Implementing GLU on Dynamic Heterogeneous Systems

R. Jagannathan and C. Dodd
SRI International
Menlo Park, CA 94025

Extended Abstract

Abstract
We consider the problem of efficiently implementing GLU on a dynamic heterogeneous system in which processors and links may arbitrarily fail, or exhibit fluctuating capacities. We propose two distinct approaches to solving this problem. The first approach extends the static distributed-generator implementation model originally designed to balance load across multiple generators, to adapt at runtime to generator additions and deletions. The second approach proposes a dynamic distributed-generator implementation model that achieves both load balancing and adaptation by using the notion of "demand stealing." We compare the two implementation models and discuss some of the issues that need to be addressed for their effective realizations.

1 Introduction
A heterogeneous system, of which the Internet is an extreme example, consists of a collection of powerful and architecturally-dissimilar computers interconnected in a non-uniform manner. The individual computers 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. Despite the considerable and complex dynamics, a heterogeneous system is attractive simply because the aggregate computing capacity of its idle resources at any given time is substantial, far exceeding what may be available locally.
The challenge is how one goes about writing an application for such a system without being burdened by the need to balance load and the need to continually adapt.

GLU is a hybrid intensional language for parallel programming that unburdens the programmer from explicit parallelism management, including load balancing [2]. The programmer simply selects a standard parallel program architecture (PPA) to which the GLU program is then compiled. Built into the PPA is a specific strategy for parallelism management given that parallelism in a GLU program is discovered using the eduction model of computation.

The most basic PPA is the centralized generator (CG) PPA which consists of a single generator that embodies eduction, and multiple workers each of which execute external functions of the GLU program. CG-PPA is suited for GLU programs with coarse granularity of parallelism and only nominal communications running on target systems with a limited number of processors. The problem with the CG PPA is that the single generator limits its scalability, i.e., performance does not improve beyond a certain number of workers as determined by the granularity of parallelism of the program [1].

The distributed generator (DG) PPA addresses the scalability limitation of the CG PPA by employing multiple generators which work together to eductively evaluate the program. Thus, each generator is only responsible for some part of the eductive evaluation, the part depending on the program being evaluated. Specifically, the value cache which in CG-PPA was retained in the single generator is distributed across the multiple generators. Each variable value is stored in a generator which is determined by the distribution scheme applied to the name of the value. Demands for variable values are routed to the appropriate generator, where the value is stored when it is computed. Demands for term values are computed in the same generator where they are issued. Each generator has its own set of workers (either multithreaded in the same address space or as separate processes interacting via shared global memory). One distribution scheme is to apply a uniform hash function to the variable-value-name to select a generator. Another distribution scheme is to have the programmer specify the function to select a generator as an annotation associated with the variable definition. (Figure 1 shows the interaction between the generators in the DG-PPA.)

DG-PPA is designed to be more scalable than CG-PPA, and its ability to balance load depends on the distribution function. However, DG-PPA assumes that the number of generators is fixed at the start of
Figure 1: Distributed Generator PPA
execution and stays that way until execution ceases. Thus, DG-PPA cannot tolerate failure of individual generators, nor can it adapt to addition of new generators.

2 Adaptive DG-PPA with Static Load Balancing

We will refer to the DG-PPA discussed earlier as DG-PPA with static load balancing or simply as DG-PPA\textit{(static)} to distinguish it from another adaptive DG-PPA that we introduce in the next section. The adaptive DG-PPA\textit{(static)} (or ADG-PPA\textit{(static)}) an extension of the DG-PPA\textit{(static)} that tolerates dynamic addition and deletion of generators.

Consider a GLU program executing using the DG-PPA\textit{(static)} on say $N$ generators, and consider the deletion of a generator, say generator $i$. All variable values which were retained at generator $i$ are no longer available. These lost values would have to be recomputed at another generator. If we assume that all generators can detect the deletion of a generator, then whenever a generator requires a variable value from generator $i$, it can use a second distribution function to determine another generator. (This is akin to collision resolution in conventional hashing.) If all generators use the same second distribution function, then the lost values will be recomputed only once at appropriate generators. In this way, deletion of additional generators can be tolerated.

