Preliminary Performance of a Parallel GLU Implementation on a Network of Workstations

R. Jagannathan and C. Dodd
Computer Science Laboratory
SRI International
Menlo Park, California 94025
U.S.A.

Abstract

The performance of a workstation network implementation of the basic virtual architecture for parallel execution of GLU applications is studied. Preliminary results suggest that coarser granularity of parallelism in GLU applications results in superior scalability, lower overhead, and better efficiency.

1 Background

GLU is a software platform for parallel applications programming. It is designed to allow existing sequential software to be executed in parallel on extant and future parallel computer systems with only minimal programmer involvement.
The GLU platform (Figure 1) consists of two components.

1. A high-level integration language in which procedural code can be composed into a coherent, implicitly parallel application.

2. A runtime system that embodies a virtual architecture for executing GLU applications. The architecture derives speedup by exploiting implicit coarse-grain parallelism in these applications.

We briefly describe the essential aspects of the GLU integration language. (More details can be found in [4].) The GLU language, which can be thought

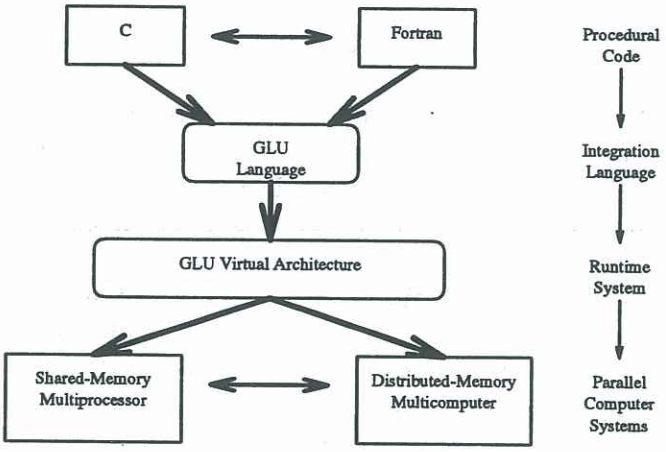

Figure 1: Architecture of the GLU Platform

of as a high-level software structure specification language, consists of two tiers. The top (coordination) tier consists of a multidimensional dataflow language (*Indexical Lucid* [3]) that is used to express the parallel structure of computation. The bottom (computation) tier is a procedural language using which computations themselves are expressed as (procedural) functions. A GLU program simply specifies the data dependencies between various procedural functions. These data dependencies implicitly express function-level and data-level parallelism which can be exploited at execution time. The parallelism is granular; the grain size depends on the complexity of the procedural functions. Each procedural function is written in a conventional language such as C – in many cases, using existing code.

In this paper, we focus on a simple virtual architecture for exploiting granular parallelism and study its performance by considering its implementation on a network of workstations.

2 The Basic Virtual Architecture

We describe the basic virtual architecture that has been designed to exploit granular parallelism in GLU programs. Before doing so, we consider the model of computation underlying the basic virtual architecture, namely *eduction.*

Eduction is a novel model of parallel computation which loosely corresponds to lazy dynamic dataflow [2]. An important consequence of laziness is that, while it shares the greater asynchrony of its eager counterpart, it avoids superfluous computation altogether. This is particularly important since procedural functions with side effects in GLU programs should not be invoked when unnecessary, even if the side effects are benign as far as the computation is concerned.

The most natural way to realize eduction is by tagged demand-driven execution. Basically, a term in a GLU program is evaluated (at a particular context) only when it is demanded at that context (as identified by the demand's tag). The evaluation consists of two phases: first, simultaneous demands for the constituent subterms at appropriate contexts are propagated and, second, the function associated with the term is applied when the demanded values of each satisfied subterm are available, thereby producing the originally demanded tagged result.

An important aspect of eduction is that it does not recompute the value of a term at a given context (or tag) if that value has been previously demanded. Instead, the tagged value is stored (in what is referred to as the variable-value store) when the initial demand for it is satisfied, and is used to satisfy subsequent demands for the value. (Effective management of the variable-value store is critical to the successful realization of eduction.)

