MPP Parallel Haskell
(Preliminary Draft)

Kei Davis

Los Alamos National Laboratory,
CIC-19, MS B256, Los Alamos, NM 87545, U.S.A.
kei@lanl.gov http://www.c3.lanl.gov/~kei/kei.html

Abstract. MPP Haskell is a parallel implementation of the Haskell functional language for the Thinking Machines Inc. CM-5 large-scale distributed-memory multiprocessor. MPP Haskell is a derivative of GUM, a message-based parallel implementation of Haskell. GUM was carefully designed to minimise performance loss from low-bandwidth and high-latency communications; as an apparent consequence MPP Haskell, with the benefit of high-bandwidth and low-latency communications, achieves remarkably scalable performance.

1 Background

Functional programming languages have long been touted as ideal for parallel implementation. Despite this fact, and the existence of numerous functional language implementations, very few parallel implementations have become generally available; further work in this direction seemed worthwhile.

1.1 Choice of Language

Assuming the goal to be a parallel, non-strict, purely functional language implementation, and given sharply limited resources (man-hours), a practical course would be the port of an existing language implementation to a new platform. Two existing implementations were seriously considered, both high-quality, reasonably mature, and actively supported. The first was Concurrent Clean from the University of Nigmegan [3]. One of its attractions was that it was already being ported to quite diverse hardware and software (OS) platforms, suggesting a careful design to make such ports relatively easy. While the Concurrent Clean team were agreeable to another port, their estimate of the work involved considerably exceeded the available resources.

The other candidate was GUM, described by Trinder et al [5] as a freely available, portable, parallel implementation of the non-strict purely-functional language Haskell. Its use of PVM, a widely ported communications API and distributed computing infrastructure, for communication and task coordination was deliberately chosen to lessen the task of porting. GUM appeared to be a promising starting point.

1.2 Choice of Platform

Before choosing a target platform it is perhaps worth considering current trends in parallel computing. Certainly distributed computing, e.g. on networks of workstations, is a growing and scalable approach. GUM has been ported to such a platform [5]. Commercially, the trend appears to be towards shared memory architectures with a modest number of processors, and away from massively parallel (hundreds or thousands of processors) distributed-memory architectures. Examples of the former include the Sun SPARCServer, DEC AlphaServer, and dual- or multi-processor Intel P5 and P6 systems; GUM has been ported to a Sun multiprocessor. At the same time commercial offerings of massively parallel distributed memory machines are dwindling: the CM-5 was the last of such machines from Thinking Machines Corporation (TMC); the T3E is the last from Cray Research Inc. (CRI); NCR is replacing their loosely coupled MPP with a series of shared-memory MPPs. On the U.S. government-funded development front the ASCI Red proposal for a very large (8096 P6 processor) message-passing distributed-memory machine was recently defeated in favor of the competing ASCI Blue proposal for a very large multi-processor shared-memory machine.

In view of this, a port of GUM to a large distributed-memory multi-processor might appear to be development for a dead-end architecture. Nonetheless, we chose to regard having access to a choice of such machines as an excellent opportunity for the experiment while such facilities remain extant.

Two distributed-memory multiprocessors were readily available: a TMC CM-5 and a CRI T3D. Both are based on GUM-supported processors (SPARC and DEC Alpha, respectively) and run implementations of PVM. While the T3D is the more modern and powerful machine, the fact that it does not run a GUM-supported operating system or a GUM-required GNU C (gcc) compiler, both of which the CM-5 do, made the latter the preferable choice.

2 The TMC CM-5

The Los Alamos National Laboratory CM-5 comprises 1056 computational nodes, each having the maximum 32MB main memory, a SPARC Cypress CPU running at 32 MHz, and four vector processors. Each user program is allocated to a *partition*—group of computational nodes—by one of several *partition managers*, each of which runs on its own processor (currently a SPARC 10 model 51). The number of nodes in a partition is constrained to be a power of two between 32 and 1024. Partitions may be time-shared or dedicated to a single user program.

Point-to-point communications (message passing) between the processors is via the *data network*, which supports multiple messages in simultaneous transit. The programmer's interface to the data network is via the CMMD message-passing (and timing) library. While the data network has a 32 MB/sec/node bandwidth, 'ping-pong' tests show, for example, a total time (including hardware and software latencies) of 1.5-1.9 microseconds to transmit a one-word message. The CMMD library provides both synchronous and asynchronous message passing.

