Millipede: Easy Parallel Programming in Available Distributed Environments[*] [**] (Extended Abstract)

Roy Friedman[1], Maxim Goldin[2], Ayal Itzkovitz[2], and Assaf Schuster[2]

[1] Department of Computer Science, Cornell University, Ithaca, NY 14850.[***]
[2] Department of Computer Science, The Technion, Haifa, 32000, Israel.

Abstract. MILLIPEDE is a generic run-time system for executing parallel programming languages in distributed environments. In this project, a set of basic constructs which are sufficient for most parallel programming languages is identified. These constructs are implemented on top of a cluster of workstations such that in order to run a specific parallel programming language in this distributed environment, all that is needed is a compiler, or a preprocessor, that maps the source language parallel code to the MILLIPEDE constructs. Some performance measurements of parallel programs on MILLIPEDE are also presented.

1 Introduction

The idea of using a cluster of workstations as a cost/effective high-performance parallel computer is becoming feasible as local area networks become faster and faster. This has led to the development of many run-time systems that support parallel programming languages on cluster environments in recent years [1, 3, 4, 5, 7, 8, 9, 10, 11, 12]. (A detailed comparison with these systems and others appears in the long version of this paper.) These run-time systems mainly differ by the memory consistency protocols they support, the load balancing schemes they employ, and the programming language they are targeted for. However, each of these run-time systems supports only one parallel programming language.

This means that in order to use an existing run-time system, one needs to adapt himself/herself to the one and only programming language which is supported by it. In this project, we have developed MILLIPEDE, a generic run-time system for parallel applications, which can support a variety of parallel programming languages. MILLIPEDE provides a flexible interface for creating parallel activities in the system, a distributed shared memory management, load balancing, and synchronization methods in the form of library routines. These

[*] The MILLIPEDE project is supported by Intel Academic Relations Grant, by Microsoft R&D group, and by a grant from the Israeli Ministry of Industry and Commerce.
[**] The MILLIPEDE URL: `http://www.cs.technion.ac.il/~assaf/millipede.html`
[***] This author is currently supported by ARPA/ONR grant N00014-92-J-1866. Much of this work was done while this author was still with the Department of Computer Science at the Technion.

can be used by a compiler, or a preprocessor, to map the primitives of the language to the constructs provided by MILLIPEDE. Thus, the only thing that is required in order to implement most of the parallel programming languages on MILLIPEDE is to make the appropriate changes in its compiler, adjusting it to MILLIPEDE's interface, which is much easier than developing a new run-time system from scratch.

The goal of MILLIPEDE is therefore to support a large variety of parallel programming languages using commodity parts and standard operating systems so that most people can continue using their "favorite" parallel programming language on the same run-time environment which works with their existing equipment. Also, since it is relatively easy to incorporate new programming languages to the system, languages which are not supported at this point can be added on demand. For people that have no experience with parallel programming, MILLIPEDE currently supports the PARC programming language, which is a simple, yet powerfull, extension of C to parallel programming [2]. (We elaborate on PARC in the full version of this paper [6].) Other languages that we are currently supporting include a parallel version of C++ (CParPar), Java, and Par-Fortran 90. We intend to support Cilk, SPLASH macros, and Split-C soon.

For further flexibility and in order to be able to adapt to the particular needs of different applications, MILLIPEDE supports both *strong* and *weak consistency* memory management, which can be chosen by the application, and a variant of *release consistency* is being implemented. Also, several load balancing schemes have been implemented, and they can also be picked by the application. These protocols and schemes are described in more detail in the full version of this paper [6]. MILLIPEDE is fully implemented on MACH, and a more advanced version is under final stages of implementation on Windows-NT^{TM}.

We have conducted several performance measurements on the MACH implementation, and initial results are very encouraging. These results prove that at least for a large class of programs, it is possible to provide reasonable performance while running on a cluster of off-the-shelf workstations with an unmodified operating system, connected by an off-the-shelf network, and using a generic run-time system and a friendly programming language. Most programs achieve good speedups, while the speedups for programs that have natural parallelization is close to linear. Also, there seems to be a correlation between the problem sizes and the speedups, indicating that when the problem is large enough, the benefits of using our system outweighs the overhead imposed by it. We believe that by further optimizing the implementation, we will be able to achieve even better performance.

