GLU Implementation Architectures for Heterogeneous Systems

R. Jagannathan and C. Dodd
SRI International
Menlo Park, California 94025, U.S.A.
email: {jagan,dodd}@csl.sri.com
url: http://www.csl.sri.com/GLU.html

ABSTRACT

While a heterogeneous system offers substantial aggregate processing power to exploit parallelism in GLU programs, its processors and links exhibit fluctuating loads and can fail arbitrarily. On such a system, effective exploitation of coarse-grain parallelism requires the runtime strategy for load distribution to adapt to system dynamics. In this paper, we describe two eduction-based architectures for implementing GLU on heterogeneous systems that use different approaches to cope with dynamically fluctuating loads, processor failures, as well as processor additions. The first implementation architecture, which is based on multiple distributed generators, deals with fluctuating loads as well as processor changes using an adaptive value-demand distribution function. The second implementation architecture, which is also based on multiple distributed generators, uses dynamic inter-generator term-demand migration to adapt to fluctuating loads as well as processor changes.

1. Introduction

A typical heterogeneous system consists of a large collection of powerful and architecturally-dissimilar computers interconnected in a non-uniform manner. Heterogeneous systems are increasingly common — from a laboratory network of workstations running different flavors of Unix to a campus-wide network of PCs and workstations running different OS to the Internet itself.

The individual computers of a heterogeneous system typically exhibit fluctuating loads as they cater to a diverse set of both local and remote uses. In addition, these computers can fail, and new or previously failed computers can become available. Similarly, the network links connecting the various computers can exhibit fluctuating traffic loads or they can completely fail. Despite such dynamics, a heterogeneous system remains attractive because the aggregate computing capacity of its idle resources at any given time is quite substantial, far exceeding what may be available locally.

Most of the parallel programs have been developed using message-passing libraries such as PVM [4] and MPI.[5] These programs typically assume a static, homogeneous, and failure-free target system. Such programs have to be substantially revised or even rewritten to be able to effectively cope with the heterogeneity and dynamics found in a heterogeneous system. GLU [8,9] is a very high-level system for parallel programming in which computational functions are explicitly specified in C or Fortran while parallelism at the granularity of the computational functions is expressed implicitly in

Lucid.[2] GLU programs are architecture independent. In fact, they do not prescribe or assume *how* they are to be evaluated. Their functionality is independent of their many possible operational interpretations. It is this independence from operational semantics that enables several differing yet equally-correct implementations of GLU.

The eduction model of computation [1,3] is embodied in all existing implementations of GLU.[7,6,8,9] Eduction exploits parallelism implicit in Lucid programs, and it only exploits parallelism that is useful in computing the needed output. In addition, eduction can easily be extended to be fault tolerant.[10,2] The basic eductive implementation architecture of GLU is the centralized generator architecture (CGA) shown in Figure 1.