Each partition on the CM-5 runs the CMOST operating system, described by TMC as an enhanced UNIX operating system giving access to all facilities normally available in SunOS 4.x.x. What is undocumented fact is that partition manager runs a complete version of the OS while each PE runs a so-called *microkernel* that provides a very small subset of the full OS functionality, and much of that via the partition manager.

The CM-5/CMOST/CMMD environment supports two programming models: *host-node* and *hostless*. In the host-node model the user provides two programs, one to run on a host processor (a partition manager), and one that is replicated on each node of the partition. In the hostless model a single user program is replicated on each node of the partition with an invisible host provided by the CMMD library. In both cases the host is responsible for starting and monitoring the node programs and acting as their I/O server. The PVM implementation runs on top of CMMD.

While it is possible in the existing environment to perform all software development on the CM-5 itself, in practice it is required that editing, compilation, and linking be performed on front ends, themselves Sun workstations.

3 Porting GUM

Despite the apparent ease with which a port of GUM to the CM-5 could be achieved, there were a number of unexpected impediments.

The first step was to build the Glasgow Haskell compiler on one of the front ends. No difficulty was encountered. The second was to run a serial Haskell program (produced with the new compiler) on a CM-5 host node. Again there was little difficulty.

Next was an attempt to run a parallel Haskell program under PVM on the CM-5. At this point the surprising dearth of OS services on the nodes elements became painfully obvious, e.g. not even supporting something so basic to UNIX as *exec*(). After working around this it quickly became clear that the PVM implementation was, for several reasons, not up to the task.

(It is worth noting that the hardware is perfectly capable of supporting MIMD programming—multiple disparate processes on each node, with arbitrary inter- and intra-node communications between processes, with full OS support for each process. Such is the nature of 'scientific computing' that such flexibility is not in great demand and so not provided.)

In short, PVM was scrapped in favor of direct use of the CMMD library. Since the Glasgow Haskell compiler produces (in essence) two programs, a *system manager*, and the parallel program proper which is duplicated on each processing element, it seemed reasonable to use the host-node programming model with the system manager as the host and the parallel program duplicated on each node. This proved to be infeasible, mainly because of the intricate requirements of the host to provide time-critical I/O services.

Sacrificing one node to serve as system manager provided a satisfactory solution. Because precisely the same code must be replicated at each node (with no possibility of using *exec*() to replace it at runtime), the system manager was merged with the parallel program proper.

4 Performance

Following Trinder et al [5] we give performance results for the simple divide-and-conquer parallel factorial program given in Appendix A. Figure 1 displays the speedups for varying numbers of processors and problem sizes. The cut-off value—the second parameter to `parfac`—was 1000 in all cases. The number of processors ranged from one to 97.[1] Four lines are plotted: $y = x$—the ideal speedup, and in increasing order of relative speedups lines corresponding to problem sizes of 10^5, 10^6, and 10^7, respectively.

