Scalable Computing: Practice and Experience
Volume 7, Number 3, pp. 9–25. http://www.scpe.org

ISSN 1895-1767
© 2006 SWPS

MANAGING HETEROGENEITY IN A GRID PARALLEL HASKELL

A. D. AL ZAIN , P. W. TRINDER , G. J. MICHAELSON*, AND H-W. LOIDL†

Abstract. Computational Grids potentially offer cheap large-scale high-performance systems, but are a very challenging architecture, being heterogeneous, shared and hierarchical. Rather than requiring a programmer to explicitly manage this complex environment, we recommend using a high-level parallel functional language, like GpH, with largely automatic management of parallel coordination.

We present *GRID-GUM*, an initial port of the distributed virtual shared-memory implementation of GpH for computational Grids. We show that, *GRID-GUM* delivers acceptable speedups on relatively low latency homogeneous and heterogeneous computational Grids. Moreover, we find that for heterogeneous computational Grids, load management limits performance.

We present the initial design of *GRID-GUM*2, that incorporates new load management mechanisms that cheaply and effectively combine static and dynamic information to adapt to heterogeneous Grids. The mechanisms are evaluated by measuring four non-trivial programs with different parallel properties. The measurements show that the new mechanisms improve load distribution over the original implementation, reducing runtime by factors ranging from 17% to 57%, and the greatest improvement is obtained for the most dynamic program.

Key words. Parallel Computing, Programming Languages

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 Glasgow parallel Haskell (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 [1].

This paper presents *GRID-GUM*, a port of Gum to computational Grids using the de-facto standard Globus Toolkit, in Section 4. Measurements in Section 6 show that *GRID-GUM* gives good performance in some instances, e. g. on homogeneous low-latency multi-clusters. However for heterogeneous architectures load management emerges as the performance-limiting issue.

We present the initial design of *GRID-GUM*2 in Section 7, which incorporates new load distribution mechanisms for virtual shared-memory over a wide area network. The new mechanisms are decentralised, obtaining complete static information during start up, and then cheaply propagating partial dynamic information during execution. The effectiveness of the new mechanisms for multi-clusters Grid environment 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 8. Related work is discussed in Section 9, and we conclude in Section 10.

2. Grids & the Globus Toolkit.

2.1. Overview.

Grid technology is an infrastructure which provides the ability to dynamically link distributed resources as an ensemble to support these execution of large scale, resource-intensive applications [19].

The idea behind the Grid is to serve as an enabling technology for a broad set of applications in science, business, entertainment, health and other areas. Using Berman's classification [19], we are working with computational Grids, which use the Grid to aggregate substantial computational resources in order to tackle problems that cannot be solved on a single system

2.2. Globus Toolkit.

The Globus Toolkit is open source software with an open architecture, comprising a collection of software components designed to support the development of applications for high performance distributed computing environments or "Grids" [20]. The three main components are:

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

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

- Resource Management: allocation and management of Grid resources;
- Information Services: providing information about Grid resources;
- Data Management: accessing and managing data in a Grid environment.

Globus Toolkit is similar to a distributed operating system with uniform access to system features. Globus Toolkit uses a standard application programming interface (API) for sending data and work to other machines which can be expressed in terms of extensible *resource specification language (RSL)*, which is used as a common notation for describing resource requirements. While, RSL is no more sophisticated than other systems for cluster computing e. g. a Beowulf cluster running standard Linux distribution, there are components which might be very useful for *GRID-GUM* in the long run: for example for monitoring system behaviour.

The Grid architecture in the Globus Toolkit [26] identifies the fundamental system components, specifies the purpose and function of these components, and indicates how these components interact with one another. Grid layers defines a slim API for resource and connectivity protocols, so that collective services have a simple interface to work with; on fabric layer, many and often specialised resources are covered (e.g. storage, sensors), not just the usual for parallel computing such as memory, CPU etc. The fabric layer provides the resources that are shared by the Grid: CPU time, storage, sensors. The connectivity layer defines the core communication and authentication protocols required for Grid-specific multi-clusters transactions. In the resource layer there are *information protocols* that tells us about the state of the resource and *management protocols* that negotiate access to a resource. The collective layer includes directory services, scheduling, data replication services, workload management, col-laboratory services and monitoring services.

3. Gum and GpH.

3.1. GpH. GpH is a parallel dialect of the functional language Haskell. Its only extension to Haskell is a primitive, `par`, which indicates a possible parallel execution for a program expression. All dynamic control of the parallelism is completely implicit. This programming model encourages the generation of massive amounts of fine-grained parallelism and puts even higher importance on the efficiency of its management in the runtime environment.

3.2. Gum. Gum (*Graph reduction for a Unified Machine model*) is a parallel runtime environment, implements a functional language and is based on parallel graph reduction [7]. In this model a program is represented as a graph structure and parallelism is exploited by reducing independent subgraphs in parallel. The most natural implementation of parallel graph reduction uses a shared heap for memory management. Gum implements a virtual shared heap on a distributed memory model, using Pvm as generic communication library for transferring data. For efficient compilation we use a state-of-the-art, optimising compiler, namely the Glasgow Haskell Compiler [2]. Originally Gum was defined for homogeneous clusters and currently does not consider information on latencies or the load on other nodes. Gum uses blind load distributed mechanism, where requests are sent to random processing elements (PEs).

Gum has a simple run-time model. Essentially, in the course of execution, PEs generate *sparks* representing potential parallel activities which may subsequently be realised as *threads*. Idle PEs which lack local sparks may request work from other randomly chosen PEs by sending them *fish* messages. If a fished PE does not have spare sparks then it will pass the message onto another PE. Thus, Gum utilises a *pull* approach for work stealing to dynamically balance activity across PEs.

In a homogeneous HPC, the Gum model assumes that all PEs have the same processing and communication characteristics. It is also assumed that a parallel program has sole use of the HPC so its performance is not affected by unpredictable concurrent usage. Thus load balance in Gum may be maintained without reference to run-time loads, with communication overhead from excess fishing restricted through a very simple throttling mechanism.

3.3. Communication Libraries. Gum is independent of the library used to communicate between PEs. Gum was originally based on the Pvm communication library but now has been adapted to use Mpi, in particular Mpich and Mpich-G2. We summarise these libraries before considering their integration into Gum. Pvm(*Parallel Virtual Machine*) emerged as one of the most popular cluster message-passing systems in 1992 [21].

The Mpi (*Message Passing Interface*) Standard defines a library of routines that implement the message passing model [22], and it has a richer set of constructs than Pvm. Mpich is a popular implementation of the Mpi standard [23]. Mpich-G2 is a Grid-enabled implementation of the Mpi standard [25]. It is a port of

Mpich, built on top of services provided from the Globus Toolkit to support efficient, transparent execution in the Grid heterogeneous environments.

4. *GRID-GUM*. To port GpH to computational Grids its Gum runtime environment must be ported to the Grid collective layer as depicted in Figure 4.1. Gum sits above the collective layer provided by Globus Toolkit, which in turn provides a unified distributed environment on the clusters comprising the underlying Grid. Such integration depends on the provision of appropriate communication libraries within the collective layer to link Gum transparently to the Grid: this is considered in more detail in the next section.

