

Improving Performance of Distributed Haskell in Mosix Clusters

Lori Collins¹, Murray Gross¹ and P.A. Whitlock¹

Department of Computer and Information Sciences, Brooklyn College
2900 Bedford Avenue, Brooklyn, NY 11210-2889

Abstract. We present experimental results demonstrating a qualitative improvement in the performance of a Parallel Haskell implementation in a cluster environment in which PVM intermachine communication has been replaced by process migration under the control of the Mosix patches to the Linux kernel. Together with the software modifications that have been made to the Haskell run-time system, this performance improvement has implications in the area of automatic run-time optimization.

Introduction

Much progress has been recently made in the development of parallel and distributed Haskell compilers[1–4]. Haskell is a functional language[5] that exploits lazy evaluation which, in some cases, speeds up computations. We have implemented the Glasgow distributed Haskell (GDH)[6] compiler and run-time system on a cluster of computers managed by the Mosix distributed operating system[7]. Mosix mimics the behavior of a symmetric multiprocessor by allowing dynamic process migration. We expected that for parallel Haskell programs running on the cluster, the greatest efficiency would be achieved using distributed Haskell with multiple calculations executing on each real processor. Such behavior was observed by others executing conventional parallel procedural programs on the cluster[8]. However, the GdH compiler, version 5.04, distributed only one computation per processor. Investigation of the run-time environment revealed that this was due to the Parallel Virtual Machine[9] (PVM) functions called by the Haskell compiler to achieve distributions of tasks. Since we use the compiler with a distributed operating system, it is not necessary to use a message-passing library. Therefore, the Haskell run-time system was modified to remove all calls to PVM functions and extensive testing was performed. This paper describes the changes made and reports the results of the tests.

Parallel and Distributed Haskell

Functional languages' main dissimilarity to imperative languages is that they do not consist of a sequence of steps to solve a problem; instead, they translate tasks into expressions that are then evaluated. These expressions look similar to those

expressions that are found in a mathematical context, which makes the language more intuitive. In fact, functional languages have been said to offer programmers a closer semantic feel because their syntax tends to yield code very similar to formal statements of algorithms[10]; indeed, a “pidgin” Haskell is frequently used to write out such formal descriptions, which then need only minor modifications to convert them to executable code.

While functional languages provide a number of advantages over procedural languages to the programmer[11], four characteristics are particularly important in the area of parallel and distributed execution: (1) functions in functional languages are generally first class values, i.e., can be passed from one function to another in exactly the same way as ordinary data; (2) once a value is bound to a variable it cannot change (referential transparency); (3) in some functional languages, Haskell in particular, function evaluation is lazy and (4) side effects are forbidden except in the rare cases such as I/O where they cannot be avoided[12].

Now, by lazy execution we mean that computation of values is delayed until they are actually required, which, among other things, effectively eliminates the commonplace practice of pre-computing values that might (or might not!) be used in subsequent computation. This is clearly important in parallel computing, because it means that there is no need to delay parallel threads while values that will not be used are being computed. This provides an obvious and, in many cases, major potential for reducing overall (wall-clock) computing time.

In combination with lazy execution, the first-class status of functions, referential transparency and the prohibition against side effects provide potentially radical reductions in coordination delay and overhead. Once a value is computed by the first process that requires it, it may be freely used by any process in the computation without fear of change at an inappropriate time, or, indeed, any change at all. The end result is that processes can block on values only while they are currently being computed (a delay that cannot possibly be avoided), and there is no need to be concerned with whether a value is “current,” or even, for that matter, available. If it is available, it is current, and if it is not yet available, it can be computed by whichever process first requires it.

The effect, then, of the specific language characteristics we have considered above is to cause parallel Haskell programs to serialize on critical paths rather than on the programmer’s conceptual scaffolding. Since it is clearly impossible to improve performance beyond what is obtained on critical paths, optimal performance is obtained without programmer intervention, with the obvious consequence that the programmer can concentrate without distraction on underlying algorithms. The entire issue of scheduling and coordination has been abstracted out of the problem by the simply expedient of using an appropriate programming language.

Mosix Operating System

Our distributed calculations are run on a cluster that consists of 15 interconnected computers currently running the Debian Linux operating system with

OpenMosix patches[7] overlaid on the operating systems. Mosix is a cluster management system that makes a cluster of separate computers run like a symmetric multiprocessor (SMP) by using algorithms and techniques that support resource sharing by dynamic process migration[7]. It works by extending the kernel so that the nodes can cooperate and share resources. This is ideal in heterogeneous configurations, since it automatically performs load balancing on processors[13].

