0.051402	0.040040	0.026203	0.053859
16384	0.200423	0.198648	0.139345	0.113941	0.090788
65536	1.114900	0.838003	0.551171	0.416367	0.415118
262144	4.347228	3.320504	2.244137	1.886342	1.573712
1048576	19.572310	14.155470	9.634108	7.612682	6.472802
4194304	84.458091	61.388649	40.490998	32.687391	27.872342

## 7 Summary and Conclusions

It is interesting to note that even with  $2^{25}$  elements it was impossible to gain anything in the C-MPI environment from distributing the data, even among 16 processors using the  $O(n \log_2(n))$  algorithm. In fact, as the size of the dataset remained constant and more processors were added, the time to accomplish the work went up rather than down. In this case, the communication time dominates the computation time. However, it was certainly possible to gain processing speed using the  $O(n^2)$  algorithm in the C-MPI environment. Adding more processors helped when using the less efficient algorithm, which required sufficient processor time that the communication time did not dominate the computation time.

Results showing the threaded scheme timings slower than the C-MPI timings are

not surprising. The overhead associated with scheme, even an efficient implementation such as MZScheme, will not allow performance similar to the C-MPI programming environment .

What the researcher must now consider is whether or not the programming ease associated with the scheme environment is dramatic enough to allow its use in a distributed parallel environment. Example code for the worker method below illustrates the ease with which the functional language may be programmed. In some sense the beauty of the functional approach is that it does not require state changing operations. A listener is established, input and output ports are created, communication accepted, data input port read, sorted, and returned without a single assignment statement!

```
(define sortwork
  (lambda (port)
    (let* ((l (tcp-listen port)))
      (let-values
        (((i o) (tcp-accept l)))
        (begin
          (tcp-close l)
          (let ((ls (read i)))
            (write (sort ls) o)
            (tcp-abandon-port i)
            (tcp-abandon-port o))))))))
```

## 8 Directions for Future Study

It is the authors' opinion that considerable research effort should be invested into the production of simple high level programming libraries for the scheme and other functional language environments which will allow their use in distributed processing. If scheme were to enjoy the utility provided to C, C++ and FORTRAN by the MPI environment, then scheme, with its ease of programming, may enjoy much wider usage.

Many times, when considering real time applications (which are being attacked more and more by functional language programmers) the issue is not native speed, which certainly is a consideration, but throughput, which matters more. If a system breaks, the time to repair must be considered, and with its ease of programming functional languages must be a consideration.

## 9 References

- [1] <http://www.lam-mpi.org/>
- [2] <http://www.plt-scheme.org/software/mzscheme/>
- [3] M. L. Liu, Distributed Computing Principles and Applications, Pearson Addison-Wesley, 2004.
- [4] Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2004.
- [5] Maurice Eggen and Roger Eggen, "Efficiency of Searching Knowledge Bases for Robotic Control," Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, Volume III, June 2005, pages 1085-1091
- [6] Roger Eggen, Maurice Eggen, Sanjay Ahuja, and Paul Elliott, "Efficiency Considerations Between Common Web Applications using the Soap Protocol," Proceedings of the third IASTED International Conference on Communications, Internet and Information Technology, September 2004, pages 461-465.
- [7] Maurice Eggen and Roger Eggen, "Efficiency Considerations of PERL and Python in Distributed Processing," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, Volume I, June 2004, pages 1237-1242
- [8] Roger Eggen, Maurice Eggen and Sean Gwizdak, "SOAP and XML as Parallel Distributed Environments: An Empirical Cost/Benefit Evaluation," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, Volume II, June 2002, pages 631-637.
- [9] Roger Eggen and Maurice Eggen, "Efficiency of Distributed Parallel Processing using Java RMI, Sockets and CORBA," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications,

CSREA Press, Volume II, June 2001, pages  
888-893

[10] George Springer and Daniel Friedman,  
Scheme and the Art of Programming,  
McGraw Hill, 1989