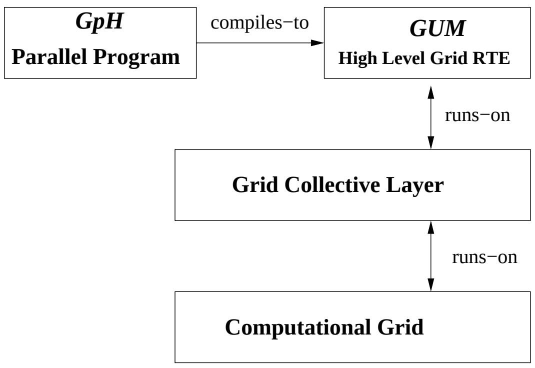


Fig. 4.1. *System Architecture*

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 [3]. The communication management in *GRID-GUM* is similar to Gum, but it uses a different communication library: built around Mpich-G2, and hence the Globus Toolkit [8] as middle-ware. While Gum uses a system manager process to start and stop parallel execution, *GRID-GUM* generates an RSL file internally at the beginning of the execution. This file contains: the PE name, port number, and certificate name, environment variables, arguments for the executable program, the directory where the executable program is located in the specified PE, and the executable program's name. This RSL file is used by MpichG2 to spawn the specified number of PEs.

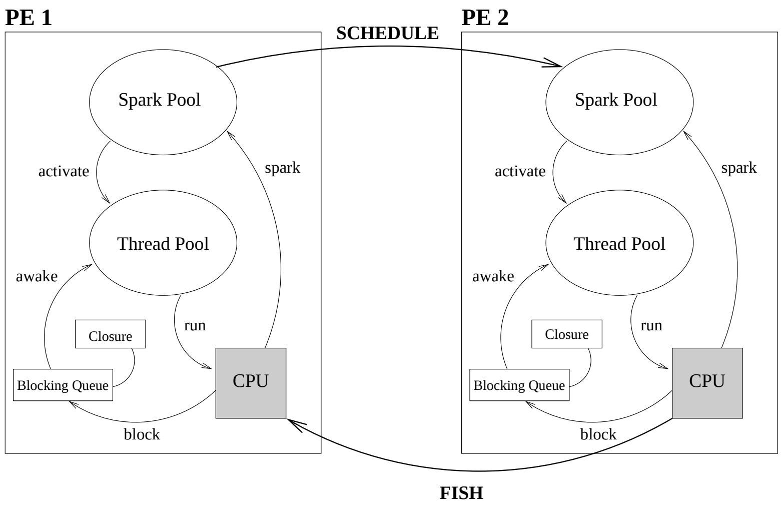


Fig. 4.2. *Interaction of the components of a* Gum *processing element.*

Figure 4.2 illustrates the load distribution mechanism in *GRID-GUM*, depicting the logical components on each 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 4.2 show load distribution 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. Figure 4 summarises the *GRID-GUM* load distribution mechanism, it deals locating work (Figure 4.4), and handling work requests (Figure 4.3), where these activities are performed in the main scheduler loop between thread time slices.

```
IF received FISH THEN
  IF sparks availabel THEN
    send spark in SCHEDULE
      to originPE
  ELSE
    IF FISH exceed age
      THEN
      return back to orginPE
    ELSE
      destPE = random PE
        from PEs list
      send FISH to destPE
```

Fig. 4.3. *Work request*

```
IF idle THEN
  IF runnable thread THEN
    evaluate new thread
  ELSE
    IF spark in spark pool THEN
      active new spark
    ELSE
      IF last SCHEDULE from
        mainPE THEN
        destPE = mainPE
      ELSE
        destPE = random PE from
          PEs list
      send FISH to destPE
```

Fig. 4.4. *Work location*

The load distribution mechanisms in GRID-GUM

5. Measurement Framework.

5.1. Hardware Apparatus. The measurements have been performed on five Beowulf clusters: three located at Heriot-Watt Riccarton campus (*Edin1, Edin2,* and *Edin3*), a cluster located at Ludwig-Maximilians University Munich (*Muni*), and a cluster located at Heriot-Watt boarder campus(*SBC*); see Tables 5.1 and 5.2 for the characteristic of these Beowulfs.

All run-times in the coming tables represent the median of three executions to ameliorate the impact of operating system and shared network interaction. In addition, tables include at the bottom, the minimum, maximum and the geometric mean (root mean square) values.

Table 5.1
Characteristics of Beowulf Clusters

	CPU MHz	Cache kB	Memory kB
Edin1	534	128	254856
Edin2	1395	256	191164
Edin3	1816	512	247816
SBC	933	256	110292
Muni	1529	256	515500

Table 5.2
Approximate Latency between Clusters (ms)

	Edin1	Edin2	Edin3	SBC	Muni
Edin1	0.20	0.27	0.35	2.03	35.8
Edin2	0.27	0.15	0.20	2.03	35.8
Edin3	0.35	0.20	0.20	2.03	35.8
SBC	2.03	2.03	2.03	0.15	32.8
Muni	35.8	35.8	35.8	32.8	0.13

5.2. Software Apparatus. The programs measured in this experiment are classified by the communication degree, which is the number of messages the program sends per second, so we can study the impact of the latency of the network on program behaviour. Six programs are measured in this experiment. Three have low communication degree, `parFib`, `queens` and `sumEuler`, and the other three have relatively high communication degree, `raytracer`, `matMult`, and `linSolv`.

The `parFib` computes Fibonacci numbers. The `sumEuler` program computes the sum over the application of the Euler totient function over an integer list. The `queens` program places a chess pieces on a board. The `raytracer` calculates a 2D image of a given scene of 3D objects by tracing all rays in a given scene of 3D objects by tracing all rays in a given grid, or window. The `matMult` multiples two matrices. The `linSolv` program finds an exact solution of a linear system of equations. See Table 5.3

Table 5.3
Programs Characteristics and Performance

Program	Application Area	Paradigm	Regularity
`queens`	AI	Div-Conq.	Regular
`parFib`	Numeric	Div-Conq.	Regular
`linSolv`	Symb. algebra	Data Para.	Limit irreg.
`sumEuler`	Nume. Analysis	Data Para.	Irregular
`matMult`	Numeric	Divi-Conq.	Irregular
`raytracer`	Graphic	Data Para.	High irreg.

6. *GRID-GUM* Performance. In developing *GRID-GUM*, a crucial first step was to ensure that Gum could seamlessly support GpH in a Grid environment. In particular, it was important to demonstrate conclusively that the HPC-oriented Gum communication layer could be modified for transparent use in a heterogeneous Grid. As discussed above, Gum communication is based on Pvm, where communication in widely used Grid environments like Globus Toolkit is based on special forms of Mpi. While there is some evidence that Pvm and Mpi offer comparable behaviours, it was not known whether the additional Grid control layers might add unacceptable overheads costs to Gum, rendering its use inappropriate for parallel functional programming support in a Grid.

