Supporting High-Level Grid Parallel Programming: the Design and Implementation of `Grid-GUM2`

A. D. Al Zain[*], P. W. Trinder, G. J. Michaelson
School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
E-mail: {ceeatia,trinder,greg}@macs.hw.ac.uk

H-W. Loidl
Ludwig-Maximilians-Universität München,
Institut für Informatik, D 80538 München, Germany,
E-mail: hwloidl@informatik.uni-muenchen.de

Abstract

Computational Grids are much more complex than classical high-performance architectures: they have high, and dynamically varying communication latencies, and comprise heterogeneous collections of processing elements. We argue that such a complex and dynamic architecture is extremely challenging for the programmer to explicitly manage, and advocate a high-level parallel programming language, like GpH, where the programmer controls only a few key aspects of the parallel coordination. Such a high-level language requires a sophisticated implementation to manage parallel execution.

This paper presents the design and implementation of `Grid-GUM`*2, a Grid-specific runtime environment for GpH. The core of the design are monitoring mechanisms that collect static and partial dynamic information, and new load management mechanisms that use the information to better schedule tasks on computational Grids. These mechanisms are built on top of a virtual shared memory (VSM)*[1]*, graph-reduction-based computation engine. A systematic evaluation of the implementation, reported elsewhere, shows acceptable scaleup and good speedups on a range of computational Grids.*

1 Introduction

Computational Grids potentially offer low-cost, yet large-scale high performance platform. Computational Grids are most commonly used to execute large numbers of independent sequential programs, e.g. under Condor [7] or LSF Platform [20]. In contrast we consider the *parallel* execution of programs on computational Grids, where all of the computational resources are available to a single large program. The challenge, however, is that the components of the parallel program must communicate and synchronise.

Computational Grids are much harder to utilise effectively for parallelism than a classical high-performance computer (HPC). A classical HPC typically comprises a large number of homogeneous processing elements (PEs), communicating using an interconnect with uniform, and relatively low, latency. Typically PEs and interconnect are dedicated to the sole use of the program for its entire execution. In contrast, a computational Grid is typically *heterogeneous* in the sense that it combines clusters of varying sizes and different clusters typically contain PEs with different performance. Moreover the interconnects are highly variable, with different latencies within, and between, each cluster. Moreover the interconnect between clusters is

[*]This work is supported by Symbolic Computation Infrastructure for Europe EU FP VI I3-026133.

[1]Virtual shared memory is a programming model, allows processors on a distributed-memory machine to be programmed as if they had shared memory

typically both high-latency and shared, and as a consequence communication latency may vary unpredictably during program execution.

We argue that computational Grid architectures are too complex and dynamic for programmers to readily manage at a relatively low-level, e.g. using the common SPMD idiom. We advocate specifing the parallelism at a high-level of abstraction and relying on a sophisticated language implementation to dynamically manage the low-level parallel coordination aspects on the computational Grid. The high-level language we propose is Glasgow parallel Haskell (GpH) [17], and its `GUM` runtime environment has been engineered to deliver good performance on classical HPCs and clusters [15].

Noteworthy features of the original `GUM` design are a virtual shared memory (VSM), that is implemented on top of a portable message passing library. The underlying (sequential) computation engine uses graph-reduction, the favoured mechanism for implementing functional languages. The combination of the single assignment semantics of the abstract machine language and the high degree of optimisation that is provided by the sequential computation engine (and compiler), make an VSM approach, with automatic synchronisation of tasks and distribution of work viable. In particular, the coherence problem over several physical memories is much reduced. The parallel execution model favours the generation of large amounts of parallelism so as to hide high remote access time by switching to other tasks in the meantime. Detailed measurements in [3] give evidence to the efficiency of this mechanism on Grid architectures.