Most clustering software either coordinates parallel programs operating on the independent units of the cluster or facilitates communication between independently executing programs on separate processors using parallel libraries[14, 15]. In contrast, Mosix achieves its effect by physically moving images of independent processes off its "home" machine onto slaves in a manner designed to balance the load across the available CPU's. Requests for system services are trapped by the Mosix software, executed on the home machine when they cannot be performed on the slave machines and then the results are transmitted back to the slave machine in a manner that is transparent to the running processes. In effect, each process "believes" it is still running on the machine on which it was spawned. A special high-efficiency file system[13] that is part of the OpenMosix package permits processors to access files on other processors with minimal overhead¹. This means that data can be distributed across the entire cluster and there are only a few clearly defined conditions that prevent dynamic migration of tasks from one machine to another.

Modification to the GdH compiler

Two major changes were made to the GdH run-time system to optimize its performance in a Mosix environment. In the original design PVM was used for all remote process creation and communication. Rather than building a virtual machine of virtual processors provided by PVM processes, our (modified) version of the run-time system builds a virtual machine of virtual processors created by the simple expedient of forking local processes (which will be moved about to remote processors by Mosix). The `send()` and `recv()` calls in PVM were replaced by local system calls. Much discussion revolved around how individual values were to be passed between the processes. For example, with the Mosix OS, it would be possible to use multiple pipes to transmit data since Mosix would trap the writes to the pipe. However, this is not a general solution and it was decided to use an UDP-based communication. After error testing of our new GdH-BC compiler² it was necessary to test whether the hypothesized efficiencies had actually been achieved[16].

¹ It would be useful to gather statistics on how well Mosix is doing to determine its effect and efficacy with respect to automatic run-time optimization

² Changes to the Haskell compiler were performed by Dino Klein and Qing Shou in Spring, 2003. The testing of the changes were initiated in Fall, 2003 by Qing Zhou and Kerim Simsek and continued into Fall, 2004.

Performance Results

Baseline tests were needed to gain an improved understanding on how these changes have affected the system. This section describes the results from the runs of the sequential code `ffactsm_seq`, used to validate the parallel results, and the parallel code `ffactsm_par`, from the Parfact benchmark suite. The `ffactsm_seq` program was compiled with the non-parallel GHC[5] and run on a single processor. The program `ffactsm_par` was compiled by our new modified PVM-free version of GdH-BC and run on fourteen or fifteen real processors. There were twenty-six sets of arguments run three times each with varying execution times. The serial and parallel calculations gave the same answers in all cases. In addition to validating our results we have calculated, shown in figure 1, the speedup between the serial and parallel calculation. The speedup is a measure of relative performance defined as:

$$S = \frac{\text{Execution time on a single processor, best sequential algorithm}}{\text{Execution time using } p \text{ processors}} \quad (1)$$

As expected we saw a decrease in execution time from our parallel calculations in comparison to our serial execution times. However it should be noted that Mosix gathers execution time statistics before distributing a calculation. So quickly completed calculations are not distributed as extensively as longer calculations.

Fig. 1. Observed speedup, as defined by Eq. (1), when the test program was run in parallel using GdH-BC

While testing `ffactsm_par` with different numbers of processors, we decided to see what would happen if we requested more processes than processors (i.e., with different numbers of virtual processors). We varied the number of processes between 15 and 60. It was immediately qualitatively clear that there was an improvement in performance because a reduction in total execution time was obtained. A series of quantitative experiments confirmed the qualitative findings. In figure 2, the total timings from runs with a fixed number of processes are shown. In the figure, we have included the average of the three best run results to show overall the decrease we have seen when more than one process per processor is requested.

As we add processes up to a number equal to the number of processors the execution time decreases approximately linearly, as we might expect. However,

Fig. 2. Averaged total timings from parallel runs of the test program with a fixed number of processes

once there are more processes than processors the execution time continues to drop until an equilibrium point is reached and the running time begins to increase with the number of processors. We think it safe to say that the increase in running time when the number of processes increases beyond a "balance point" is the result of increases in system overhead associated with process coordination and load balancing.

Conclusions

We conclude that the original design of the GdH run-time system, which refused to permit more than one PVM session, enforces a performance barrier that need not be present. Though PVM is freely available and widely used, its technical deficiencies have sparked the development of more modern message passing library standards[15]. So, rather than a micromanagement approach to load distribution as used in GdH, we find that at least under some circumstances, a contention driven supervisory model outperforms an excessively limited load-balancing approach. We need to continue to conduct further tests on the modified run-time system using available model applications and additional model (toy) applications developed for testing and validation. We feel that Haskell is a better fit to parallel and distributed calculations than its more problematic imperative counterparts because of its inherent parallelism, which speaks to the importance of continued development of language and compiler.

