LA-UR -87-673

CONF-8610357--1

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

LA-UR--87-673

DE87 006066

TITLE: PARALLEL GRAPH REDUCTION ON A SUPERCOMPUTER: A STATUS REPORT

AUTHOR(S): Randy Michelsen, C-10
Lauren Smith, C-8
Elizabeth Williams, C-8
Bonnie Yantis, C-10

SUBMITTED TO: To appear in "Proceedings of Santa Fe Graph Reduction Workshop." Springer-Verlag

DISCLAIMER

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO 836 R4
ST NO 2629 5/81

Parallel Graph Reduction on a Supercomputer: A Status Report

Randy Michelsen
Lauren Smith [1]
Elizabeth Williams
Bonnie Yantis

Computing and Communications Division[2]
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

We describe an ongoing effort to develop a parallel graph reduction run-time system hosted on a multiprocessor supercomputer. This run-time system is presently augmented by the functional language compiler of the ALFALFA system. Admittedly, parallel graph reduction is hardly a novel idea. The interesting notion is the provision of a parallel execution environment sufficiently powerful to support development of large prototypical scientific and symbolic codes in a functional language. This will allow the investigation of the functional programming paradigm within these application domains in an empirical fashion. In this paper, we discuss the motivation for the effort, describe the basic elements of the implementation, and provide some preliminary insight distilled from our experience with an initial version of the run-time system.

1. INTRODUCTION

The continuing demand for increased computational power, especially in the realm of scientific computing, has spurred interest in parallel computation. Today, most application development for computationally intensive algorithms intended for high-performance computers is done in Fortran. Hence, there exists a considerable body of knowledge regarding its use and performance in such domains. We are interested in collecting similar data relating to use of the functional (or applicative) paradigm.

Toward this end, we are currently developing a parallel graph reduction run-time system for a supercomputer incorporating four high-speed processors. The code generation is performed by an existing functional language compiler resident on a traditional uniprocessor. Initially, the focus of the effort was the conversion, with minimal change, of a run-time system originally developed for a distributed memory multiprocessor to a shared memory multiprocessor. This strategy enabled us, in the near term, to rely upon an existing compiler and concentrate our energy instead on the development of the reduction run-time system. Subsequently, the emphasis will shift to performance tuning of the run-time system and the investigation of related compilation and scheduling issues.

Our motivation for undertaking this task was essentially twofold. A common complaint from potential users of functional languages outside of the language research community is the pragmatic impossibility of developing programs of a "realistic" size simply due to performance-related issues. The definition of realistic is of course open to interpretation, but it is worthy of note that typical production programs at Los Alamos National Laboratory (hereafter, the Laboratory) consist of 100,000

---

[1] currently at Department of Defense, 9800 Savage Road, Fort Meade, MD 20755, electronic mail address: lls@mimsy.umd.edu

[2] This work was performed under the auspices of the Department of Energy.

or more lines of Fortran. The performance deficiencies usually associated with functional languages may be attributed in part to the fact that most implementations currently available are intended for language research and lack the maturity (and implied fine tuning) of production compilers. Furthermore, the implementations generally exist on machines lacking the requisite processing and memory capacity for large-scale scientific computations. Our effort is particularly apropos given the latter circumstance.

We were also interested in exploring the performance ramifications of a large physical memory shared by cooperating reduction "engines." There have been graph reduction systems implemented on distributed memory machines or networks of machines, including several discussed in this volume. Yet, given the usual logical view of a graph space global to all reduction activity, a large shared memory architecture seems particularly well suited to parallel graph reduction.

As a result of these interests and practical concerns such as machine availability, we chose the CRAY X-MP computer as the current implementation target. This architecture combines a small number of high-performance processors with a large shared memory.

Section 2 contains background information and a brief discussion of the CRAY X-MP architecture. [2] Section 3 describes the design of the initial run-time system. Section 4 discusses early experience with the system and a recently initiated follow-up development effort. A summary is presented in Section 5.

2. BACKGROUND

2.1. The ALFALFA System

The work described here is closely related to the work on ALFALFA by Benjamin Goldberg and Paul Hudak of Yale University. We shall present a very brief description of the ALFALFA system, but the interested reader is referred to B. Goldberg's paper in this volume for a more complete treatment of the subject. ALFALFA consists of a compiler for the functional language ALFL [3] and an associated parallel graph reduction system resident on an Intel iPSC multiprocessor. A source program is parsed into an intermediate graph form called LIF (Lambda Intermediate Form). This graph is then annotated as a result of extensive type, sharing, and strictness analysis. The annotated graph is converted into a serial combinator [4] program graph. Finally, C procedures are generated from this graph for the combinators. This C program is then compiled and linked with the graph reduction system resident on the processing nodes of iPSC.

