Managing Heterogeneity in a Grid Parallel Haskell

A. Al Zain[1], P. W. Trinder[1], H-W. Loidl[2], and G. J. Michaelson[1]

[1] School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K. {ceeatia,trinder,greg}@macs.hw.ac.uk

[2] Ludwig-Maximilians-Universität München, Institut für Informatik, D 80538 München, Germany, hwloidl@informatik.uni-muenchen.de

Abstract. `Grid-GUM` is a distributed virtual shared-memory implementation of a high-level parallel language for computational Grids. While the implementation delivers good speedups on multiple homogeneous clusters with low-latency interconnect, on heterogeneous clusters, however, poor load balance limits performance. Here we present new load management mechanisms that combine static and partial dynamic information to adapt to heterogeneous Grids. The mechanisms are evaluated by measuring four non-trivial programs with different parallel properties, and show runtime improvements between 17% and 57%, with the most dynamic program giving the greatest improvement.

1 Introduction

Hardware price/performance ratios and improved middleware and network technologies make cluster computing and computational Grids increasingly attractive. These architectures are typically heterogeneous in the sense that they combine processing elements with different CPU speeds and memory characteristics. Parallel programming on such heterogeneous architectures is more challenging than on classical homogeneous high performance architectures.

Rather than requiring the programmer to explicitly manage low level issues such as heterogeneity we advocate a high-level parallel programming language, specifically GpH, where the programmer controls only a few key parallel coordination aspects. The remaining coordination aspects including heterogeneity are dynamically managed by a sophisticated runtime environment, GUM. GUM has been engineered to deliver good performance on classical HPCs and clusters [6]. `Grid-GUM` is a port of GUM to computational Grids using the Globus Toolkit, the de-facto standard. As `Grid-GUM` implements a virtual shared-memory over a wide-area network, it is perhaps surprising that it gives good performance in some instances, e.g. on homogeneous low-latency multiclusters [1]. However for heterogeneous architectures load management emerges as the performance-limiting issue.

This paper presents `Grid-GUM2`, incorporating new load management mechanisms for heterogeneous architectures. The new mechanisms are decentralised, obtain complete static information during start up, and then cheaply propagate partial dynamic information during execution. The existing load management techniques in `Grid-GUM` are described in Section 2. The design of the new thread, load and communication management mechanisms is given in Section 3. The effectiveness of the new mechanisms for heterogeneous clusters is investigated using four non-trivial programs from a range of application areas, and with varying degrees of irregular parallelism and using both data parallel and divide-and-conquer paradigms in Section 4. Related work is discussed in Section 5, and we conclude in Section 6.

2 GpH and Grid-GUM

GpH (*Glasgow parallel Haskell*) [7] is a modest and conservative extension of Haskell 98 with very high level coordination, i.e. control of parallel execution. Parallel and sequential composition primitives introduce and synchronise threads. Evaluation strategies are polymorphic higher-order functions that abstract over the primitives to provide parameterisable, reusable high level coordination control.
Grid-GUM `Grid-GUM` extends the existing `GUM` memory management, and thread management techniques. In particular, it implements a virtual shared heap over a wide-area network [8]. The communication management in `Grid-GUM` is similar to `GUM`, but uses MPICH-G2, a Grid-enabled implementation of the MPI standard [5], and hence the Globus Toolkit as middle-ware.

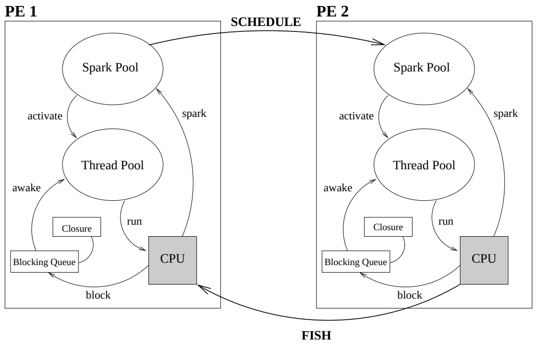


Fig. 1. Interaction of the components of a GUM processing element.