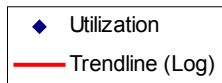
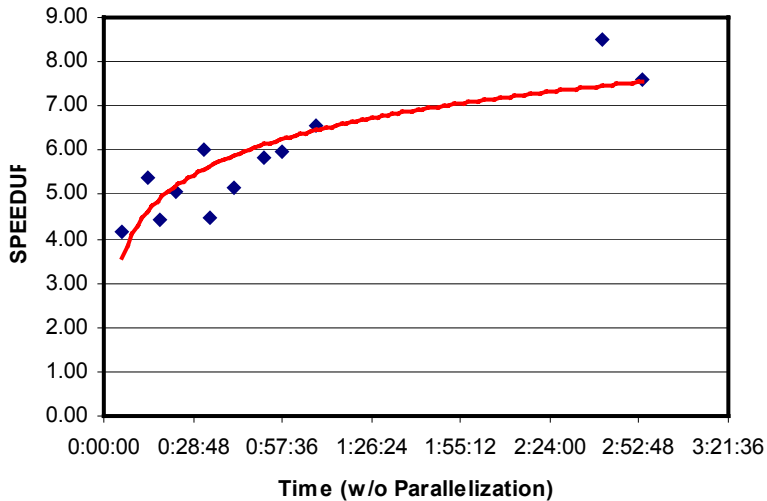
Acknowledgments

The establishment of the cluster of computers was made possible by ONR Grant N00014-96-1-1-1057. The initial development of the GdH-BC compiler was done by Dino Klein and Qing Shou. Testing of the compiler was begun by Qing Shou and Kerim Simsek. One of us, L.C., is supported by . . .

References

1. Hammond, K., Mattson, J.S, Partridge, A.S., Peyton Jones, S.L, and Trinder, P.W.: GUM: a portable parallel implementation of Haskell. Proceedings of the

- ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, Philadelphia, PA (1996) 79–88
2. Peyton Jones, S.L, Gordon, A.D, and Finne, S.O.: Concurrent Haskell. Proc ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida (1996) 295–308
 3. Trinder, P.W.: Motivation for Glasgow distributed Haskell, a non-strict Functional Language. In “Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA’99)”, Sendai, Japan, World Scientific (1999) 72–81
 4. Trinder, P.W., Loidl, H-W., and Pointon R.F.: Parallel and distributed haskells. *Journal of Functional Programming* **12** (2002) 469–510.
 5. Peyton Jones, S.L, Hall, C., Hammond, K., and Partain, W.: The Glasgow Haskell compiler: a technical overview. UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele, (1993) 249–257
 6. Pointon, R.F., Trinder, P.W., and Loidl H-W.: The Design and Implementation of GdH. In: Markus Mohnen, Pieter W. M. Koopman (Eds.) *Implementation of Functional Languages*, 12th International Workshop, Lecture Notes in Computer Science **2011** (2001) 53–70
 7. Barak, A., La’adan, O., and Shiloh, A.: Scalable cluster computing with MOSIX for LINUX. Proc. 5th Annual Linux Expo, Atlanta, GA (1999) 95–100
 8. Dexter, Scott: CIS Department, Brooklyn College. Private communication.
 9. Sunderam, V.: PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience* **2** No. 4 (1990) 315–339
 10. Thompson, S.: Haskell: The craft of functional programming, 2nd edition. Harlow: Addison Wesley (1999)
 11. Hughes, J.: Why functional programming matters. *Computer Journal* **32** (1990) 1–23
 12. Wadler, P.L.: How to declare an imperative. *ACM Computing Surveys* **29** (1997) 240-263
 13. Amar, L., Barak, A., and Shiloh A.: The MOSIX Parallel I/O System for Scalable I/O Performance. Proc. 14-th IASTED Int. Conference on Parallel and Distributed Computing and Systems, Cambridge, MA (2002) 495–500
 14. Loidl, H., Hammond, K., Loogen, R., and Trinder, P.: GpH and Eden: Comparing two parallel functional languages on a beowulf cluster. *Journal of Functional Programming* **2** (2000) 39–53.
 15. Wilkinson, B. and Allen, M.: *Parallel programming: Techniques and applications using networked workstations and parallel computers*. Pearson, New Jersey (2004)
 16. <http://146.245.249.159/kerim/timing.pdf>



Sums of Best of Three Runs for 15-60 STEP 5 Processes