The eduction model exploits two kinds of parallelism. The first kind is functional parallelism, which is exploited in the simultaneous computation of tagged values of subterms demanded to produce a tagged value of a term. The granularity of parallelism is coarse if the subterms correspond to procedural functions. Given the program fragment, `f( g( y ), h( z ) )`, if the tagged value `f(...):c` is demanded (`c` is the tag of the value that is sought), demands for `g(...):c` and `h(...):c` would be simultaneously made, which in turn would demand tagged values `y:c` and `z:c`. When the latter become available, functions `g` and `h` can be applied simultaneously, resulting in functional parallelism.

The second kind of parallelism is context parallelism (which loosely corresponds to data parallelism). This is exploited in the simultaneous eval-

uation of the same term at multiple contexts. As a simple example, with the above program fragment, the term corresponding to `f(...)` can be demanded simultaneously at contexts `c` and `d` and the demanded tagged values of subterms at the two contexts can be computed independently since there is no contextual dependency within the term. Further note that the exploitation of context parallelism can occur independent of the exploitation of the functional parallelism.

A more elaborate expression of context parallelism is in "tournament computation" which essentially corresponds to a multi-dimensional computation tree. (The tournament method is a standard technique in computer algorithms.) All subcomputations at a given tree level can proceed in parallel, giving rise to context parallelism. Figure 2 illustrates the computation associated with a tournament to compute the maximum of a sequence of numbers. The exploitation of context parallelism at each level should be obvious.

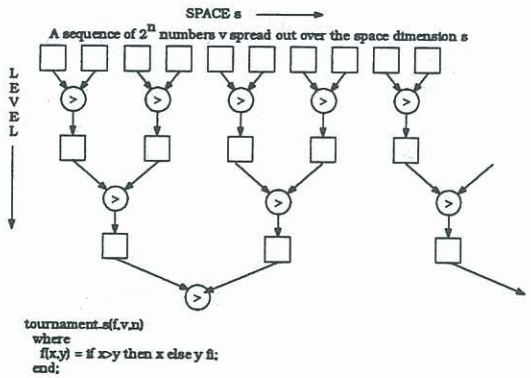

Figure 2: Dynamic View of Tournament Computation

Figure 3 illustrates the basic virtual architecture for parallel execution of GLU applications. The basic architecture consists of a generator (G) and several executors (E). The generator emulates demand propagation, fine-grain operation execution, result collection and storage. The generator uses the executors to apply coarse-grain functions of GLU programs to their arguments, producing results that are returned to the generator. The parallelism exploited is in the concurrent execution of coarse-grain functions by the various executors. The grain size depends on the structure of the GLU program itself. The load distribution strategy for the limited archi-

tecture is completely dynamic; that is, each executor waits on the generator to assign a coarse-grain function to execute and waits again upon returning the result. Furthermore, the basic architecture can accept addition of executors at any point in the computation. The architecture is limited in performance because there is an upper limit on the number of executors that a generator can keep busy with work. This limit depends on a number of implementation factors such as the bandwidth available between generator and executors, the overhead associated with demand propagation and value storage management.

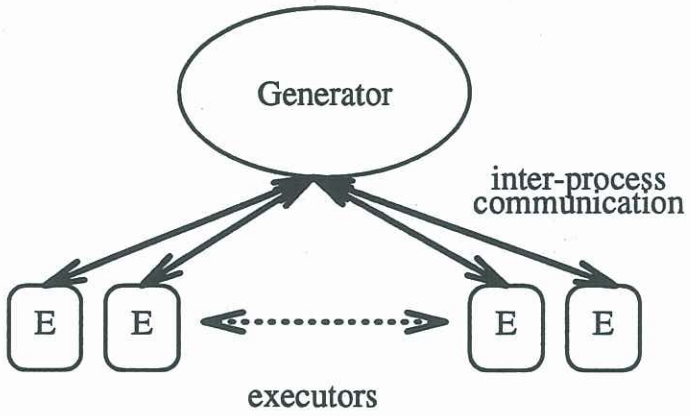

Figure 3: The Basic Virtual Architecture

3 Experiments

The basic architecture has been implemented on a network of workstations with the generator and executors being realized as processes and the communication between them being realized by an asynchronous remote procedure call mechanism. Typically, the generator runs on one workstation (known as the "home" workstation) and each of executors runs on other autonomous workstations. (We use "executor" and "processor" interchangeably.) In particular, the basic architecture has been implemented on a network of Sun SparcStations (SS1+). Each workstation has at least 8 megabytes of memory and a local swapping device. It is capable of delivering approximately 16 MIPS. The workstations share a common file server for permanent file

storage and retrieval. (For the purposes of this experiment, we only consider upto eight workstations.)