Table 6.1
Dynamic Program Properties on 16 PEs

program Name	comm library	No of Threads	Alloc Rate MB/s	comm Degree Pkts/s	Average Pkt Size Byte
`parFib`	Pvm	26595	55.3	65.5	5.5
	Mpich	26595	52.7	58.0	5.5
	Mpich-G2	26595	43.2	14.8	5.6
`sumEuler`	Pvm	82	52.8	2.09	90.2
	Mpich	82	47.9	1.4	90.3
	Mpich-G2	82	45.7	0.7	90.2
`raytracer`	Pvm	350	60.0	46.7	321.7
	Mpich	350	61.4	45.5	320.4
	Mpich-G2	350	49.5	62.9	323.0
`linSolv`	Pvm	242	40.3	5.5	290.6
	Mpich	242	40.8	3.1	300.1
	Mpich-G2	242	26.5	2.5	276.3
`matMult`	Pvm	144	39.0	67.3	208.8
	Mpich	144	40.1	52.2	213.3
	Mpich-G2	144	40.0	31.2	209.3
`queens`	Pvm	24	38.8	0.2	851.8
	Mpich	24	37.0	0.2	818.9
	Mpich-G2	24	34.0	0.1	846.1

Table 6.2
Speedup on 16 PEs

program	comm	Runtime		Speedup		%variance	
Name	library	Seq sec	16 PE sec	Wall Clock	Exec	Wall Clock	Exec
`parFib`	Pvm	413.7	22.8	14.8	17.1	00%	00%
	Mpich	409.4	20.5	6.8	19.8	54%	-15%
	Mpich-G2	465.1	26.3	2.3	17.6	84%	-2%
`sumEuler`	Pvm	1607.1	131.8	11.1	12.1	00%	00%
	Mpich	1585.1	139.2	8.8	11.3	20%	6%
	Mpich-G2	1598.1	188.1	3.5	8.4	68%	30%
`raytracer`	Pvm	2855.4	315.3	8.9	9.6	00%	00%
	Mpich	2782.7	365.2	7.8	8.9	12%	7%
	Mpich-G2	2782.7	301.7	6.8	9.2	22%	4%
`linSolv`	Pvm	834.2	102.6	6.5	8.9	00%	00%
	Mpich	828.4	110.5	5.5	7.3	15%	17%
	Mpich-G2	828.9	112.2	5.1	7.3	21%	17%
`matMult`	Pvm	891.9	150.2	5.9	5.9	00%	00%
	Mpich	891.9	191.9	4.6	4.6	21%	21%
	Mpich-G2	916.3	292.6	3.1	5.0	47%	15%
`queens`	Pvm	2802.7	375.1	7.4	7.4	00%	00%
	Mpich	2802.7	390.9	7.1	7.1	4%	4%
	Mpich-G2	2816.4	567.8	4.9	6.2	33%	16%
Min	Pvm						
	Mpich					4%	-15%
	Mpich-G2					21%	-2%
Max	Pvm						
	Mpich					54%	21%
	Mpich-G2					84%	30%
Geometric Mean	Pvm						
	Mpich					26%	13%
	Mpich-G2					49%	16%

6.1. Communication Library Impact. This experiment investigates the impact of using different communication libraries on the performance on a single cluster.

The measurements in this section have been performed on the Edin1 cluster. In Table 6.2, the fifth and sixth columns record the wall-clock and execution speedup. The wall-clock time is the execution time plus the startup time. The seventh and the last columns show the percentage variance of the wall-clock and execution speedup relative to the Gum/Pvm implementation speedup.

Overall, the Gum/Pvm implementation consistently shows the best wall-clock speedup and Gum/Mpich-G2 the worst marked as the average packet size, in Gum level, shrinks. As shown in measurements in the Table 6.1, the average packet size is relatively small for `parFib`, and `sumEuler`, and the wall-clock speedup variance is big between the different Gum implementations for these programs. For `raytracer`, `matMult`, and `linSolv` the average packet size is significantly larger and the wall-clock speedup variance is smaller.

The main source of overhead for the communication is the time needed for packing and unpacking in the communication libraries. Good performance for small packets is important for Gum, since parallel functional programs have massive amount of fine grained parallelism including many small messages. This is untypical for general parallel applications, and Mpi implementations may well be tuned for the common case of large packet sizes. However, the big difference between Mpich and Mpich-G2 is related to the extra startup security checking overhead which Globus Toolkit adds for Mpich-G2

Comparison of the execution-time speedup of the Gum implementations with the different GpH programs shows that no implementation is always better than the others. However, the differences in execution-time speedup are less marked than the differences on the wall-clock speedup.

To summarise:

- For programs with long execution time the performance of Gum is independent of the communication libraries (Table 6.2);

- For small programs Gum with Pvm gives the best wall clock speedup and Gum with Mpich-G2 the worst (Table 6.2);
- Mpich-G2 has a high startup cost relative to Pvm or Mpich (Table 6.2).

6.2. *GRID-GUM* on Multiple Clusters.

6.2.1. Low Latency Multi-Cluster. This experiment investigates the performance impact of executing GpH programs on multiple heterogeneous clusters with moderate latency interconnect.

Table 6.3
Heterogeneous Clusters and Low Latency Interconnect Results

	`raytracer`			`queens(13)`		
	Speedup		Rtime	Speedup		Rtime
	F	S	Sec.	F	S	Sec.
F	1.0	3.3	1483.3	1.0	3.2	719.5
S	0.3	1.0	4894.0	0.3	1.0	2324.7
FF	1.9	6.3	772.8	1.8	6.0	384.6
FS	1.2	4.0	1199.4	0.9	3.0	753.5
SS	0.5	1.8	2698.5	0.6	1.9	1176.9
SF	0.7	2.3	2106.1	1.0	3.4	666.3
FFF	2.7	8.9	545.1	2.8	9.3	249.5
FFS	2.0	6.7	728.6	0.9	3.0	768.2
FSS	1.4	4.8	1002.3	0.9	3.1	733.6
SSS	0.8	2.9	1663.0	0.9	2.9	795.7
SSF	1.4	4.6	1047.8	1.1	3.7	627.6
SFF	1.4	4.8	1002.1	1.5	4.8	478.3

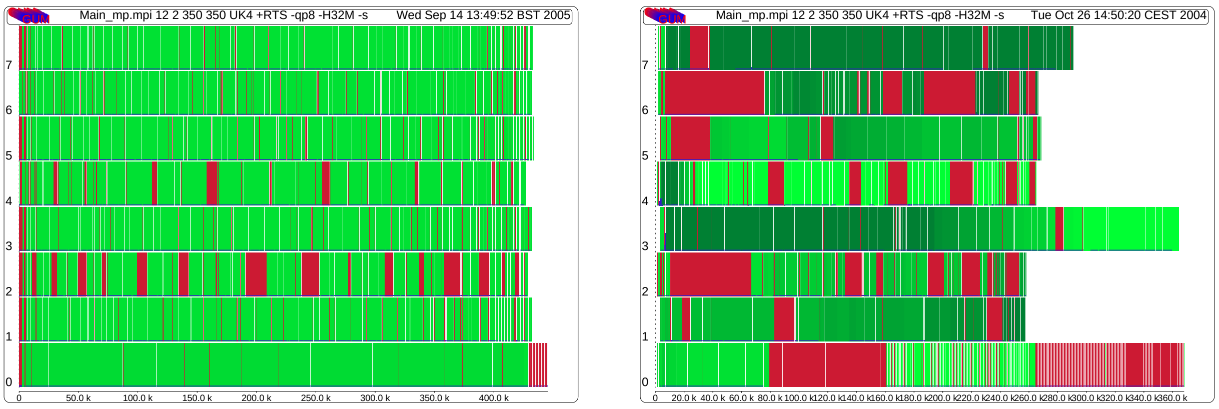
	`raytracer`			`queens(13)`		
	Speedup		Rtime	Speedup		Rtime
	F	S	Sec.	F	S	Sec.
FFFF	3.4	11.4	425.9	2.7	9.0	258.2
FFFS	2.7	9.0	538.8	1.4	4.8	483.0
FFSS	2.2	7.2	675.7	1.2	4.0	578.0
FSSS	1.7	5.8	833.1	1.2	4.1	561.6
SSSS	1.1	3.8	1280.9	0.9	3.1	741.6
SSSF	1.6	5.3	916.2	1.2	4.1	560.3
SSFF	1.4	4.8	1006.7	1.2	4.1	563.5
SFFF	1.4	4.6	1046.3	1.9	6.1	375.5
FFFFF	4.0	12.9	376.7	4.0	12.8	181.1
FFFFS	3.5	11.5	422.9	2.8	9.1	254.5
FFFSS	2.9	9.4	519.2	1.3	4.2	544.9
FFSSS	2.4	7.9	615.3	1.3	4.3	530.1
FSSSS	1.9	6.4	755.6	1.2	4.0	577.7
SSSSF	1.7	5.7	850.7	1.2	4.1	560.5
SSSFF	1.8	6.2	786.0	1.5	4.9	474.3
SSFFF	1.8	6.1	790.4	1.9	6.1	375.4
SFFFF	1.9	6.5	747.6	2.2	7.3	316.5