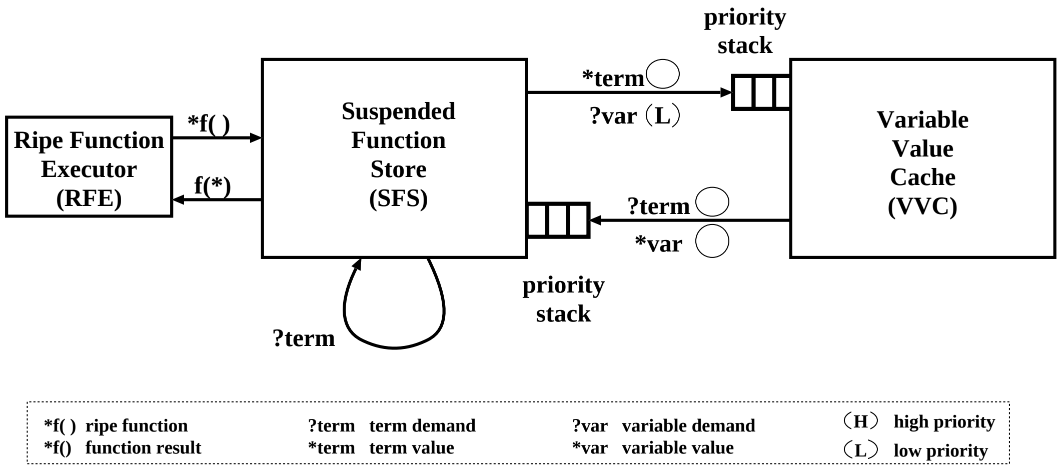

Fig. 1. Centralized Generator Architecture

The CGA consists of a single generator and several workers. The Lucid part of a GLU program is evaluated within the generator using eduction. While lightweight ripe C functions are evaluated within the generator itself, heavyweight ripe C functions are remotely executed at the workers. Parallelism is exploited to the extent different ripe functions are simultaneously executed on different workers, and to the extent the generator and a worker are simultaneously busy.

The generator consists of a suspended function store (SFS) and a variable value cache (VVC). The SFS is responsible for demand propagation. The VVC is responsible for retaining values of variables that have been already computed. The worker consists of the ripe function executor (RFE). The RFE is responsible for executing ripe functions.

The SFS sends demands for variable values (`?var`) to the VVC, and sends ripe C functions (`f(*)`) to the RFE. It internally processes demands for terms (`?term`).

The VVC sends demands for terms (`?term`) to the SFS. and variable values (`*var`) in response to demands for them to the SFS.

The RFE sends results of ripe C function execution (`*f()`) to the SFS in response to requests for ripe function execution issued by the SFS.

CGA is limited by number of workers (i.e., number of RFEs) that can exist without saturating the SFS. This mainly depends on granularity of parallelism being exploited i.e., the amount of time taken to execute the ripe C function at the RFE to the amount of time taken to transport the function arguments and function result between SFS and RFE. Coarser granularity of parallelism would allow for a larger number of workers to be supported by the generator than a finer granularity of parallelism. Regardless, the number of workers that a generator can support is bounded, beyond which performance scaling is not possible.

CGA naturally supports dynamic addition and deletion of workers (which contain RFEs). The ripe function distribution strategy at the SFS is "first-come first-serve" which usually results in good load balancing. However, load balancing can be poor when different workers operate at different speeds and the degree of parallelism is limited. In particular, the total execution time can lengthen with an unusually slow worker than without it.

While the CGA can tolerate failure of workers by reissuing ripe functions to other workers, it cannot tolerate the failure of the generator.

Any implementation of GLU, such as the one based on CGA, needs to be evaluated in terms of (a) how scalable it is, and (b) how adaptive it is. A GLU architecture is considered scalable if a linear increase in application performance can be derived by a linear increase in the architectural complexity. In particular, the scalability of CGA is limited as linear increase in the number of RFEs will not continue to result in linear increase in application performance as the generator is the bottleneck. A GLU architecture is considered adaptive if it can effectively tolerate addition and deletion of architectural elements without adversely affecting application performance. In particular, the adaptivity of CGA is limited as the failure of the generator results in total failure.

We now consider a family of GLU implementation architectures that are all based on the use of distributed generators to achieve both scalability and adaptivity.

2. Distributed Generator Architecture

The idea behind the "generic" distributed generator architecture (DGA) is that the generator itself is replicated. That is, there are multiple instances of SFS and VVC. As before, associated with each SFS are multiple instances of RFE corresponding to multiple workers associated with a generator. Scaling is achieved by adding generators, as well as adding workers to individual generators. The multiple generators provide for replication that aids adaptivity. Figure 2 illustrates the relationship between the SFSs and VVCs in the DGA. Eductive evaluation is distributed across the multiple generators in one of two ways:

1. Distributing *value demands* to different VVCs for processing and subsequent retrieval.

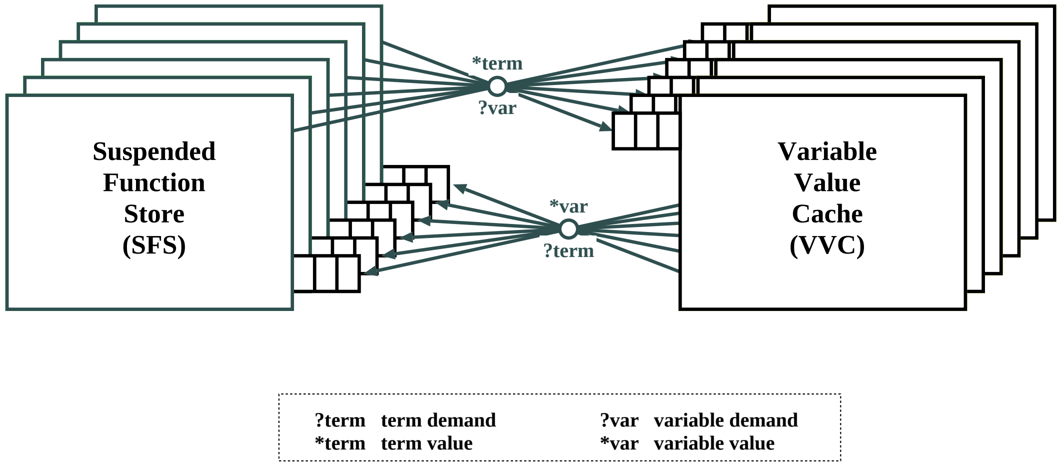

Fig. 2. Distributed Generator Architecture

2. Distributing *term demands* to different SFSs for further processing.

The DGA is the basis for several implementation architectures, each of which are designed to achieve scalability and adaptivity in different ways. The two distributed generator architectures that we discuss are the static DGA (SDGA) and the dynamic DGA (DDGA).

3. Static DGA

The basic idea (Figure 3) in the SDGA is that value demands are distributed to VVCs in different generators based on the context associated with the demands. Terms demands are processed within the generator itself.

The mapping from the value demand context to a specific VVC consists of an initial static part and a subsequent dynamic part (Figure 4). The initial part of the mapping, which is associated with the variable whose value is being demanded, maps the context of the value demand to an index in the virtual VVC vector. The second part of the mapping, which is dynamically determined at runtime, maps the index to a specific VVC. Assuming that the size of the virtual VVC vector is much larger than the number of VVCs, the effect is that different VVCs, at any given time, are allocated different (possibly unequal) “chunks” of the virtual VVC vector.

When a generator fails, its loss is initially detected by all generators which have an outstanding value demand being processed in the failed generator. (Eventually, all generators are informed of the loss of a given generator.) The second part of the value demand mapping is dynamically adjusted so that the indices of the virtual VVC vector assigned to the failed generator are reassigned to other generators. The same adjustment is simultaneously made at every generator. Generators with outstanding value demands to the failed generator reissue the demands using the latest mapping.