The following parallel applications have been used to evaluate the performance of the particular implementation.

1. Parallel (block decomposition method) 400x400 matrix product using 200x200-sized and 100x100-sized blocks;

2. parallel mergesort of 512k elements using chunks of size 64k and size 8k elements; and

3. parallel raytracing of 8kx8k pixels using 4k pixel regions.

Each of the applications essentially embodies a tournament computation [5, 1]; thus, each application has considerable implicit context parallelism. The context parallelism is granular; the coarseness of granularity is controlled by the application. Thus, the granularity of parallelism in matrix product with 200x200-sized submatrices as the basic blocks is coarser than matrix product with 100x100-sized submatricies. Similarly, the granularity of parallelism in mergesort with chunks of 64k elements is coarser than with chunks of 8k elements.

We examine three aspects of the basic architecture implementation.

1. **Scalability.** Scalability refers to the the ability to sustain a certain efficiency (of speedup) with increasing number of processors. Efficiency for a certain application instance and for a certain number of processors is defined as the ratio of the best possible elapsed time and the observed elapsed time using GLU. Note that the best possible elapsed time ignores all forms of overhead including communication latency. For each application, we consider the effect of the number of processors on its efficiency.

2. **Generator Overhead.** The generator of the basic architecture coordinates the parallel execution of a GLU application. Thus, the ratio of the time spent in coordination (which includes communication to and from executors and implementation of eduction) to the total elapsed is viewed as overhead associated with the generator. For each application, we consider the effect of the number of processors on the overhead incurred by the generator.

3. **Optimal Granularity.** We think of grain size as simply the amount of computing performed (measured as time) at each executor per unit

communication cost (which is the time taken to exchange one byte of data.) Grain size depends on the application itself and on the efficiency of the communication facility. For each number of processors, we consider the effect of different grain sizes of parallel mergesort (by modifying the extent of data decomposition) on the efficiency.

4 Results

The results are shown below in three graphs: Figures 4, 5, and 6. Figure 4 shows the scalability of the architecture; Figure 5 shows the effect of additional processors on the overhead of the generator; Figure 6 shows the effect of different granularity of parallelism for the parallel mergesort application on efficiency.

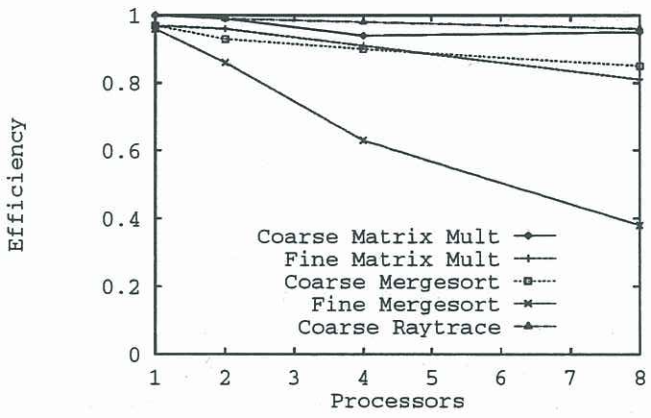

Figure 4: Effect of processors on efficiency

From Figure 4, we can generally infer that efficiency of a GLU application degrades with additional processors. The extent of degradation is insignificant when the granularity of parallelism is coarse as in coarse versions of raytracing and matrix product. In these two cases, the efficiency remains above 95% for upto eight processors. The extent of degradation

is noticeable when the granularity of parallelism is not as coarse as in the coarse version of mergesort and fine version of matrix product. In these two cases, the efficiency falls to 80% with eight processors. With (relatively) fine-grain mergesort, efficiency dips to under 40% with eight processors. We can thus infer that scalability of the basic architecture appears to be better when the granularity of parallelism is coarse. Of course, the actual grain size is influenced by particular implementations of the architecture. That is to say, for a workstation network implementation with relatively large communication latencies to be scalable will require coarser granularity than a shared-memory implementation with relatively small communication latencies.