This paper presents the design and implementation of `Grid-GUM2`, a Grid-specific runtime environment for GpH and further development of the above design. `Grid-GUM2` has been implemented using Globus TK2 and MPICH-G2, and a systematic evaluation reported elsewhere shows acceptable scaleup on medium-scale Grids, e.g. a speedup of 28 on 41 PEs located in 3 clusters in Scotland and Germany; and good speedups on all combinations of high/low latency, and homo/hetero-geneous computational Grids [3]. Section 2.1 presents GpH and the GUM implementations. Section 3 presents the core design elements of `Grid-GUM2`, namely monitoring mechanisms that collect static and partial dynamic information, and new load management mechanisms that use the information to better schedule tasks on computational Grids. Related work is discussed in Section 5, and Section 6 concludes.

2 Background

2.1 Glasgow Parallel Haskell (GpH)

GpH is a semi-explicit parallel functional language, enabling the programmer to specify parallelism with relatively little effort using high level parallel coordination constructs. It is a modest and conservative extension of Haskell 98, a non-strict purely-functional programming language [17]. GpH extends Haskell 98 with a parallel composition `par`, and an expression `e1 `par` e2` (here we use Haskell's infix operator notation) has the same value as `e2`. Its dynamic effect is to indicate that `e1` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e2`. Results from `e1`'s evaluation are available in `e2` which shares subgraphs evaluated in `e1` e.g. through common variables. GpH programs also sequence the evaluation of expressions using the `seq` sequential composition. For example a parallel naive `nfib` function, based on the fibonacci function, can be written as follows.

```
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 ‘par‘ (nf1 ‘seq‘ (nf1+nf2+1))
          where nf1 = parfib (n-1)
                nf2 = parfib (n-2)
```

2.2 `GUM` - A Parallel Haskell Runtime Environment

`GUM` is a portable, parallel runtime environment (RTE) for GpH. `GUM` implements a specific DSM model of parallel execution, namely graph reduction on a distributed, but virtually shared, graph. Graph segments are communicated in a message passing architecture designed to provide an architecture neutral and portable runtime environment. Here we describe the key components for a Grid context, namely program initialisation and load distribution, for `GUM` 4.06 using the PVM communications library [10]. Full descriptions of `GUM` are available in [12] and [18].

2.3 `GUM` Thread Management

The unit of computation in `GUM` is a lightweight thread, and each logical PE is an operating system process that co-schedules multiple lightweight threads as outlined below and detailed in [12] and [18]. Threads are automatically synchronised using the graph structure, and each PE maintains a pool of runnable threads. Parallelism is initiated by the *par* combinator. Operationally, when the expression `x 'par' e` is evaluated, the heap object referred to by the variable x is *sparked*, and then e is evaluated. By design sparking a reducable expression, or *thunk* is relatively cheap operation, and sparks may freely be discarded if they become too numerous. If a PE is idle, a spark may be converted to a thread and executed. Threads are more heavyweight than sparks as they must record the current execution state.

2.4 GUM Load Distribution

GUM uses dynamic, decentralised, and blind load management. The load distribution mechanism is designed for a flat architecture with uniform PE speed and communication latency, and works as follow. If (and only if) a PE has no runnable threads, it creates a thread to execute from a spark in its spark pool, if there is one.

If there are no local sparks, then the PE sends a FISH message to a PE *chosen at random*. A FISH message requests work and specifies the PE requesting work. The random selection of a PE to seek work from is termed *blind* load distribution, as no attempt is made to seek work from a 'good' source of work.

If a FISH recipient has an empty spark pool it forwards the FISH to another PE chosen at random. If a FISH recipient has a spark it sends it to the source PE as a SCHEDULE message. If the PE that receives a FISH has a useful spark, it sends a SCHEDULE message to the PE that originated the FISH, containing the sparked thunk packaged with nearby graph. The originating PE unpacks the graph, and adds the newly-acquired thunk to its local spark pool. To maintain the virtual graph, an ACK message is then sent to record the new location of the thunk. A sequence of messages initiated by a FISH is shown in Figure 1.

2.5 GUM Performance

The GUM implementation of GpH delivers good performance for a range of parallel benchmark applications on a variety of parallel architectures, including shared and distributed-memory architectures [16]. GUM's performance is also comparable with other mature parallel functional languages [15].