The measurements in Table 6.3 use Mpich communication library on SBC and Edin3 Beowulf clusters described in Table 5.1. Each SBC machine is labelled S (Slow) and each Edin3 machine is labelled F (Fast). Two programs are measured: `raytracer` with relatively high communication degree, and `queens` with relatively low communication degree. The first column shows different combinations of machines. The second and the fifth columns record the speedup using F's sequential runtime for `raytracer` and `queens` respectively. The third and the sixth columns records the speedup using S's sequential runtime, and the fourth and the last columns show the wall-clock time. The first machine in the configuration string is where the program starts.

Table 6.3 shows that, replacing a local machine S by a faster remote machine F decreases the runtime and increases the speedup. For example in Table 6.3, SSS cluster requires 1663.0s to finish the computation of `raytracer`; however, if S machine has been replaced by F remote machine, the runtime is decreased by 37%. Interestingly, this result supports the idea of using a fast remote machine to improve the performance of a GpH parallel program, and it shows that *GRID-GUM* can cope with moderate latency network without modification.

However, it is observable that *GRID-GUM*, with its blind load mechanism, often gives unsatisfactory scheduling in heterogeneous Grid multi-clusters. For example, replacing one of the FFF machines by a slower remote machine S increases the runtime of `queens` from 249.5s to 768.2s, i. e. by a factor of three. Likewise, adding a slower remote machine S to two FF local machines increases the runtime of `queens` from 384.6s to 768.2s i. e. by a factor of two.

GRID-GUM shows relatively poor performance on heterogeneous cluster for many programs, and that is due to poor load management. For example, Figure 6.1 shows *GRID-GUM* per-PE activity profile for `raytracer` on a heterogeneous and a homogeneous cluster. 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



a) Homogeneous clusters

b) Heterogeneous cluster

FIG. 6.1. *per PE Activity Profile for* `raytracer`

(green in a colour profile) indicating a larger number of runnable threads. Gaps in the horizontal lines (red areas in the colour profile) indicate idleness.

Figure 6.1.a depicts the performance on homogeneous cluster where all PEs have the same CPU speed. Figure 6.1.b depicts the performance on heterogeneous cluster where there are four fast machines (0-3) and four slow machines (4-7). All PEs in Figure 6.1.a are uniformly loaded, and finish at the same time, in contrast the PEs in Figure 6.1.b have numerous idle periods, and finish at different times. Figure 6.1.b also shows long idle periods at the beginning of the computation, where only a small amount of parallelism is available, and blocking on data that is remotely evaluated will cause the entire PE to remain idle until new work is obtained (see the start of PE 6). Matching the profile in Figure 6.1.a, the fast processors in Figure 6.1.b (0-3) show a fairly balanced load and finish at about the same time. Towards the end only PE 3 has useful work, and the main PE 0 has to wait for it to finish. Considering the runtime of the heterogeneous cluster, 368.0s, is almost two times greater than the runtime of the homogeneous cluster, 220.0s.

To summarise:

- Replacing a local PE with a faster remote PE reduces execution time (Table 6.3);
- *GRID-GUM*'s load balancing mechanism does not deliver good scheduling in a heterogeneous GRID multi-clusters (Figure 6.1);
- In a moderate latency configuration, latency is not the dominating factor, since *GRID-GUM* can overlap communication with computation, provided a sufficient amount of parallelism is available (Table 6.3).

6.2.2. High Latency Multi-Cluster. This experiment investigates the performance impact of executing GPH programs on multiple homogeneous clusters with a high latency interconnect. We measure programs with both low and high communication degrees.

The measurements in Table 6.4, and 6.5 use MPICH-G2 communication library on the Muni and Edin2 Beowulf clusters described Table 5.1. Each Muni machine is labelled M and each Edin2 machine is labelled E

Five programs have been tested: two programs with relatively low communication degree `parFib`, and `sumEuler`, and three programs with relatively high communication degree `raytracer`, `linSolv`, and `matMult`, see Table 6.1.

For programs with a low communication degree, Table 6.4 shows that adding a remote machine M decreases the runtime. Even on multi-clusters configurations with very high latency between the clusters, the additional computational power outweighs the expensive but infrequent communication. It also shows that replacing a local machine E by a remote machine M does not grossly deteriorate performance. For example, in Table 6.4, in an EEE configuration `sumEuler` requires 899.7s to finish, machine M is added $EEEM$ the runtime decreases by 26.0%. Furthermore, replacing a local machine E by a remote machine M, yielding a EEM configuration, shows little change in the runtime (3.5%). In short, using remote machines in high latency communications does not have impact on the performance of low communication degree programs.

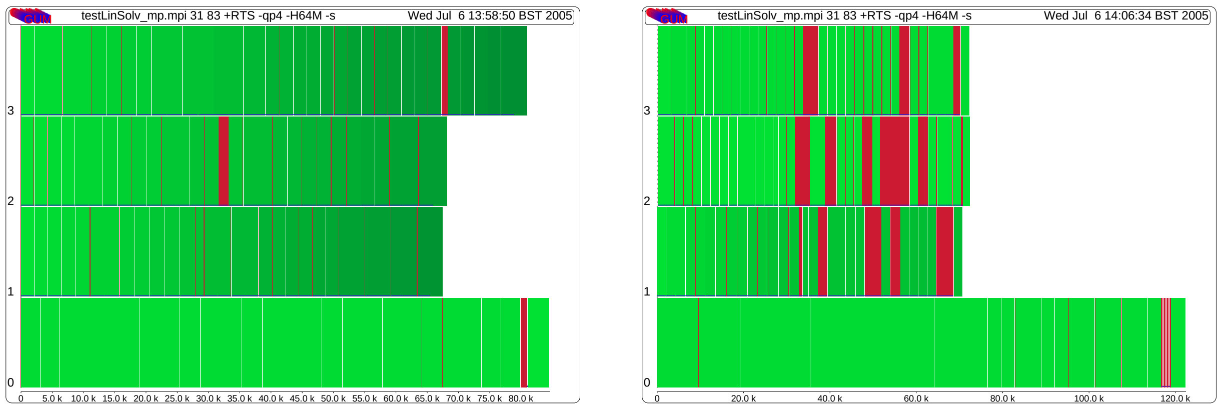
Table 6.5 shows, programs with a high communication degree, replacing a local machine with a slightly faster remote machine increases the runtime and decreases the speedup. For instance `linSolv` on two lo-