This approach does not work when generators are added. For example, if there are $N$ generators when execution starts, and an additional generator is introduced, say generator $N + 1$, the distribution function has to account for the additional generator. Instead of using modulo $N$, the distribution function has to use modulo $N + 1$ to select from $N + 1$ generators. This implies a different mapping of variable values to the $N + 1$ generators than with the original $N$ generators. In particular, most of the variable values will end up being recomputed as they will be in generators that the new distribution function will miss. One solution to avoid recomputation is to “reshuffle” the variable values across generators as per the new distribution function. This is likely to be prohibitively expensive specially if generators are added quite frequently.

We propose a solution that avoids recomputation of already-available variable values. The idea is to send the demand for a variable value not just to a single generator but to all generators that would have been selected by each incarnation of the distribution function. If one of them has the value, it will be returned. If
none of them has the value, the generator selected using the latest incarnation of the distribution function computes the value. Note that absence of value can be explicitly determined by requiring all generators that do not have a demanded value to return a negative acknowledgement. Alternatively, absence of a value can be implicitly determined by using timeouts. For example, in the case that a single generator is added to $N$ generators, demand for a value is sent to generators selected using modulo $N$ and modulo $N + 1$. If the value has already been computed and available, it will be returned by one of the generators. Otherwise, the value is computed by the generator specified by the distribution function modulo $N + 1$.

3 Adaptive DG-PPA with Dynamic Load Balancing

We propose a DG-PPA that uses a dynamic technique for distributing variable values to various generators. The interesting aspect of this PPA is that the same technique inherently facilitates adaptation to addition and deletion of generators.

We produce the DG-PPA(dynamic) by parallelizing each component of the eduction engine. We loosen behavioral constraints of the component parts, taking advantage of the characteristics of eduction, to minimize the amount of global communication required.

The basic idea is that demands for variable values are enqueued in the local generator, i.e., in the generator issuing the demand. (Note that demands for terms are as usual processed locally.) The variable value manager dequeues a demand from its local queue for further processing unless the queue is empty. In this case, the demand is “stolen” from a queue in another generator that is non-empty. The value being demanded, if computed afresh, is stored locally.

When a demand for a variable value is processed by the local VVM, it is first looked up locally. If it is locally unavailable, its computation is conditionally started pending a lookup in each of the other generators. This requires the use of a broadcast or multicast mechanism. If the value is found in some other generator, the local processing of the demand is aborted. If the value is not found, local processing continues until the value is computed.
4 Issues

It should be evident how each of the proposed PPAs adapt to generators being added and deleted while a program is being executed. Some of the issues that need to be addressed for effective realization of the proposed implementation models — ADG-PPA(static) and ADG-PPA(dynamic) — include the following.

4.1 Global Lookups versus Redundant Computation

Both PPAs use broadcast (or multicast) to lookup variable values in remote generators. With ADG-PPA(static), the multicast group is limited to the number of different generators specified by the various incarnations of the distribution function. Thus, if generator additions and deletions is infrequent, the size of the group will not be large. With ADG-PPA(dynamic), the multicast group is all the generators as any of the generators could have stolen the original demand. Thus, the effect of increased communications due to multicasting is more prominent.

Careful tuning of the conditions under which global lookups are done is required to balance between expensive global lookups and redundant computation.

One approach to reducing communications with both PPAs is to allow some redundancy in values computed. For example, the size of the group can be bounded to reduce traffic while risking recomputing values that might already be available in other generators.

Another approach to reducing communications with both PPAs is to only broadcast demands of values that are expensive to recompute. Other demands can be recomputed in the local generator itself.

4.2 Balancing Load Distribution

ADG-PPA(dynamic) inherently results in better distribution of load when compared to ADG-PPA(static). With ADG-PPA(dynamic), it is important to determine the generator from which demand should be stolen, as this choice will have an effect on load balancing. With ADG-PPA(static), the load distribution is at the mercy of the programmer-determined distribution function (or in the uniformity of the hash function). It would be useful for the compiler to suggest distribution functions based on an analysis of the thr program.
4.3 Generator-Worker Interaction

As each generator is assumed to have its own set of dedicated workers, it is important that the latency between the generator and worker is kept to a minimum. Assuming that the generator runs on a multiprocessor, the workers and generator can interact via shared memory. It would be even more efficient for the workers to manifest as threads within the generator.

References