GUM can also deliver comparable performance to conventional parallel paradigms. For example [15] compares the performance of a GpH and a C with PVM matrix multiplication programs. The program multiplies square matrices of arbitrary precision integers, and the C program uses the Gnu Multi-Precision library and the GNU C compiler. The sequential C program is 5 times faster, but the GpH program has better speedups and on 16 PEs the C+PVM program is just 1.6 times faster than the GpH program. The sizes of the GpH and C+PVM programs differ substantially, though: the C+PVM program is 6 times longer than the GpH program.

2.6 Grid-GUM1

Grid-GUM1 is a port of GUM to the Grid [2]. The key part of the port is to utilise the MPICH-G2 communication library [14] in the GUM communication layer. MPICH-G2 in turn uses the Globus Toolkit2 middleware, as illustrated in Figure 2.

The heterogeneity and high communications interconnect have a direct impact on Grid-GUM1 performance. On heterogeneous computational Grids, or those with high communication latency, Grid-GUM1 only delivers acceptable speedups for low-communication degree programs like queens[2], and little speedup for high-communication degree programs like raytracer[3]. In contrast, on heterogeneous and high communications interconnect environments, Grid-GUM1 fails to deliver accpetable speedup. Extensive experimental evaluation of Grid-GUM1 is available in [2].

3 Grid-GUM2 Design

This section presents the design of Grid-GUM2, an RTE designed to improve performance on the Grid using sophisticated load management. This load management is dynamic, decentralised and mostly passive using complete static and partial dynamic information about PEs, processes and network. The salient features of Grid-GUM2 are *monitoring* and *adaptive load scheduling* mechanisms.

3.1 Monitoring Information

The adaptive load sheduling implemented in Grid-GUM2 requires knowledge about static and dynamic properties at runtime. Hence Grid-GUM2 is designed to provide the necessary information using monitoring mechanisms [1]. For location awareness, Grid-GUM2 observes all PEs in the parallel system, and its own PE using the communication library (MPICH-G2). Acquiring the other data about the current system state of one PE needs sophisticated runtime support. This information is continuously collected by

[2]queens program places queen chess pieces on the chess board so that they do not check each other.
[3]raytracer program calculates a 2D image of a scene of 3D objects by tracing all rays in a window.

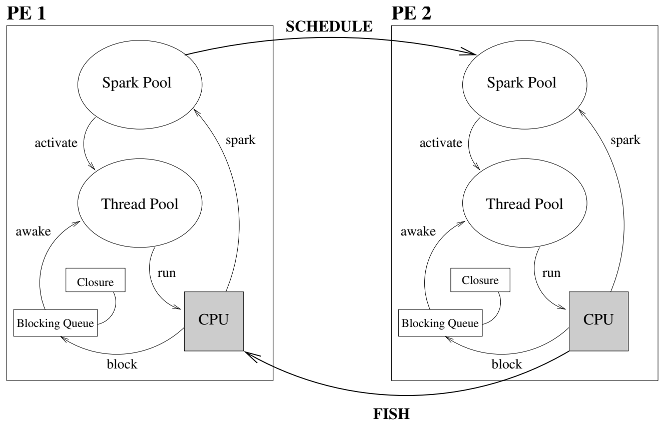

Figure 1. Fish/Schedule/Ack Sequence

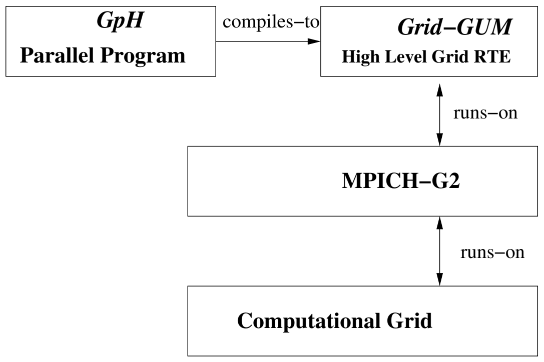

Figure 2. `Grid-GUM`1 System Architecture

PEStatic Table

PE	CPU MHz	IP Address	Sync.Time
1	3343.6	xxx.xxx.149.456	12:04:05.11
2	3343.6	xxx.xxx.149.456	12:04:05.01
⋮	⋮	⋮	⋮
11	1800.9	xxx.xxx.566.149	13:04:15.50
12	1804.0	xxx.xxx.566.149	13:04:15.23