Table 6.4
Low Communication Degree Programs

	`parFib`(45)			`sumEuler`		
	Rtime	Spedup		Rtime	Spedup	
	Sec.	E	M	Sec.	E	M
M	867.5	1.2	1.0	3138.5	1.0	1.0
E	1070.1	1.0	0.8	3227.6	1.0	0.9
MM	431.0	2.2	2.0	1270.4	2.5	2.4
EM	480.6	2.2	1.8	1308.8	2.4	2.3
EE	536.8	1.9	1.6	1332.8	2.4	2.3
MMM	298.8	3.5	2.9	869.9	3.7	3.6
EMM	331.1	3.2	2.6	838.7	3.8	3.7
EEM	338.9	3.1	2.5	867.9	3.7	3.6
EEE	374.8	2.8	2.3	899.7	3.5	3.4

	`parFib`(45)			`sumEuler`		
	Rtime	Spedup		Rtime	Spedup	
	Sec.	E	M	Sec.	E	M
MMMM	241.9	4.4	3.5	629.7	5.1	4.9
EMMM	251.3	4.2	3.4	670.7	4.8	4.6
EEMM	268.0	3.9	3.2	665.8	4.8	4.7
EEEM	274.9	3.8	3.1	665.5	4.8	4.7
EEEE	292.9	3.6	2.9	662.2	4.8	4.7
MMMMM	205.9	5.0	4.2	523.2	6.1	5.9
EMMMM	212.7	5.0	4.0	544.0	5.9	5.7
EEMMM	226.2	4.7	3.8	553.7	5.8	5.6
EEEMM	224.7	4.7	3.8	620.8	5.1	5.0
EEEEM	234.0	4.5	3.7	588.4	5.4	5.3
EEEEE	251.3	4.2	3.4	570.8	5.6	5.4

cal machines EE takes 174.9s, if one of the local machine is replaced by a remote machine EM, the runtime increases by 41.4%. Note that for all programs the runtime increases when adding a remote machine in such a way. Furthermore, a configuration of the form $EMM\ldots\ M$ is always worst among the configurations with the same number of PEs. This is because the local machine E, which has all the work in the beginning of the execution, has to communicate with the other machines through a high latency network, which becomes a bottleneck in the execution. Finally, configurations of the form $E\ldots E$ or $M\ldots M$ are usually the best configurations, because all machines communicate with others through the low latency network.



a) Homogeneous low latency interconnect

b) Heterogeneous high latency interconnect

Fig. 6.2. *per PE Activity Profile for* `linSolv` *on multi-Clusters*

GRID-GUM shows relatively poor performance on high latency interconnect multi-clusters for many programs. For example, Figure 6.2 shows *GRID-GUM* per-PE activity profile for `linSolv` on a homogeneous low and a heterogeneous high latency interconnect multi-clusters. Figure 6.2.a depicts the performance on homogeneous low latency interconnect cluster. Figure 6.2.b depicts the performance on heterogeneous high latency interconnect where PE 0 & PE 1 and PE 2 & PE 3 are connected pairwise by a low latency network, and with a high latency network between the pairs.

In Figure 6.2.b the PEs exhibit significantly more idle time, i. e. gaps in the horizontal line, and complete at different times. In contrast, the work is fairly evenly balanced in Figure 6.2.a. The idle time in Figure 6.2.b is due to PEs waiting for data to without other threads execute.

Table 6.5
High Communication Degree Programs

	`raytracer`			`matMult`			`linSolv`		
	Rtime	Speedup		Rtime	Speedup		Rtime	Speedup	
	Sec.	E	M	Sec.	E	M	Sec.	E	M
M	903.8	1.1	1.0	265.8	0.9	1.0	290.0	1.0	1.0
E	1027.8	1.0	0.8	259.9	1.0	1.0	299.3	1.0	0.9
MM	548.6	1.8	1.4	228.8	1.1	1.1	196.5	1.5	1.4
EM	624.6	1.6	1.4	393.0	0.6	0.6	232.0	1.2	1.2
EE	545.7	1.8	1.6	227.8	1.1	1.1	164.9	1.8	1.7
MMM	383.7	2.6	2.3	133.0	1.9	1.9	139.4	2.1	2.0
EMM	535.8	1.9	1.6	297.7	0.8	0.8	231.8	1.2	1.2
EEM	494.9	2.0	1.8	201.9	1.2	1.3	141.1	2.1	2.0
EEE	387.5	2.6	2.3	137.8	1.8	1.9	136.8	2.1	2.1
MMMM	312.8	3.2	2.8	121.9	2.1	2.1	119.5	2.5	2.4
EMMM	497.6	2.0	1.8	295.0	0.8	0.9	142.5	2.1	2.0
EEMM	421.7	2.4	2.1	213.9	1.2	1.2	134.9	2.2	2.1
EEEM	377.9	2.7	2.3	145.9	1.7	1.8	120.9	2.4	2.3
EEEE	326.4	3.1	2.7	114.8	2.2	2.3	117.1	2.5	2.4
MMMMM	287.8	3.5	3.1	108.6	2.3	2.4	104.4	2.8	2.7
EMMMM	473.8	2.1	1.9	290.8	0.8	0.9	147.0	2.0	1.9
EEMMM	413.7	2.4	2.1	228.8	1.1	1.1	142.1	2.1	2.0
EEEMM	378.7	2.7	2.3	150.9	1.7	1.7	104.9	2.8	2.7
EEEEM	329.9	3.1	2.7	125.1	2.0	2.1	107.7	2.7	2.7
EEEEE	279.8	3.6	3.2	95.9	2.7	2.7	102.9	2.9	2.8

To summarise:

- For high communication degree programs *GRID-GUM* delivers poor performance on high latency multi-clusters (Table 6.5);
- For low communication degree programs *GRID-GUM* can delver good performance on high latency multi-clusters (Table 6.4);
- The poor performance of *GRID-GUM* on high latency multi-clusters is primarily due to poor load management (Figure 6.2).

7. ***GRID-GUM*2.** Based on the results in previous section, it is essential to modify *GRID-GUM* for execution on a computational Grid, (*GRID-GUM*2). *GRID-GUM*2 uses the monitored information to provide a good load distribution over the Grid using the following policies:

- An idle PE sends a FISH message only to a PE that has high load relative to its CPU speed.
- PEs have a preference for obtaining work from PEs that currently have low communication latency.
- The recipient PE switches from passive to active load distribution if a FISH message received from another cluster.

The new *GRID-GUM*2 mechanism has two main components: information collection and adaptive load distribution. The information collection is supported by a monitoring mechanism to provide the current state information of the Grid network. The monitoring mechanism performs during the whole course of execution. It collects static information like CPU speed at the start of program in *PEStatic* table (Figure 7.1), and dynamic information such as load and latency during the execution in *PEDynamic* and *ComMap* tables (Figures 7.2 and 7.3) respectively. The adaptive load distribution of *GRID-GUM*2 comprises the following aspects:

- Resource-level load distribution: programs executed do not required specific resource, present on only same PEs. Idle PEs use load distribution mechanism in*GRID-GUM*2 to seek work from PEs relatively

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. 7.1. *PEStatic Table*

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

FIG. 7.2. *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. 7.3. *ComMap Table*

```
IF received fish THEN
 update tables with data
   from fishing PE
 IF sparks availabelTHEN
   IF fishing PE is local THEN
     send sparks in sechedule
       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. 7.4. *Work Request*

```
IF idle 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. 7.5. *Work Location*

The Load Distribution Mechanisms in GRID-GUM2

heavily loaded.