`Grid-GUM` uses an *evaluate-and-die* thread management model with *sparks*, i.e. pointers to graph structures, to represent potential parallelism. Sparks are generated by an explicit `par` construct in the program and maintained by the runtime-system in a flat *spark pool*. A sparked expression may be executed by an independent thread. However, if a thread needs the value of the expression, and no other thread is evaluating it, the demanding thread will perform the computation itself. This behaviour is called *thread subsumption* because the potentially parallel work is inlined by another thread.

Figure 1 illustrates the load management mechanism in `Grid-GUM`, depicting the logical components on each Processing Element (PE) of the `Grid-GUM` abstract machine. When activated a spark causes a new thread to be generated. Threads that are not currently being executed reside in the thread pool. When the CPU is idle, and the thread pool is empty, a spark will be activated to generate a thread. If a running thread blocks on unavailable data, it is added to the blocking queue of that node until the data becomes available.

The thick arrows between the PEs in Figure 1 show load management messages exchanged in `Grid-GUM`. Initially all processors, except for the main PE, will be idle, with no local sparks available. PE2 sends a `FISH` message to a random-chosen PE. On arrival of this message, PE1 will search for a spark and, if available, send it to PE2. This mechanism is usually called *work stealing* or *passive load distribution*, since an idle processor has to ask for work. `Grid-GUM` also improves load distribution by using *Limited Thread* mechanism which includes specifying a hard limit on the total number of live threads, i.e. runnable or blocked threads in the thread pool. To summarise, the `Grid-GUM` load mechanism deals with both locating work (Figure 2.a), and handling work requests (Figure 2.b), where these activities are performed in the main scheduler loop between thread time slices.

```
IF idle THEN
  IF runnable thread THEN
    execute runnable thread
  ELSE
    IF spark in spark pool then
      create runnable thread
      execute runnable thread
    ELSE
       send fish to random PE
```

Fig 2.a. Work locating

```
IF received fish THEN
  IF sparks available THEN
    send spark
      to fishing PE
  ELSE
    forward fish to random PE
```

Fig 2.b. Work request handling

`Grid-GUM` **performance** In earlier work we have investigated the performance of GpH programs on `Grid-GUM`, with varying numbers of clusters and interconnection latencies [1]. We find that `Grid-GUM` delivers good and predictable speedups for several configurations, e.g. multiple homogeneous clusters with a low-latency interconnect, or programs that perform little communication on multiple clusters with a high latency interconnect. However other configurations, including heterogeneous clusters, `Grid-GUM` gives poor performance due to poor load management.

3 Grid-GUM2

Process and thread scheduling methods are typically classified into several subcategories as depicted in figure 2. *Local* scheduling performed by the operating system of a PE consists of the assignment of processes to the time slice of the PE. *Global* scheduling, in contrast, deals with the execution of a set of processes in a multiprocessor system. In *static* scheduling, PEs are assigned tasks before the execution begins. Unlike local scheduling, *dynamic* relies on the assumption that limited knowledge about the processing resources is available *a priori*, thereby, no scheduling decision is made until the execution begins. Dynamic scheduling is subdivided into two categories: a) *centralised* where a single PE is responsible for scheduling, and b) *distributed* where all PEs participate in the scheduling. Distributed techniques can either operate in a *passive* mode where idle PEs request work, or on an *active* mode where PEs are assigned work without a prior request. This section presents `Grid-GUM2`, a new adapted load scheduling to

