Efficiency of Functional Languages in Client-Server Applications

*Dr. Maurice Eggen  
Dr. Gerald Pitts  
Department of Computer Science  
Trinity University  
San Antonio, Texas  
Phone 210 999 7487  
Fax 210 999 7477  
meggen@cs.trinity.edu

Dr. Roger Eggen  
Computing and Information Sciences  
University of North Florida  
Jacksonville, Florida  
Phone 904 620 1326  
Fax 904 620 2988  
ree@unf.edu

[Submitted to PDPTA 06]

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 R5RS-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.
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 and eight 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. The results are summarized in the tables below.
Within a week we should see our final results. Preliminary results show scheme to be somewhat slower than C-MPI, but results are yet to be finalized.

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

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!

```scheme
(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