- Dependent load distribution: *GRID-GUM*2 aims for an efficient load distribution mechanism to a single parallel program with dependent tasks.
- Decentralised information services: *GRID-GUM*2 maintains a decentralised scheme where every PE is responsible for maintaining state information of some nearby PEs and share it with other PEs.
- Dynamic load distribution: *GRID-GUM*2 assumes that limited knowledge about the load and PEs are available *a priori*, and load distribution decisions have to be made during the execution.
- Decentralised load distribution organisation: *GRID-GUM*2 distributes the load distribution decision to every PE. Therefore, each PE acts as both a load distributer and a computational resource.
- Redistribution support: *GRID-GUM*2 supports work placement which enhance system reliability and flexibility.
- Adaptive load distribution: *GRID-GUM*2 is a mainly passive load distribution system where lightly loaded PEs have to explicitly ask for work from PEs with excess load. However, if an idle PE requests work from a PE residing outside its cluster and the request originated from relatively powerful cluster, it changes from a passive to an active system and the recipient PE sends more work to the idle PE.

The core of *GRID-GUM*2 load distribution can be summarised as work location (Figure 7.4), and work request handling (Figure 7.5).

8. *GRID-GUM*2 Performance on Heterogeneous Architecture. This experiment investigates the performance impact of using the adaptive load distribution of *GRID-GUM*2 on multiple heterogeneous clusters with moderate latency interconnect.

The measurements in Table 8.1 use *GRID-GUM* and *GRID-GUM*2 on Edin1 and Edin2 Beowulf clusters described in Table 5.1. Four GPH programs are measured in this experiment: `queens`, `sumEuler`, `linSolv` and `raytracer` described in Section 5. In Table 8.1, The second and third columns record the run-time using *GRID-GUM* and *GRID-GUM*2 in seconds respectively. The last column shows the percentage improvement of

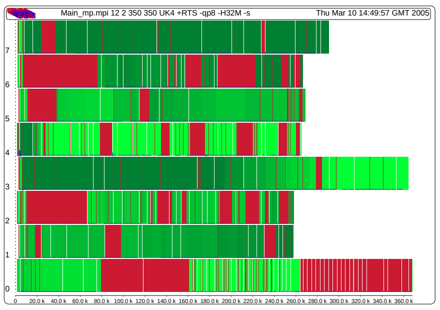
*GRID-GUM*2.

Table 8.1
Performance On Heterogeneous Architecture

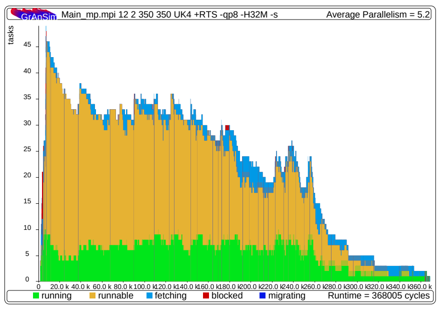
Program	Run-time (s)		Improvement %
	GRID-GUM	*GRID-GUM*2	
`queens`	668	310	53%
`sumEuler`	570	279	51%
`linSolv`	217	180	17%
`raytracer`	1340	572	57%
Min			17%
Max			57%
Geometric Mean			47.3%

Table 8.1 shows that, *GRID-GUM*2 outperforms *GRID-GUM* on multiple heterogeneous clusters with moderate latency interconnect as far as the execution time is concerned. *GRID-GUM*2 shows run-time improvements between 17% and 57%. The greatest improvement are given with the most dynamic program, `raytracer`. Through the rest of this sub-section we consider studying in more details the behaviour of `raytracer` in multi-clusters heterogeneous architecture.

`raytracer` has highly irregular execution, and consequently is very sensitive to changes in parallel environment. Figure 8.1 shows per-PE and overall activity profiles 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). An overall activity profile shows the behaviour of the program at each instant of its execution.



a) Per-PE activity profile

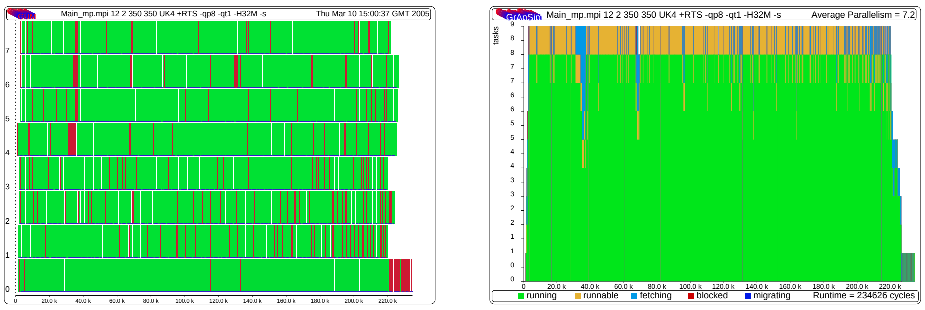


b) Overall activity profile

Fig. 8.1. *GRID-GUM:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

Figure 8.1.a shows a poor load distribution of *GRID-GUM* with `raytracer` to calculate an image with resolution 350×350 using eight heterogeneous machines, i. e. four fast and four slow machines. PEs as depicted in Figure 8.1 have numerous idle period and finish at different time. From Figure 8.1.b, it is observable that there are a considerable number of runnable threads waiting to be evaluated at most of the execution time. This may explain the poor load distribution in *GRID-GUM*. PEs with slow CPU speed in a heterogeneous architecture in *GRID-GUM* show the same demand of seeking work as PEs with fast CPU speed. This concludes that PEs with slow CPU speed accumulate and activate sparks as PEs with fast CPU speed. If a spark has been activated, it remains in its local PE as runnable or blocked thread in the thread pool and it can not be evaluated by another PE. Considering that PEs have different capabilities of evaluating their own threads explains the reason that there are many runnable threads are waiting to be evaluated while there are some PEs are idle.

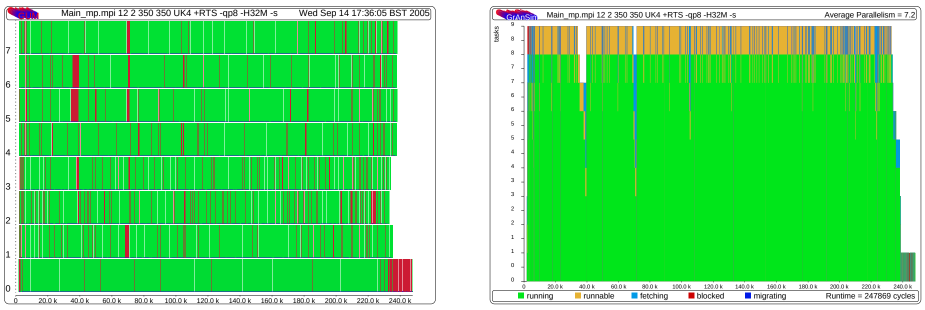
GRID-GUM provides explicit control over the load distribution by specifying a hard limit on the total number of live threads, i. e. runnable or blocked threads. Figure 8.2 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution 350×350, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). *GRID-GUM* in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.



a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.2. *GRID-GUM with Thread Limitation:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

In Figure 8.2, *GRID-GUM* with thread limitation shows an efficient load distribution in a heterogeneous architecture with moderate latency interconnect. it completes the image manipulation in 327 s, while the version of *GRID-GUM* does not employ thread limitation requires 441 s. Expectedly, for the same problem *GRID-GUM*2 has similar performance, i. e. 338 s, with *GRID-GUM* using thread limitation (Figure 8.3).