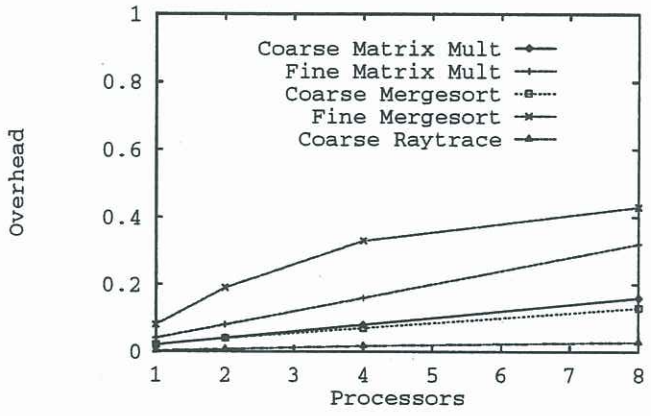

Figure 5: Effect of processors on generator overhead

Figure 5 explains why the scalability of the basic architecture is better with coarser granularity. The overhead incurred by the generator in terms of communications with executors grows as the number of processors increases. The extent of growth is much greater when the granularity of parallelism is small (as with fine-grain mergesort) than when it is large (as with coarse-grain raytrace).

From Figure 6, we can infer that the effect of differences in granularity is

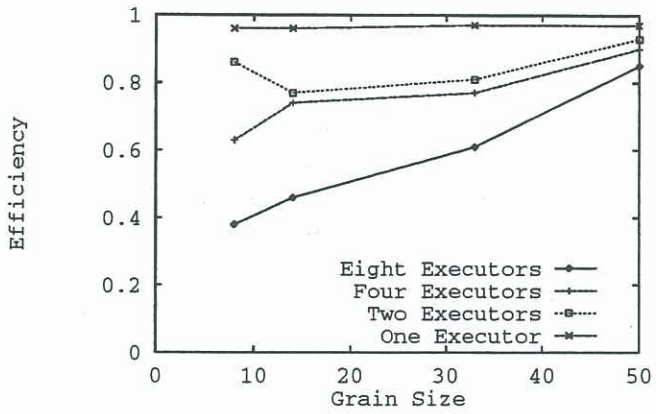

Figure 6: Effect of grain size on efficiency

more pronounced with large number of processors than with small numbers. For example, with two executors, the efficiency for mergesort is between 75–90% for different granularities whereas, with eight executors, the efficiency is less than 40% for relatively fine granularity of parallelism and over 80% for relatively coarse granularity of parallelism.

5 Conclusions

We have considered the performance of the basic virtual architecture for executing GLU programs in parallel as implemented on a workstation network. The experiments, while preliminary, have indicated that applications with coarser granularity of parallelism are likely to be more efficient than those with finer granularity of parallelism. Furthermore, applications with coarser granularity of parallelism are likely to sustain adequate efficiency over modest number of processors (workstations).

It is worth noting that the choice of grain size affects how much parallelism can be exploited and how much communications is required to do so. At one end, very coarse grain size will significantly diminish the amount

of parallelism while reducing communication requirements whereas, at the other end, very fine grain size will make significant amounts of parallelism available but at the expense of substantial communications. The "optimal" grain size depends on efficiency of the communications facility of the implementation relative to the computing power of individual processors. In the case of the workstation network, the communications facility is quite inefficient when compared to the computing power of individual processors; ergo the preference for coarser granularity of parallelism.

References

[1] E.A. Ashcroft. Tournament computations. In *Third International Symposium on Lucid and Intensional Programming*, Queen's University, Kingston, Ontario, Canada, 1990.

[2] E.A. Ashcroft, A.A. Faustini, and R. Jagannathan. Intensional parallel programming. In B.K. Szymanski, editor, *Parallel Functional Programming Languages and Environments*. ACM Press, 1991.

[3] A.A. Faustini and R. Jagannathan. Indexical Lucid. In *Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1991.

[4] R. Jagannathan and A.A. Faustini. The GLU Programming Language. Technical Report SRI-CSL-90-11, Computer Science Laboratory, SRI International, Menlo Park, California 94025, USA, November 1990.

[5] R. Jagannathan and A.A. Faustini. Tournament computations in GLU. In *Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1991.