PEDynamic Table

PE	Load	Latency (msec.)	Timestamp
1	0.89	1.59	12:05:05.11
2	1.44	1.02	12:05:03.78
⋮	⋮	⋮	⋮
11	1.23	10.07	12:04:35.50
12	0.21	12.12	12:04:35.23

Figure 3. Example of *PEStatic* and *PEDynamic* Table

the monitoring mechanism and held in local tables *PEStatic* and *PEDynamic*, illustrated in Figure 3,

Static information about PEs participating in the computation will remain constant during a program execution and is collected only once at the beginning of the execution, when all PEs synchronise in a startup barrier. Synchronisation and the respective local time are necessary for precise dynamic latency calculation.

While static information is fairly easy to retrieve, it is more complex to acquire dynamic information about the participating PEs. In short, each PE has only a partial picture of the state on other PEs. These partial pictures are exchanged by adding the dynamic information to the existing messages that need to be exchanged anyway. This lightweight approach avoids a regular broadcast of load information to all other PEs, which would be prohibitively expensive. To ensure consistency, messages between PEs are timestamped. Every time a PE receives a message, it will re-calculate all dynamic inter-PE information (e.g. latency), and it will update its table collecting dynamic intra-PE information (e.g. PE load). Even though this does not guarantee that all highly loaded PEs will be identified as such, based on the local information, it drastically increases the probability that at least one such PE is known, which is sufficient to perform adaptive load balancing.

3.2 Adaptive Load Distribution

`GUM` load management works well on closely connected systems (as discussed in Section 2.4) but, as measurements show, does not scale well on Grid architectures [2]. To address shortcomings of this mechanism on wide-area networks, we make modifications to the thread management component. The key concept in these changes is *adaptive load distribution*: the behaviour of the runtime should adjust to both the static configuration of the computational Grid and to dynamic aspects of the execution, such as the load of the individual processors.

The adaptive load distribution deals with: *startup*, *work locating*, and *work request handling* mechanisms. The key new policies for adaptive load distribution are that work is only sought from PEs which are known to be relatively heavily loaded; to give preference to local cluster resources; to send additional work to relatively powerful cluster.

During startup synchronisation, a suitable PE for the main computation is selected.`Grid-GUM`2 starts the computation in the 'biggest' cluster, i.e. the cluster with the largest sum of CPU speeds over all PEs in the cluster, a policy which is equally easy to implement.

The pseudo code in Figures 5 and 4 shows how an idle processor lPE chooses the target processor for a

```
FUNCTION choosePE(compMap, orgRatio) {
  sortBy(compMap,latency,ascending)
  WHILE (rPE=get_next_PE(comMap))
      rRatio= pe_ratio(rPE)
      IF orgRatio > rRatio THEN
        return rPE
      FI
  END
 return mainPE
END
```

Figure 4. Choose PE

```
IF get_status(lPE)==idle THEN
  IF size(thread_pool(lPE)) > 0 THEN
   evaluate (next_thread(thread_pool(lPE)))
 ELSIF size(spark_pool(lPE)) > 0 THEN
    s = activate(new_spark(lPE));
        add_to_thread_pool(s,lPE)
 FI
 IF size(spark_pool(lPE)) < watermark THEN
   update(load, PEDynamic)
   localRatio = (get_speed(lPE)/get_load(lPE))
  pe=choosePE(compMap, localRatio)
 send([FISH, PEDynamic], rPE)
```

Figure 5. Work Locating

work locating FISH message, based on latency, load and CPU speed information. The ratio between CPU speed and load is computed for all PEs in the compMap table, representing the Grid environment. Ratios are checked against the local ratio, and preference is given to nearby PEs by sorting compMap by latency in ascending order. The target PE will be a nearby PEs that recently exposed a higher load than the sender.

This policy avoids computation hot spots in the system, possibly at the expense of creating a short-lived communication hot spots. But more importantly, the load balance is improved and overall idle time of the PEs is reduced. This policy also decreases the total amount of communication through high-latency communication, which improves overall performance.