2.2. The CRAY X-MP

The CRAY X-MP is a multiprocessor derivative of the earlier CRAY-1 family of supercomputers. The basic architecture consists of an interleaved, multiport memory and input/output subsystems shared by several identical high-performance vector processors. Each processor has a major cycle time of 8.5 nanoseconds. Pipe-lining is utilized in the scalar and vector functional units, with parallel activity possible among the independent functional units. A parallel program has available to it shared registers - eight data registers, eight address registers, and 32 single-bit semaphore registers. A blocking test-and-set instruction can be performed on each semaphore register. Unlike some of its counterparts, the X-MP does not support virtual memory, but rather uses a contiguous allocation scheme for memory management. The X-MP is available in a variety of configurations, currently with up to four processors and sixteen million 64-bit words of main memory.

The primary use of modern supercomputers as "number crunchers" is reflected in the software environment of the Cray. The most mature and heavily used compiler available at the Laboratory is the vectorizing Fortran compiler developed by Cray Research Inc. Fortunately, a locally supported C compiler is also available. The C language was clearly more attractive in our development because of portability concerns and the availability of language features such as dynamically allocatable data structures.

The processors of a CRAY X-MP have traditionally been used to support multiple, independent job streams. However, a set of routines has been developed at the Laboratory to enable user program initiation and control of parallel tasks. These multitasking routines [1], written in Fortran and Cray assembly language, provide constructs for task creation, termination, and various styles of cooperation among asynchronous tasks. Among the latter are analogues of traditional synchronization mechanisms such as generalized counting semaphores, fetch/add instructions, locks, events, barriers, forks, and joins. Because of the real memory organization of the X-MP, the multitasking routines rely upon memory management utilities developed and in use at the Laboratory.

3. THE CRAY GRAPH REDUCTION RUN-TIME SYSTEM

Since one of our intentions was to provide access to Cray-class computational power for functional language programs in a timely fashion, the initial version of the Cray reduction system is a variant of the ALFALFA system developed for the Intel iPSC. Of course, given the disparate target architectures of the two efforts, there are considerable differences to accommodate. We are currently using the compiler from the ALFALFA system as the code generator for this reduction system. The C source code produced by the compiler undergoes source-to-source transformations to produce the appropriate Cray-compatible code. At this point, the source code is loaded onto the Cray. These combinator definitions are compiled and linked with the resident run-time system.

An alternative design for the run-time system was considered, but eventually rejected in favor of the design presented in this section. A primary goal in this initial implementation was to produce a run-time system highly compatible with the original ALFALFA compilation system, with some consideration given to performance factors. The candidate design was to associate with each node being reduced a separate reduction run-time system, with scheduling of these tasks performed by the host's operating system. This design was rejected because of the high overhead of task management relative to the actual work associated with the reduction activity.

In the selected design (refer to Figure 1), each physical processor executes a reduction task that is the run-time system. The term "task" is used in this context to designate an executing program image on the Cray host. The program graph and a queue of ready nodes, i.e., graph nodes that are candidates for immediate reduction, reside in shared memory. The common node queue is accessed by the reduction tasks whenever new work is desired. Access to this queue and the nodes of the program graph is controlled through the use of synchronization variables. Error handling and normal termination of the run-time system are accomplished via the notion of "synchronized meetings" provided by the multitasking routines. This construct allows a set of parallel tasks to synchronize on a single variable, denoted as the "meeting flag." After all the tasks meet at this synchronization point, a single task executes a sequential section of code. Upon the completion of this section of code, all tasks resume parallel execution. A variant of this design uses a fixed number of reduction tasks, which is strictly greater than the number of processors. When one reduction task must wait to gain access to a lock, another task may be able to reduce a node if one is on the ready queue; thus, all processors are executing if nodes are ready.

As indicated earlier, the run-time system is written primarily in C. Because of the need to interface the Fortran multitasking routines and the run-time system, a Fortran main program actually initiates the bootstrap routine for the run-time system. The C bootstrap routine initializes the ready node queue, the various synchronization variables, and constructs the initial program graph. When the initialization is complete, four parallel reduction tasks are created. One of the reduction tasks determines that it is the root (using a shared variable) and initiates the graph reduction. The tasks continue selecting nodes from the ready queue, mutating the program graph, and placing nodes on the ready queue until the graph is reduced to normal form or there is an abnormal termination.

I/O, because of its sequential nature, is accomplished through the use of a software "lock" on the I/O module. When one of the reduction tasks needs to perform I/O, it attempts to gain possession of the I/O lock. If the lock is unavailable (i.e., already possessed by another task), the requesting task waits until it is available. Memory allocation and deallocation are handled in a similar fashion.