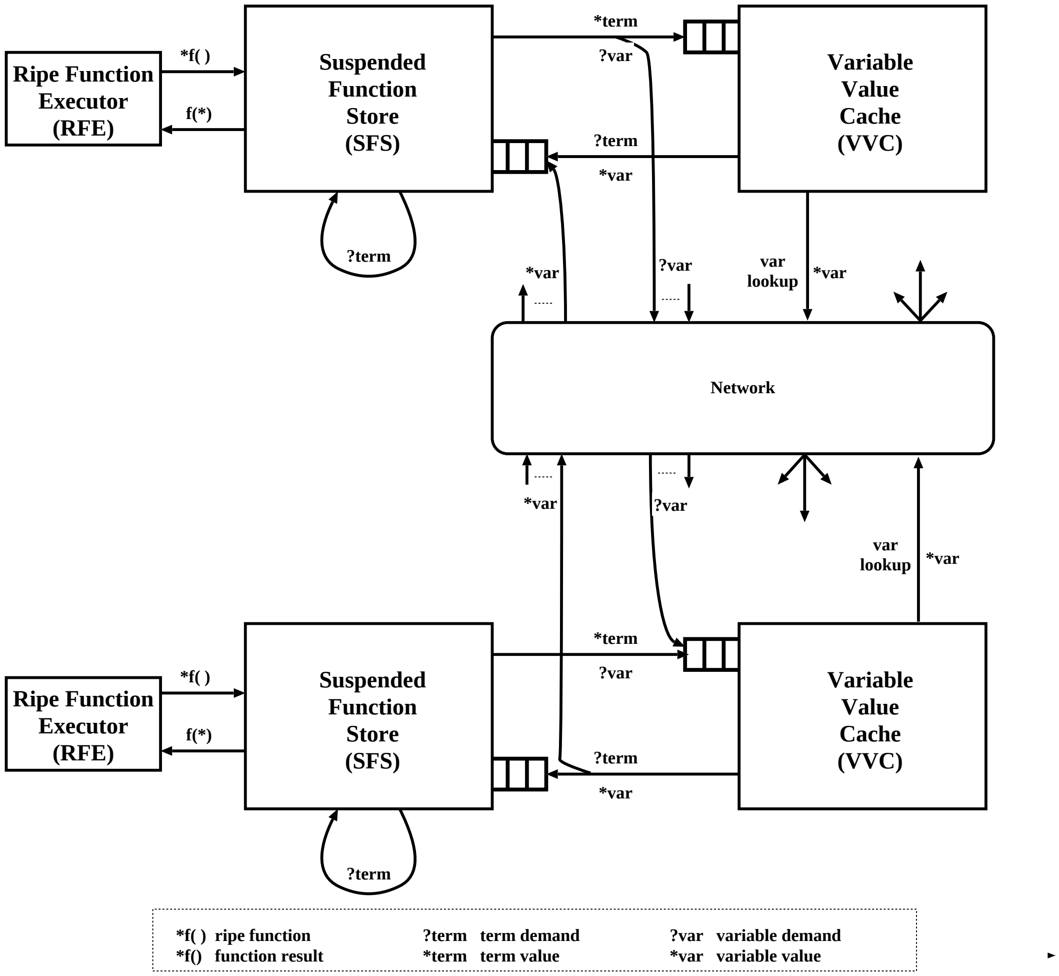

Fig. 3. Static Distributed Generator Architecture

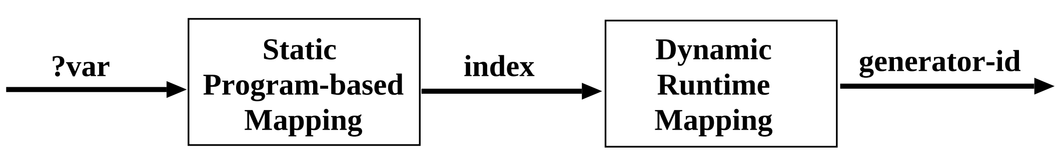

Fig. 4. SDGA Value Demand Mapping

When a generator is added, the addition is detected by all other generators. The detection causes the dynamic part of the value demand mapping to be reconfigured by redistributing the indices to the new set of generators.

The main effect of reconfiguring the dynamic part of the value demand mapping is on the way in which variable values are computed and retrieved. Given a value demand, the latest value demand mapping is used to determine the generator where the value is expected to be stored. If the value is not there, it either means that the demanded value has never been computed or that the demanded value has been computed on another generator as determined by an earlier version of the dynamic mapping. The value demand is multi-cast to all generators where the value could have been stored, using each earlier version of the dynamic mapping. Each generator consulted this way either returns the value if it has it or it returns a negative acknowledgment otherwise. If all generators return a negative acknowledgment, the value is computed afresh in the generator as given by the latest mapping.

By using a hybrid of static and dynamic mapping of value demands, SDGA takes advantage of program as well as runtime information to achieve load balancing. In particular, the number of index chunks assigned to different generators can take into account their relative capacities and proximities on an ongoing basis. However, a poor programmer-provided static mapping could result in considerable load imbalances between generators.

While the SDGA can deal with dynamic additions and deletions of generators, the number of such events has an effect on its efficacy. In particular, the number of generators that need to be consulted during a value lookup is proportional to the number of times the dynamic part of the value demand mapping has been reconfigured, which equals the number of generator additions and deletions. In the worst case, all generators have to be consulted for a value lookup, which limits the scalability of the SDGA.

One promising approach to limit the number of generators that are looked up is to move values into the "right" VVCs as per the latest value demand mapping. In particular, the ongoing garbage collection activity can be augmented with the task of moving values in the VVC of a local generator to the VVC of the appropriate generator. This would have the effect of reducing the number of multi-cast lookups of variable values.