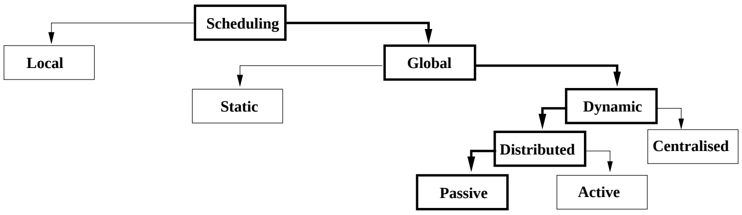


Fig. 2. `Grid-GUM2` Load Scheduling Classification

improve `Grid-GUM` performance. `Grid-GUM2` utilises static and dynamic information in a global dynamic scheduling using a distributed and mostly passive mode. The salient features of `Grid-GUM2` are:

Targeted Load Management: The challenge in load managment is to efficiently and effectively distribute the available work using dynamic information about PE CPU speed and loads, with minimal overheads. In `Grid-GUM`, if a PE is idle, it sends a *FISH* message to a randomly-chosen PE. However, given the increased heterogeneity of computational Grids, `Grid-GUM2` sources work from a PE that has a high load relative to its' speed.

Load information must be maintained dynamically, but it is prohibitively expensive to regularly broadcast complete load information to every PE in a large computational Grid. `Grid-GUM2` uses a lightweight approach maintaining partial dynamic PE information at almost no cost. Each PE maintains a table *PEDynamic* that contains partial timestamped dynamic information about the PEs, currently just their loads as depicted in Figure 4. The PEDynamic table is included with every message sent between PEs, and the receiving PE updates its table with more recent information. The additional cost of sending the table is minimal for the high bandwidth interconnects in typical computational Grid.

Specialised Thread Management: `Grid-GUM` has a parameter that specifies the minimum number of sparks that should be available in each PE, the *low-watermark*. If the number of sparks in a PE falls below the low-watermark it issues a FISH message to obtain more work. Such a uniform strategy for all PEs must be adapted for heterogeneous clusters. Our design is to maintain a local low-watermark for each PE, with higher watermarks for fast PEs. This encourages fast PEs to be more aggressive about obtaining work than slower PEs. However, local low watermarks are not get implemented in the system measured in the next section.

Variable Latency Communication Management: The interconnects between PEs in a computational Grid have varying static and dynamic latencies, e.g. the latency between PEs in the same cluster is much less than that between PEs on different clusters, and both are affected by network traffic. Our

PE	CPU Speed	Time Stamp
A	550 MHz	13:40:01
D	550 MHz	13:45:00
C	350 MHz	12:40:03
B	350 MHz	13:44:03
F	350 MHz	14:40:03

Fig. 3. PEStatic Table

PE	Load	time_stamp
A	2000	14:13:49
D	3000	14:13:59
B	10000	14:12:22
⋮	⋮	⋮

Fig. 4. PEDynamic Table

PE	Latency	Last Update
F	0.75 msec	12:45:20
G	2.05 msec	12:24:50
C	10.00 msec	12:50:25
⋮	⋮	⋮

Fig. 5. Communic. Table

mechanism prefers to obtain work and hence data from PEs that currently have low communication latency. `Grid-GUM2` maintains communication information at almost no cost. Each PE has a table *Communication* that contains partial timestamped dynamic information about the communication, currently just the latency to each PE as depicted in Figure 5. The latency of every message is estimated, timestamped and recorded in the Communications table with the startup messages initialising the table. The problems of maintaining synchronised distributed clocks is minimised by measuring message send and recieve times as time elapsed since the program start.

```
IF idle THEN
 IF sparks < local-watermark THEN
   send fish+local data
     to busiest PE from tables
  IF runnable-thread THEN
    execute runnable-thread
  IF spark in the spark-pool THEN
    create runnable-thread
    execute runnable-thread
  ELSE
    send fish+local data
      to busiest PE from tables
```

Fig 3.a. Work locating

```
IF received fish THEN
  update tables with data
    from fishing PE
  IF sparks available THEN
    IF fishing PE is local THEN
      send spark in schedule
        to fishing PE+local data
    Else
      send spark(s) in super-schedule
        to fishing PE+local data
  ELSE
    IF another PE has spark
      forward fish+local data
        to busiest local PE
```