a) Per-PE activity profile *b*) Overall activity profile

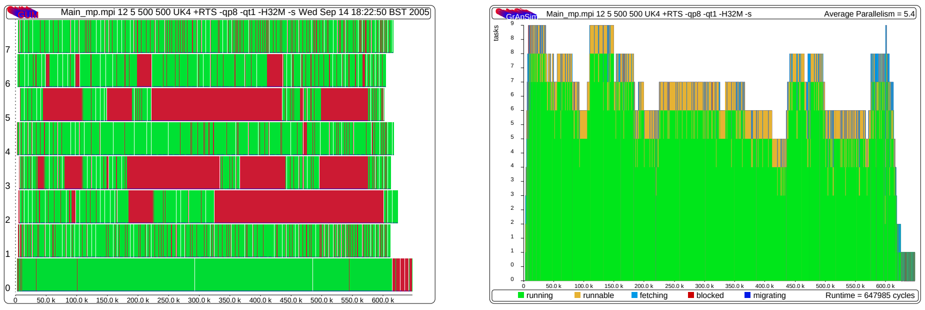
FIG. 8.3. *GRID-GUM2:* `raytracer` *with* 350*X*350 *Image on a Heterogeneous multi-Clusters*

However, *GRID-GUM*'s load distribution efficiency regress when the size of the input increased even with thread limitation. Figure 8.4 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution 500×500, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7). *GRID-GUM* in this experiment uses a hard limit of 1 on the total number of live threads in the thread pool.

PEs in Figure 8.4.a finish at the same time, but they still have numerous idle periods which deteriorate the performance. This idle periods are caused by the dependencies between threads in `raytracer`. These dependencies are effected badly by the thread limitation, which causes PEs to remain idle waiting for certain threads to be evaluated. Figure 8.4.b shows that the idle periods are not caused by lack of tasks to be evaluated. Generally speaking, thread limitation has a serious impinge on many programs performance. Figure 8.5 shows per-PE profiles for `linSolv` with and without thread limitation on 8 homogeneous machines from Edin1 Beowulf cluster.

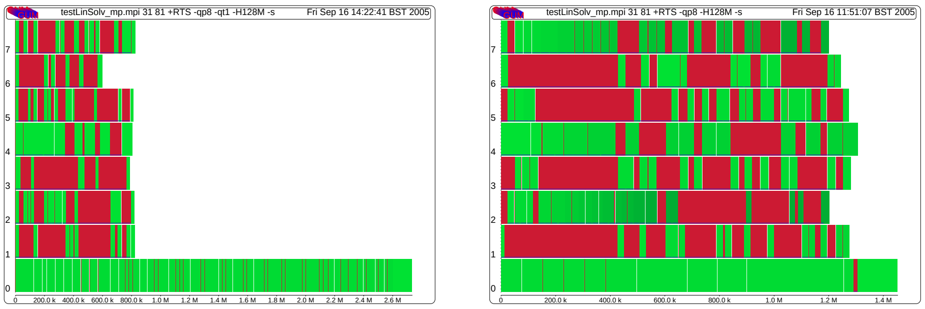
From Figure 8.5, *GRID-GUM* delivers better performance with `linSolv` without using thread limitation. *GRID-GUM* requires 2802 s to finish `linSolv` computation using thread limitation, unlike when thread limitation is excluded *GRID-GUM* requires only 1521 s to finish the same computation in the same platform.

However, *GRID-GUM*2 shows more effective load distribution in heterogeneous architecture in comparison with *GRID-GUM*'s load distribution. Figure 8.6 shows per-PE and overall activity profiles for `raytracer` to calculate an image with resolution 500×500, with execution on four fast machines (0,2,4,6), and four slow machines (1,3,5,7).



a) Per-PE activity profile *b*) Overall activity profile

Fig. 8.4. *GRID-GUM with Thread Limitation:* `raytracer` *with* 500*X*500 *Image on a Heterogeneous multi-Clusters*



a) With thread limitation *b*) With no thread limitation

Fig. 8.5. *GRID-GUM:* `linSolv` *on a Heterogeneous multi-Clusters*

PEs in Figure 8.6.a are fairly balanced and and finish at about the same time. Figure 8.6.b shows that *GRID-GUM*2 scores a good average parallelism in 8 PEs, 6.9, and generates enough tasks for all PEs at each instant of the execution time. Finally, *GRID-GUM*2 outperforms *GRID-GUM* with thread limitation in `raytracer` when the image resolution increases from 350×350 to 500×500, the execution time for *GRID-GUM*2 and *GRID-GUM* with thread limitation is 572 s and 814 s, respectively.

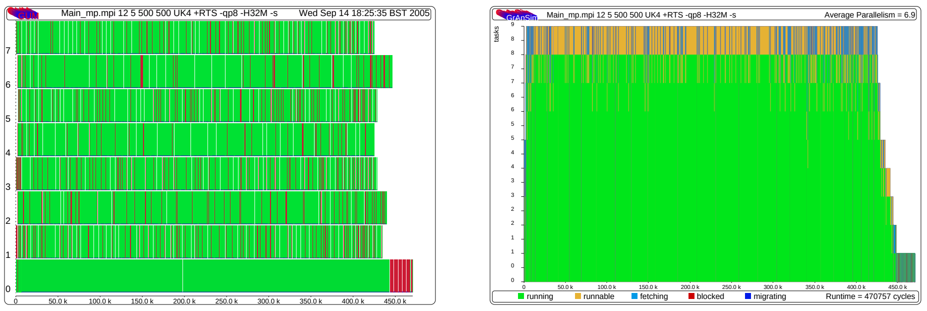
To summarise:

- *GRID-GUM*2 shows efficiency and automatic management of data and work on heterogeneous multi-clusters Grid environment (Table 8.1);
- For some programs, thread limitation improves the performance of *GRID-GUM* on heterogeneous multi-clusters but not for all (Figures 8.2 and 8.5);
- *GRID-GUM*2 outperforms *GRID-GUM* and *GRID-GUM* with thread limitation for large input sizes (Figures 8.6 and 8.4).

9. Related Work. The most closely related to our philosophy of semi-implicit management of parallelism in a high level language is the ConCert system [10] system and the Hemlock compiler [11], 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.

Under the topic of meta-computing several projects, like Harness [12], aim at provide functionality similar to *GRID-GUM*2. The characteristic difference to *GRID-GUM*2 is the automatic management of parallelism within one parallel program.

Alt *et al* apply skeletons to computational Grids [13]. 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 [14].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.



a) Per-PE activity profile *b*) Overall activity profile

FIG. 8.6. *GRID-GUM2:* `raytracer` *with* 500 × 500 *Image on a Heterogeneous multi-Clusters*

10. Conclusion. We have presented and measured two GRID-enabled runtime environments for the GPH high-level parallel programming language.

Measurements of *GRID-GUM* showed that for large programs, the performance of GUM on a single cluster is largely independent of the communication library used. Despite being designed for homogeneous clusters, *GRID-GUM* delivers good and predictable speedups on GRID multi-clusters with a low latency interconnect. In contrast, on GRID multi-clusters with heterogeneous architecture, *GRID-GUM* does not deliver good performance due to poor scheduling. In addition, on GRID multi-clusters with a high latency interconnect, Grid-GUM only delivers acceptable speedups for low communication degree programs.