2 Millipede's support in ParC's constructs

MILLIPEDE provides powerfull interface that support PARC's constructs. PARC provides four constructs for creating parallel activities in the system:

pparblock creates a given number of activities, each being specified separately. In particular, this can be used to implement **fork**-like primitives.

lparblock similar to pparblock, but the number of parallel activities created is limited to some multiplication of the number of processor, so that if more blocks are specified, some of them will be executed within the same activity. This is useful for improved performance.

pparfor creates a given number of similar parallel activities which differ only by an index which is passed to them. Mainly useful for creating parallel loops.

lparfor similar to pparfor, but the number of actual parallel activities which are created is limited to some multiplication of the number of processor in the system.

The the following synchronization and atomic access constructs are provided: **sync** which is a barrier synchronization, **faa** for atomic fetch-and-add operations, **tss** as an atomic test-and-set operation, **rst** as an atomic reset operation, and **semaphore_wait** and **semaphore_signal** for mutual exclusion in critical sections. PARC also supports two main calls for early termination of parallel activities: **pcontinue**, terminates the current parallel activity and **pbreak**, terminates the current parallel activity and its siblings.

In order to run a PARC program in MILLIPEDE, a compiler for the source code programming language must first translate it into object code, with the appropriate calls to MILLIPEDE library functions, and must add a call to MILLIPEDE initialization routine. Then, the code can be linked together with the MILLIPEDE library to create the executable which runs both the MILLIPEDE run-time system and the application. When the application is executed, typically one activity is first created on one of the nodes. Then, as the execution develops, more parallel activities may be created or terminated on various nodes of the system, and parallel activities may migrate from one node to the other, depending on the load balancing policy chosen. This process is explained in more detail in the full version of this paper [6].

3 Performance

We have conducted several performance measurements which include matrix multiplication, finding the shortest distance in graphs, the traveling salesperson problem, and a linear equation solver. Due to space limitations, we only present here the high-lights of these measurements, but the full version of this paper contains a detailed description of these results [6].

In the case of matrix multiplication and finding the shortest distance in graphs, our system achieved close to linear speed-ups with problem size (matrix size and number of nodes respectively) of 400 elements,or more and 4 nodes. For the traveling salesperson problem we achieved a speedup of 60% with 3 nodes, but only marginal speedup for the linear equation solver.

4 Conclusions and Further Research

In this work we have demonstrated that building a generic run-time system that can support a variety of programming languages and provide reasonable performance while using only commodity parts can be done. In our approach, adding support for a new parallel programming language means only slightly modifying its compiler, which increases the accessibility of distributed environments to parallel applications. This way all programming languages ported to our system can enjoy the same set of memory consistency and load balancing protocols, which can match the performance needs of their applications.

We are currently experimenting with porting more parallel programming languages to MILLIPEDE, supporting more memory consistency protocols, and developing new load balancing schemes. We are also investigating ways of incorporating fault-tolerance into our system.

References

1. H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch, and R. Schwarz. PANDA - Supporting Distributed Programming in C++. In *Proc. 7th European Conf. on Object-Oriented Programming (ECOOP)*, Kaiserslautern, July 1993.
2. Y. Ben-Asher, D.G. Feitelson, and L. Rudolph. ParC: An Extension of C for Shared Memory Parallel Processing. *Software Practice & Experience*, 1995.
3. B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON*, pages 528–537, February 1993.
4. Robert D. Blumofe and David S. Park. Scheduling Large-Scale Parallel Computations on Networks of Workstations. In *Proc. 3rd Symp. on High-Performance Distributed Computing*, pages 96–105, August 1994.
5. J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *In proc. 13th Symp. on Operating Systems Principles*, pages 152–164, October 1991.
6. R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. Technical Report LPCR-9506, Department of Computer Science, The Technion – Israel Institue of Technology, November 1995.
7. A.M. Vahdat D.P. Ghormley and T.E. Anderson. Efficient, Portable, and Robust Extension of Operating System Functionality. Technical Report CS-94-842, University of California - Berkeley, 1994.
8. P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory On Standard Workstations and Operating Systems. In *USENIX*, pages 115–131, January 1994.
9. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.
10. H. Mehl. Distributed Shared Memory: A Survey. Technical Report SFB124-33/92, University of Kaiserslautern, Department of Computer Science, P.O.Box 3049, D-6750 Kaiserslautern, Germany, 1992.
11. A. Mohindra and U. Ramachandran. A Survey of Distributed Shared Memory in Loosely-Coupled Systems. Technical Report GIT-CC-91/01, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, January 1991.
12. B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *COMPUTER*, pages 52–59, August 1991.