Fig 3.b. Work request handling

`Grid-GUM2` introduces a novel mechanism to deal with cases where there is no work available on local, i.e low-latency, PEs. When an idle PE requests work from a PE residing outside its cluster, then a scheduling decision is automatically taken on the basis of minimising intra-cluster communication. One can distinguish two scenarios: a) if the request originated from relatively powerful cluster, then multiple sparks, i.e. several work items, are returned in a *Super-*

Schedule message; b) if the request originated from a relatively weak cluster then the request is served as usual, i.e. as a single spark. Note that the idea of `Super-Schedule` is not evaluated in this work since it is under implementation.

The core of `Grid-GUM2` load management can be summarised as work location (Figure 3.a), and work request handling (Figure 3.b).

4 Performance Comparison

Program	Application Area	Paradigm	Regularity	Runtimes & Improvement %		
				`Grid-GUM` Basic	`Grid-GUM` + Thread Limi.	`Grid-GUM2`
`queens`	AI	Div-Conq.	Regular	668	333 50%	310 53%
`sumEuler`	Nume. Analysis	Data Para.	Limit irreg.	570	343 39%	279 51%
`linSolv`	Symb. algebra	Data Para.	Limit irreg.	217	306 −40%	180 17%
`raytracer`	Graphic	Data Para.	High irreg.	1340	814 39%	572 57%

Table 1. Programs characteristics and performance

Table 1 compares the runtime for four GpH programs with different paradigms, and parallelism regularities, on multiple heterogeneous clusters with moderate latency interconnect. The four programs are measured in this experiment are: `queens` which places chess pieces on a board; `sumEuler` which computes the sum of the Euler totient values of a list; `linSolv` which finds an exact solution of a linear system of equations; `raytracer` which calculates a 2D image of a given scene of 3D objects by tracing all rays in a given grid, or window. All run-times are the median of three executions to ameliorate the impact of the operating system and shared network interaction. The experiments were performed on eight machines: four machines with fast CPU speed (1395 MHz) and four with slow CPU speed (534 MHz).

Overall, the dynamic adaptive scheduling of `Grid-GUM2` consistently shows the best performance on heterogeneous Grid multi-clusters.

Due to limited space we focus on invstigating the behaviour of one benchmark program, `raytracer`. In particular, `raytracer` has highly irregular execution, and consequently is very sensitive to changes in parallel environment.

Figure 6 compares the runtime achieved for the `raytracer` using different load management mechanisms on a heterogeneous environment, where there are the same numbers of slow and fast machines. For more than 4 PEs the best results are achieved for `Grid-GUM2` with better scalability and decreasing runtime than basic `Grid-GUM` and `Grid-GUM` with thread limitation.

Figure 7 shows per-PE activity profile for `raytracer`, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). A per-PE activity profile shows the behaviour for each of the PEs (y-axis) over execution time (x-axis). Each PE is visualised as a horizontal line, with darker shades of gray indicating a larger number of runnable threads. Gaps in the horizontal lines

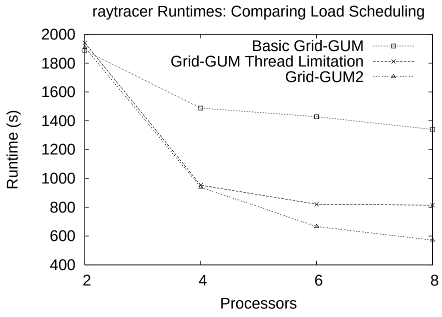
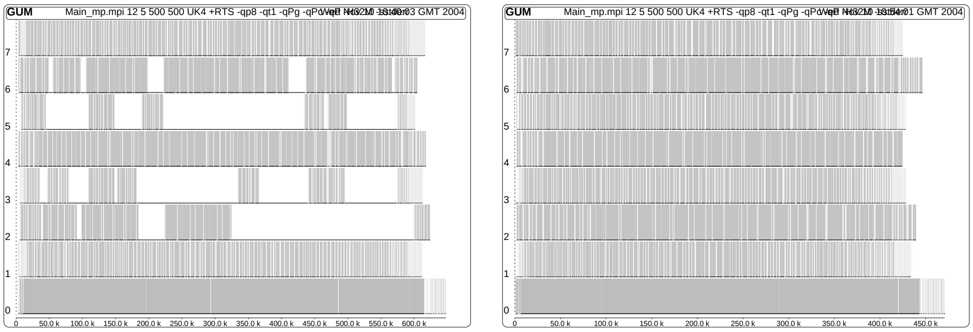