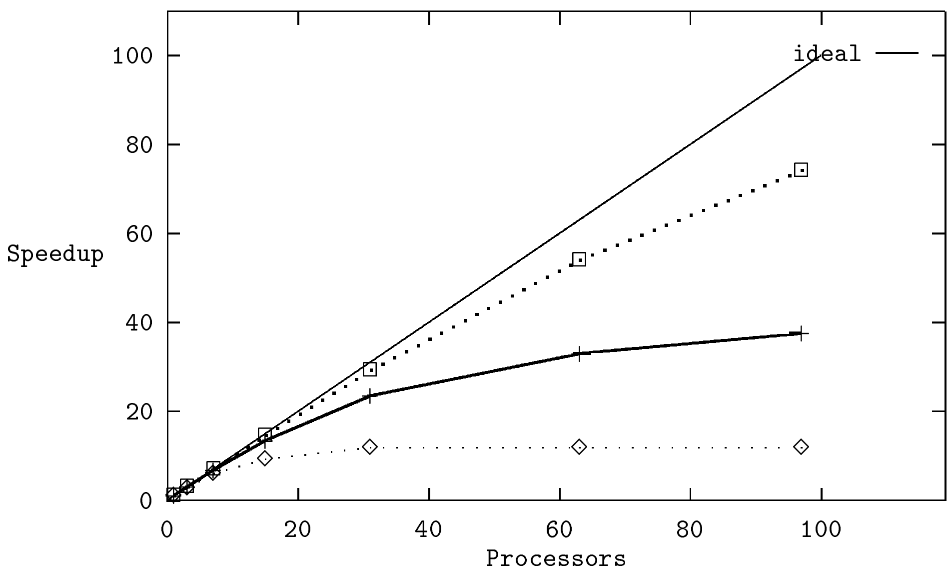

Fig. 1. Absolute speedups

It is interesting to observe that the potential number of threads must considerably exceed the number of processors to achieve good speedup with the chosen cut-off value; for example for problem size 10^5 with corresponding number of threads approximately equal to 100 the best speedup obtainable was 11.8. Increasing both the problem size and cut-off value by a factor of ten, so keeping the number of threads the same, only increased the speedup on 31 processors from 11.8 to 13.7 (not shown in figure).

[1] This odd number because of apparent limitations in the GUM profiling tools.

5 Conclusion

Though the performance testing to date is very sketchy, we can make a few tentative observations and conclusions.

Scalability. We can get good absolute speedups with upwards to one hundred processors. What order of speedup can potentially be obtained using the full capacity of the CM-5—1024 processors—has yet to be explored.

Usefulness. While old and slow by contemporary standards, the CM-5 is very much a laboratory 'workhorse'. A critical feature of a MPP in the laboratory environment is the existence of the vector units on each node. Using the foreign language interface of the Glasgow Haskell compiler it should be possible to provide access to the vector units. Even were that provided there would unfortunately remain a much greater impediment to its general use: it is probably fair to say that interest in functional programming in the 'scientific computing' community is very close to nil.

A direction forward. The Glasgow Haskell compiler provides only `par` and `seq` primitives for controlling parallel execution. A much nicer scheme has been proposed by Mirani and Hudak [2] wherein language primitives allow the definition of *first-class schedules* for specifying partial orders on the evaluation of subexpressions, and of *first-class maps* for specifying mappings of processes to processors. The feasibility of incorporating this scheme into the compiler would be worth investigating.

6 Acknowledgments

Many thanks are due to Simon Peyton Jones, Hans Loidl, and especially to Phil Trinder and Partain, for their cheerful provision of a steady stream of information and explanation on the inner workings of GUM and the Glasgow Haskell compilation system.

References

1. Hudak, P., et al: Report on the programming language Haskell. ACM SIGPLAN Notices, Vol. 27, No. 5, May 1992.
2. Mirani, R., and Hudak, P.: First-class schedules and virtual maps. Proceedings of Functional Programming Languages and Computer Architecture, La Jolla, CA, USA, 1995.
3. Nöcker, E.G.J.M.H., J.E.W. Smetsers, M.C.J.D. van Eekelen and M.J. Plasmeijer: Concurrent Clean. In Proc. of Parallel Architectures and Languages Europe (PARLE '91), Springer-Verlag, LNCS 505, pp. 202-219, 1991.
4. Roe, P.: Parallel programming using functional Languages. Ph.D. thesis, CSC 91/R3, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, U.K., 1991.

5. Trinder, P.W., Hammond, K., Mattson, J.S., Partridge, A.S., and Peyton Jones, S.L.: GUM: a portable parallel implementation of Haskell. Proceedings of Programming Language Design and Implementation, Philadelphia, USA, 1996.
6. The Connection Machine CM-5 technical summary. Thinking Machines Corporation, Cambridge, MA, USA, 1991.

A The parallel factorial program

```
module Main(main) where

import Parallel

pfc :: Int -> Int -> Int -> Int
pfc x y c
   | y - x > c = f1 `par`
                 (f2 `seq` (f1+f2))
   | x == y    = x
   | otherwise = pf x m + pf (m+1) y
   where
      m = (x+y) `div` 2
      f1 = pfc x m c
      f2 = pfc (m+1) y c

pf :: Int -> Int -> Int
pf x y
   | x < y = pf x m + pf (m+1) y
   | otherwise = x
   where
     m = (x+y) `div` 2

parfact x c = pfc 1 x c

main
 = getArgs exit ( \[a1, a2] ->
    let x = fst (head (readDec a1))
        c = fst (head (readDec a2))
    in
        appendChan stdout
                   (show (parfact x c))
                   exit done)
```