Error handling is accomplished by a synchronized meeting. If a serious error is detected by one of the reduction tasks, the task sets the synchronization variable and initiates a meeting. The other

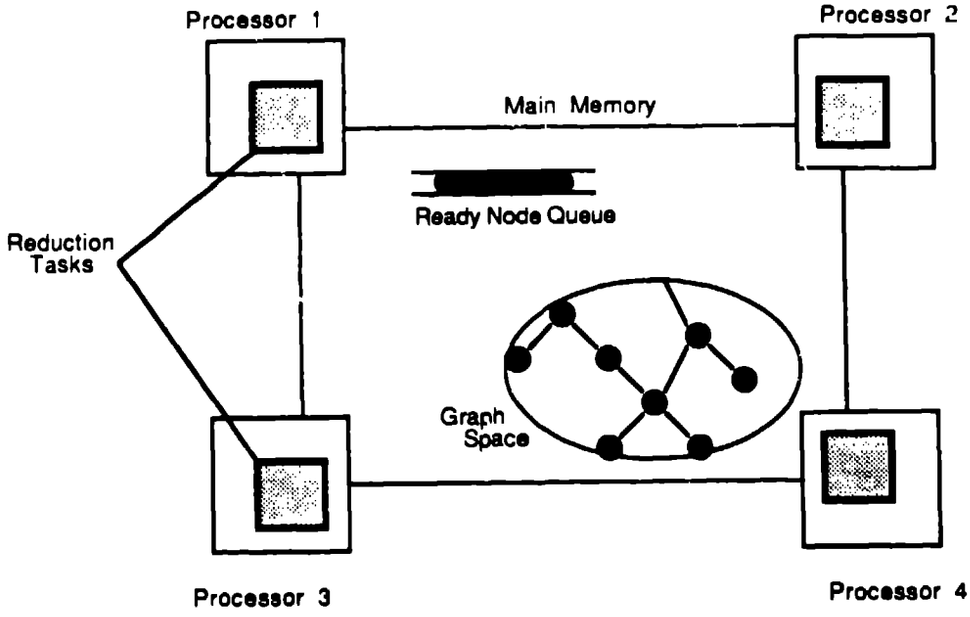

Figure 1. The Cray run-time system.

reducers will synchronize and the initiating task will execute the appropriate error handling. To ensure correct behavior in the advent of a meeting requested as a result of a fatal error, the tasks of the run-time system periodically inspect the meeting flags associated with this event. This inspection is critical in order to avoid potential deadlock as a result of tasks waiting for locks retained by abnormally terminated tasks. The occurrence of a fatal error will cause logging of state information, followed by the termination of the reduction tasks.

4. EARLY EXPERIENCE

At this point, the reduction system has been applied to simple ALFL programs. Several performance obstacles have been identified on the basis of our limited experience with the Cray environment, including the multitasking routines. The granularity of computation is far too small to be practical given the raw performance characteristics of the Cray processor. For example, the overhead associated with a procedure call is approximately 40 major cycles. This indicates a need to increase the granularity of the combinators generated by the compiler, as well as a need to significantly restructure the reduction system itself to reduce the cumulative performance impact of procedure invocation. Another obvious candidate for further scrutiny is synchronization. The tactic in this instance must be to reduce points of synchronization and to minimize the overhead associated with each synchronization.

These issues, among others, have been addressed in a subsequent development project targeted toward enhanced performance. In particular, the reduction system was redesigned to benefit more fully from the shared memory model, to more accurately reflect the performance characteristics of the Cray architecture, and to support measurement of performance and workload characteristics. Through careful design and utilization of a different mechanism, the minimum overhead incurred to control access to shared objects such as the ready node queue has been reduced by a factor of 7 to 15. Finally, an alternative scheme for internal data representation has been employed that is more amenable to possible future optimizations. This scheme supports memory allocation in a manner such that effective utilization of the Cray vector hardware is possible where appropriate.

5. SUMMARY

We briefly discussed the status of a parallel graph reduction system for the CRAY X-MP multiprocessor. A large portion of the initial effort was devoted to, out of sheer necessity, deciphering the interactions of several disparate software systems (e.g., memory management utilities). This effort will be fully appreciated by those acquainted with the rather meager software development support tools traditionally available on supercomputers.

The poor performance of the initial version of the run-time system prompted the development of a second redesigned system. This system is presently undergoing integration testing. This version has been tailored for increased performance on the host architecture. In particular, synchronization and other operational overheads have been significantly reduced.

In the future, we anticipate performing characterization studies of prototypical applications and additional performance tuning of the run-time system. The former should provide important data for the investigation of issues such as task granularity and scheduling. Finally, the entire question of vectorization has been ignored but must be addressed in any truly viable high-performance system.

6. REFERENCES

(1) Bobrowicz, F., "The Los Alamos Multitasking Control Library," Los Alamos National Laboratory unclassified release (in preparation).

(2) Cray Research, Inc., *CRAY X-MP Series Mainframe Reference Manual*, HR-0032 (1982), Cray Research, Inc.

(3) Hudak, P., "ALFL Reference Manual and Programmer's Guide," Research report YALEU/DCS/RR-322, 2nd ed., Computer Science Department, Yale University, October 1984.

(4) Hudak, P., and Goldberg, B., "Distributed execution of functional programs using serial combinators," *Proc. 1985 International Conference on Parallel Processing*, August 1985, pp. 831-839.