The work request handling (Figure 6) introduces a mechanism to minimise the (high-latency) communications between different clusters. It is based on information about the recipient's (lPE) and the originator's (orgPE) *clusters*, statically determined by their IP addresses and leading to an accumulated "cluster power", the sum of all CPU speeds. If the work request has originated from a relatively powerful cluster (orgClustRatio>lClustRatio), then multiple pieces of work ("sparks") will be sent in SCHEDULE messages. Otherwise, the FISH message can be served as usual. Figure 6 shows how this policy can be included in the message handling. After updating the dynamic information in the PEDynamic table, the power-load ratio of the sender cluster is compared to that of the receiver cluster. If the sender is stronger one spark will be sent. If the receiver is stronger, but the sender's cluster is stronger than the receiver's cluster, then several sparks (determined by a call to numSparks) will be sent. The idea behind this policy is to send a bigger amount of work to powerful clusters in one go. Although the sender was weak, the local work requesting mechanism in that cluster will assure a swift redistribution of this batch of work. Note that in this latter case the system effectively switches from a passive load distribution policy (work is only sent if it has been requested by another PE) to an active one (work is sent speculatively). All these mechanisms together achieve a better balance of the load on a low-latency cluster, which is even more important than on a high-latency cluster, where additional work can be retrieved rather cheaply.

4 Measurements

This section summarises the performance of the `Grid-GUM2` load distribution mechanism on the most challenging Grid configuration, namely a high-latency heterogeneous computational Grid. The measurements in this section are made on three heterogeneous Beowulf clusters: Edin1 and Edin2 connected over a low-latency interconnect, and Muni connected with the other two clusters over a high-latency interconnect, as specified in Tables 1 and 2. Because of the relative cluster sizes, many configurations have 6 Edin1 PEs for every Muni PE.

The experiments have the limitation of the number of PEs available at the cooperating sites: Edin1 (E) 30 PEs, Edin2 (E_2) 5 PEs, and Muni (M) 6 PEs.

The Grid configurations measured in this section have very similar mean CPU speeds and latencies, namely 676Mhz and approximately 9.2ms. Likewise the configurations have very similar variations in CPU speed and communications latency, namely approximately 360MHz and 15.5ms respectively. Moreover, the input size to the `raytracer` and `parFib` programs is large, and hence it is not possible to obtain a sequential runtime. As a result the sequential runtime, and hence both relative speedups and parallel efficiency, are computed from the runtime on a 7 PE configuration. That is the `raytracer` runtime from the $7E$ row of Table 3, the `parFib` from the $6E1M$ `Grid-GUM2` row of Table 4.

```
rM = received_message(orgPE,lPE)
IF message_type(rM)==FISH THEN
  update((orgPE |-> get_latency(rM)), PEDynamic)
  update((orgPE |-> get_load(rM)), comMap)
  lRatio= pe_ratio(lPE); orgRatio = pe_ratio(orgPE)
  IF size(spark_pool(lPE))== 0 THEN
     send([FISH, PEDynamic], choosePE(compMap,orgRatio))
 ELSE /* local work available */
    IF orgRatio > lRatio THEN
      IF clusterId(orgPE) == clusterId(lPE) THEN
         s = get_next_sparks(1,spark_pool(lPE)))
         send([SCHEDULE, s, PEDynamic], orgPE)
      ELSE
        lClusterRatio = cluster_ratio(clusterId(lPE))
        orgClusterRatio = cluster_ratio(clusterId(orgPE))
        IF orgClusterRatio > lClusterRatio THEN
           s = get_next_sparks(numSparks(orgClusterRatio,lClusterRatio),
                                         spark_pool(lPE))
          send([SCHEDULE, s, PEDynamic], orgPE)
        ELSE
          send([FISH, PEDynamic], choosePE(compMap,orgRatio))
        FI /* forward FISH */
   ELSIF orgRatio <= lRatio THEN
    IF message_age(rM) > age_limit() THEN
      send([NULL, PEDynamic], orgPE)
    ELSE
      pe = choosePE(compMap, orgRatio)
      send([FISH, PEDynamic], pe)
    FI
FI  /* message==FISH */
```