4. Dynamic DGA

With the DDGA, all demands are distributed dynamically, based on what is currently happening in all the generators (Figure 5).

Term demand distribution is the primary mechanism by which load is distributed and balanced in the DDG. As long as all generators are busy, there is no need to distribute term demands to balance the load. When a generator is idle, demand(s)

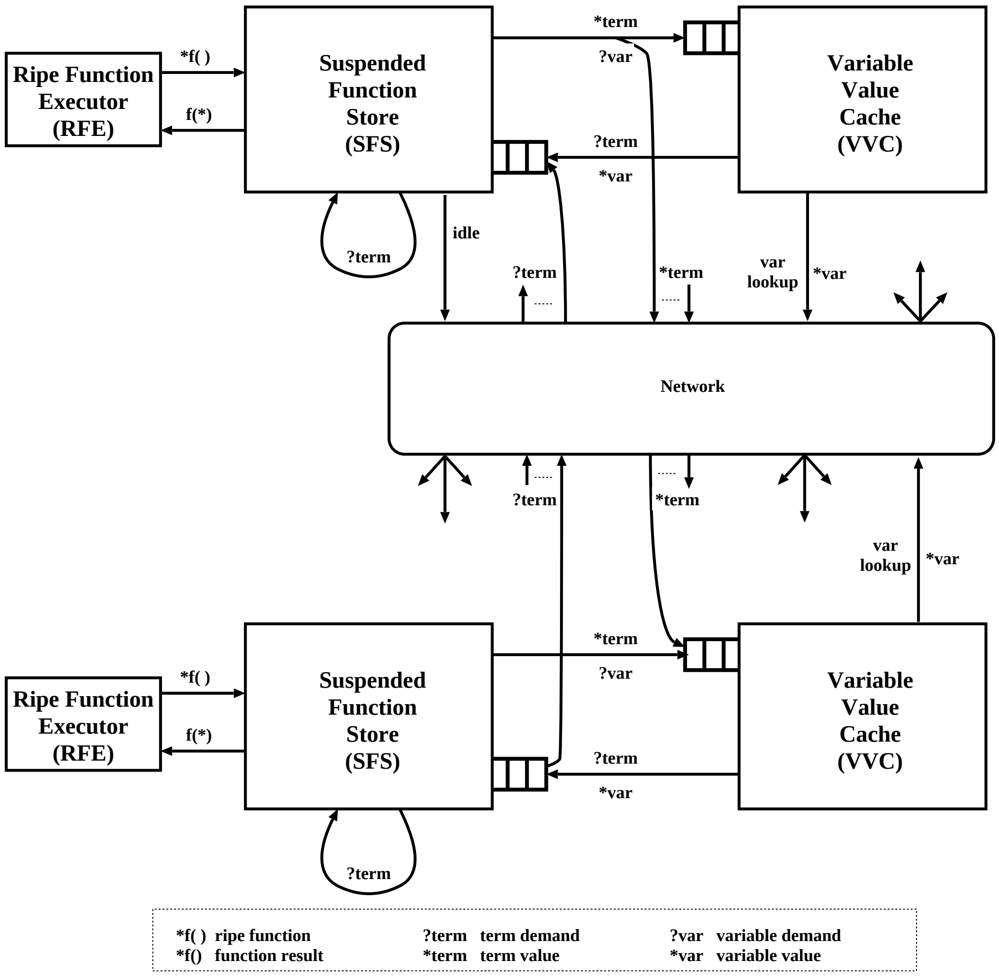

Fig. 5. Dynamic Distributed Generator Architecture

must be sent to it so as to keep it busy. We call this "demand migration", as the idle generator needs to go off and find some other busy generator to get demands from.

There are several possible policies to choose from when migrating demands. A generator may choose some other generator at random to steal a demand from, or it might attempt to find the longest queue and steal from that. It could also advertise the fact that is idle by group broadcast, causing busy generators to send demand(s) to it.

Since term demands can be moved about at any time, there is no global knowledge in the system about where variables are cached. As a result, accurate handling of variable demands will require a global broadcast lookup, which is quite expensive. Fortunately, failure to correctly correctly lookup a variable demand in all generators will only result in recomputing the value; inefficient but still correct.