We have presented the initial design of *GRID-GUM*2 that incorporates new load management mechanisms, informed by the *GRID-GUM* results. *GRID-GUM*2 achieves good parallel performance for a typical set of symbolic applications running on two heterogeneous clusters connected via the Globus Toolkit, realising a small but typical computational GRID. The improved performance is achieved by dynamically distributing work between the machines on top of a virtual shared memory implementation. No explicit thread placement or scheduling has to be done by the programmer. In particular, our system makes contributions towards load distribution on such wide-area networks

We conclude that, with appropriate load management strategies, acceptable performance can be obtained on hereogeneous computational GRIDs from a distributed virtual shared heap implementation of a high-level parallel language.

REFERENCES

[1] H-W. LOIDL, F. RUBIO, N. SCAIFE, K. HAMMOND, S. HORIGUCHI, U. KLUSIK, R. LOOGEN, G. J. MICHAELSON, R. PEÑA, Á. J. REBÓN PORTILLO, S. PRIEBE AND P. W. TRINDER, *Comparing Parallel Functional Languages: Programming and Performance*, in Higher-order and Symbolic Computation, Kluwer Academic Publishers, 16(3),2003.
[2] GHC,*The Glasgow Haskell Compiler*, Department of Computing Science, University of Glasgow (`http://www.dcs.gla.ac.uk/`), January 1998, "The Glasgow Haskell Compiler compiles code written in the functional programming language Haskell" URL: `http://www.dcs.gla.ac.uk/fp/software/ghc/`
[3] 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, 1996, Philadephia USA.
[4] T. L. CASAVANT AND J. G. KUHL, *A Taxonomy of Scheduling in General-Purpose Distribution Computing Systems*, in IEEE Transactions on Software Engineering, 14(2),1988, ISSN 0098-5589, pages 141–154, IEEE Press, Piscataway NJ USA.
[5] Y-T. WANG, AND R. J. T. MORRIS, *Load Sharing in Distributed Systems* , In Scheduling and Load Balancing in Parallel and Distributed Systems, 1995, Shirazi, A. and Hurson, A. R. and Kavi, K. M., eds, IEEE Transactions on Software Engineering, pp. 7–20, ACM.
[6] D. L. EAGER, E. D. LAZOWSKA AND J. ZAHORJAN, *A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract)*, in SIGMETRICS '85: Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems, 1985, ISBN 0-89791-169-5, pp. 1–3, Austin Texas United States, ACM Press.
[7] S. L.PEYTON JONES, C. CLACK, J. SALKILD AND M. HARDIE, *GRIP—a High-Performance Architecture for Parallel Graph Reduction*, in Intl. Conf. on Functional Programming Languages and Computer Architecture, pp. 98–112, September 1987, LNCS 274, Portland Oregon, Springer-Verlag.
[8] I. FOSTER, AND C. KESSELMAN, *Globus: A Metacomputing Infrastructure Toolkit*, in "The International Journal of Supercomputer Applications and High Performance Computing", 11(2), pp. 115–128, 1997.

[9] A. Al Zain, P. Trinder, H-W. Loidl and G. Michaelson, *Grid-GUM: Towards Grid-Enabled Haskell*, in Draft Proceedings of IFL'04—Intl. Workshop on the Implementation of Functional Languages, Septamber 2004, Lübeck Germany.

[10] Trustless Grid Computing in ConCert, *B-Y. Evan Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning*, In Proceedings of the GRID 2002 Workshop, 2536 of LNCS, Springer-Verlag, 2001.

[11] T. Murphy VII, *Hemlock and Concert v2 Framework*, Talk at Carnegie Mellon University, August 2003

[12] M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, M. Kohl, J. Migliardi, K. Moore, T. Moore P. Papadopoulos, S. Scott and V. Sunderam *HARNESS: A Next Generation Distributed Virtual Machine*, in Future Generation Computer Systems, 15(5/6):571-582, October 1991, Special Issue in Metacomputing.

[13] 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, June 2002.

[14] M. Aldinucci, M. Dnelutto and Dünnweber, *Optimization Techniques for Implementing Parallel Sckeletons in Grid Environments*, in CMPP'04—Intl. Workshop on Constructive Methods for Parallel Programming, July 2004, Stirling Scotland

[15] M. Litzkow, M. Livny and M. Mutka, *Condor- A Hunter of Idle Workstations*, in Proc. the 8^th InternationalConference of Distributed Computing Systems, San Jose, California, June 1988.

[16] J. Frey and T. Tannenbaum and M. Livny and I. Foster and S. Tuecke, *Condor-G: A Computation Management Agent for Multi-Institutional Grids*,in HPDC10 — Tenth International Symposium on High Performance Distributed Computing, August, 2001, IEEE Press.

[17] F. Berman and R. Wolski, *The AppLeS Project: A Status Report*, 1997.

[18] A. S. Grimshaw, M. J. Lewis, A. J. Ferrari and J. F. Karpovich, *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, Department of Computer Science, University of Virginia, 1998, Technical Report, CS-98–12.

[19] F. Berman, G. Fox and T. Hey, *The Grid: past, present, future*, in Grid Computing—Making the Global Infrastructure a Reality, pp. 9–50, John Wiley & Sons, Ltd, West Sussex, England, 2003.

[20] I. Foster and C. Kesselman, *The Globus project: a status report*, in Future Generation Computer Systems, 15(5–6), pp. 607–621, 1999.

[21] A. Geist, A. Beguelin, J. Dongerra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine*, MIT, 1994.

[22] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT, second ed. , 1999

[23] W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI Message-Passing Interface standard*, in Parallel Computing, 1996, 22(6), pp. 789–828.

[24] G. A. Geist, J. A. Kohl and P. M. Papadopoulos, *PVM and MPI: A comparison of features*, in Calculateurs Parallels, 8(2), 1996.

[25] N. Karonis, B. Toonen and I. Foster, *MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface.*, in Journal of Parallel and Distributed Computing, 2003.

[26] I. Foster, C. Kesselman and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, in Int. J. Supercomputer Applications, 2001.

[27] S. Zhou,X. Zheng, J. Wang and P. Delisle, *Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, in Software—Practise and Experience, 23(12), pp. 1305–1336, 1993.

[28] Sun Microsystems, *Grid-Engine Project*, 2001, `http://gridengine.sunsource.net/`

[29] P. W. Trinder, K. Hammond, H-W. Loidl and S. L. Peyton Jones, *Algorithm + Strategy = Parallelism*, in Journal of Functional Programming, 1998, 8(1), pp. 23–60.

Appendix A: `sumEuler.`

```
module Main(main) where

import System(getArgs)
import Strategies

sumTotient :: Int ->---lower limit of the interval
              Int ->---upper limit of the interval
              Int ->---chunk size
              Int
sumTotient lower upper c =
  sum ( map (sum . map euler) (splitAtN c [upper, upper-1 .. lower])
       ‘using‘ parList rnf)

euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])


relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

mkList :: Int -> Int -> [Int]
mkList lower upper = reverse (enumFromTo lower upper)

splitAtN :: Int -> [a] -> [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs
main = do args <- getArgs
          let
            lower = read (args!!0) :: Int---lower limit of the interval
            upper = read (args!!1) :: Int---upper limit of the interval
            c     = read (args!!2) :: Int---chunksize
          putStrLn ("Sum of Totients between [" ++
                    (show lower) ++ ‘‘.." ++ (show upper) ++ ‘‘] is ‘‘ ++
                     show (sumTotient lower upper c))
```

Edited by: Frédéric Loulergue
Received: October 20, 2005
Accepted: February 1st, 2006