Fig. 6. Runtimes for `raytracer`

indicate idleness. Figure 7.a depicts the performance on `Grid-GUM` using a limit



a) `Grid-GUM` **b**) `Grid-GUM2`

Fig. 7. per-PE activity profile for `raytracer`

of 1 for the number of live threads. This small limit is chosen because it is the only one that has any impact on a heterogeneous architecture. Figure 7.b depicts the performance on `Grid-GUM2`. All PEs in Figure 7.b are uniformly loaded, and finish at the same time, in contrast the PEs in Figure 7.a have numerous idle periods. Figure 7.a also shows long idle periods at the middle of the computation, where only a small amount of parallelism is available. Here, blocking on data that is evaluated in the PEs with slow CPU speed will cause the PEs with fast CPU speed to remain idle until new work is obtained. Note that the runtime drops from 814 s to 572 s, which almost (30%) improvement.

5 Related Work

The most closely related work is the ConCert system [4], which translates a subset of ML to machine code, for execution on a Grid architecture. In contrast to our work, parallelism is expressed via explicit synchronisation. Alt *et* *al* apply skeletons to computational Grids [3]. This work focuses on providing the application user with skeletons to capture common patterns of Grid abstractions. However, our aim is to provide more general programming language support for parallelism through an implementation that incorporates new implicit dynamic coordination-management strategies. Aldinucci *et al* also apply skeletons to computational Grids [2].This work focuses on providing a skeleton to centralise load management in the Grid environment. However, our aim is to solve load scheduling on the Grid by developing a dynamic decentralised load schedule.

6 Conclusion

The `Grid-GUM2` extension to the `Grid-GUM` RTE has been produced to efficiently and automatically manages data and work on a multi-cluster Grid environment. Achieving good performance was made possible by using a high-level parallel language and completely managing distribution at the RTE level . As future work we plan to measure the behaviour of `Grid-GUM2` using varying number of clusters and interconnect latencies. We are also considering building an analytic model of `Grid-GUM2`

References

1. A. Al Zain, P. Trinder, H-W. Loidl, and G. Michaelson. Grid-GUM: Towards Grid-Enabled Haskell. In *IFL'04 — Intl. Workshop on the Implementation of Functional Languages*, Draft Proceedings, Lübeck, Germany, September 2004.
2. M. Aldinucci, M. Dnelutto, and Dünweber. Optimization Techniques for Implementing Parallel Sckeletons in Grid Environments. In *CMPP'04 — Intl. Workshop on Constructive Methods for Parallel Programming*, Stirling, Scotland, July 2004.
3. M. Alt, H Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. In *CMPP'02 — Int. Workshop on Constructive Methods for Parallel Programming*. Dagstuhl, Berlin, June 2002.
4. B-Y. Evan Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning. Trustless Grid Computing in ConCert. In *In Proceedings of the GRID 2002 Workshop*, volume 2536 of LNCS. Springer-Verlag, 2001.
5. N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *J. of Parallel and Distributed Computing*, 2003.
6. H-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pe na, Á. J. Rebón Portillo, S. Priebe, and P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3), 2003.
7. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1):23–60, January 1998.
8. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Conf. on Programming Language Design and Implementation*, Philadephia, USA, 1996.