In order to keep down network traffic, the DDGA makes a random guess for each variable demand whether or not the value is likely to reside in some remote generator. Some fraction of the time, it will broadcast the variable demand to all generators. The rest of the time, it will handle the demand locally, possibly recomputing the value.

The probability of the random choice to broadcast the demand or not must be carefully tuned, on a per variable basis. If the cost of recomputing the variable is high (much higher than a broadcast), it should always be broadcast. If it is cheap, it should rarely be broadcast. It is also desirable to tune the choice based on network load. If too many broadcasts are making the network too busy, the probability should be reduced. If the network is idle and has spare capacity, the probability can be increased.

When a generator fails, it is necessary to regenerate all demands that were stolen by that generator and not yet satisfied. In order to do this, every generator must keep track of all stolen demands and, when a failure is detected, must regenerate those demands.

The scalability of the DDGA is highly dependent upon the scalability of the underlying broadcast/multi-cast functionality provided by the network. As more generators are added to the system, they will add additional multi-cast load to the network, so network capacity also needs to be increased to avoid a network bottleneck. By controlling the broadcast lookup probability, it is possible to keep the network load down, but this will result in additional redundant computation, limiting scalability. This is essentially a tradeoff between available network capacity and available processing power.

The DDGA is capable of adapting to the addition and loss of generators at any time, and such changes in the system should not have a undue effect on performance. When a generator is lost, any data stored only in that generator will need to be recomputed, but changes in the system will not have any effect on the lookup of variables in the VVC, unlike with the SDGA.

5. Conclusions

We have considered two different eduction-based distributed generator implementations of GLU for heterogeneous systems – static distributed generator architecture (SDGA) and dynamic distributed generator architecture (DDGA). In the SDGA, value demands are explicitly distributed to various generators. In the DDGA, terms demands are implicitly distributed to various generators. The SDGA copes with dynamic generator addition and deletion by revising the load distribution function to reflect the new configuration, and maintaining a history of past configurations. The DDGA copes with dynamic generator addition and deletion in its implicit distribution of load (using demand stealing).

The SDGA requires programmer guidance in achieving even distribution of variable values. The extent to which such an architecture will scale is dependent on the programmer's ability to select suitable distribution functions for variable values. The SDGA uses multi-cast communication to avoid recomputation of values that are available elsewhere.

The DDGA can achieve very good load balancing through its use of demand migration between generators to distribute load. The architecture trades off inter-generator communications with value recomputation and replication.

6. Acknowledgments

This work was supported by the U.S. Defense Advanced Projects Agency (AO-C367) and by the U.S. National Science Foundation (CCR-9203732).

7. References

1. E.A. Ashcroft. Dataflow and eduction: Data-driven and demand-driven distributed computation. In *Current Trends in Concurrency.* Springer Verlag, 1986. Lecture Notes in Computer Science (224).
2. E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Programming.* Oxford University Press, 1995.
3. A.A. Faustini and W.W. Wadge. An eductive interpreter for pLucid. In *Proceedings of the 1987 SIGPLAN Conference on Interpreters and Interpretive Techniques*, pages 86–91, Minneapolis, 1987.
4. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

5. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.
6. R. Jagannathan. Toward a scalably efficient parallel implementation of a hybrid intensional language. In *Second International Conference on High Performance Computing*, New Delhi, India, December 1995. Tata McGraw Hill.
7. R. Jagannathan and C. Dodd. Preliminary performance of a parallel GLU implementation on a network of workstations. In *Fifth International Symposium on Lucid and Intensional Programming*, Oakland, California, 1992.
8. R. Jagannathan and C. Dodd. GLU Programmer's Guide (Version 1.0),. Technical Report SRI-CSL-96-06, Computer Science Laboratory, SRI International, Menlo Park, California, July 1996.
9. R. Jagannathan, C. Dodd, and I. Agi. GLU: A high-level system for granular data-parallel programming. *Concurrency: Practice and Experience*, 1996.
10. R. Jagannathan and A.A. Faustini. Tournament computations in GLU. In *Fourth International Symposium on Lucid and Intensional Programming*, SRI International, Menlo Park, California, 1991.