Figure 6. Work Request Handling

	CPU	Cache	Memory	PEs
	MHz	kB	Total kB	
Edin1	534	128	254856	32
Edin2	1395	256	191164	6
Muni	1529	256	515500	7

Table 1. Beowulf Cluster Architectures

	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
Muni	35.8	35.8	35.8	32.8	0.13

Table 2. Inter-Cluster Latencies (ms)

4.1 `raytracer`

The `raytracer` is a realistic parallel program with limited amounts of highly-irregular parallelism and a relatively high communication degree. Table ?? compares the scalability of `raytracer` program under `GUM` and `Grid-GUM1`. The table shows that `GUM` and `Grid-GUM1` deliver very similar performance up to 28PEs, even although `Grid-GUM1` is executing on a high-latency heterogeneous computational Grid. More significantly, the last two cases show that when the size of the local cluster limits the `GUM` speedups, `Grid-GUM1` can scale further using PEs in a remote cluster.

Table 3 compares the scalability and parallel efficiency of the `raytracer` program under `Grid-GUM1` and `Grid-GUM2` on a high latency heterogeneous computational Grid. The efficiency comparison of the two cluster results relies on the similarity of the architectures, i.e. 6 Edinburgh PEs for every Munich PE, and obviates the requirement for a sophisticated calculation of heterogenous efficiency. The table shows that `Grid-GUM2` always improves on `Grid-GUM1` performance. Moreover, although the speedup improvement is modest on small Grids it increases with Grid size. For example on the largest, 41-PE, configuration `Grid-GUM2` gives a 46% improvement: i.e. a runtime of 1133s compared with 1652s for `Grid-GUM1`.

Although `Grid-GUM2` is always more efficient than `Grid-GUM1`, the absolute efficiency of `Grid-GUM2` falls significantly to just 38% on a 35 PE cluster. While some of the loss of efficiency is attributable to the high-level DSM programming model, reader's should recall that `raytracer` is a challenging program, i.e. exhibiting highly-irregular parallelism and high levels of communication, executing on a challenging architecture: a high latency heterogeneous Grid. Moreover Table 4 reports rather better efficiency for a less challenging program.

4.2 `parFib`

In contrast to the realistic `raytracer` program, `parFib` is an ideal parallel program with very large potential parallelism and a low communication degree. Table 4 compares the scalability and efficiency of `parFib` under `Grid-GUM1` and `Grid-GUM2` on a high latency heterogeneous computational Grid. It shows that both `Grid-GUM1` and `Grid-GUM2` deliver good, and very similar speedups. The speedups is excellent up to 21 PEs, but declines thereafter. Speedup is still increasing even between 35 and 41 PEs, with a maximum speedup of at least 27 on 41 PEs. `Grid-GUM2` is again always more efficient than `Grid-GUM1`. Moreover while the drop in absolute efficiency to 65% on 35 PEs is substantial it is far less than for the challenging `raytracer`. Section ?? suggests that even better speedups and efficiency would be obtained on either an homogeneous Grid, or a low latency Grid.

The good `Grid-GUM1` performance reported in Table 4 demonstrates that sophisticated load distribution is not required for `parFib`. That the `Grid-GUM2` performance is so similar to the `Grid-GUM1` performance

Case	No PEs	Config.	`Grid-GUM1`			`Grid-GUM2`		
			Rtime	Spdup	Eff.	Rtime	Spdup	Eff.
1	7	6E1M	2530	7	97%	2470	7	100%
2	14	12E2M	2185	8	56%	1752	10	70%
3	21	18E3M	1824	10	45%	1527	12	53%
4	28	24E4M	1776	10	34%	1359	13	45%
5	35	30E5M	1666	11	29%	1278	14	38%
6	41	$5E_2$30E6M	1652	11		1133	16	

Table 3. `raytracer`**: Scalability**

Case	No PEs	Config.	`Grid-GUM1`			`Grid-GUM2`			Impr%
			Rtime	Spdup	Eff.	Rtime	Spdup	Eff.	
1	7	6E1M	3995	7	93%	3737	7	100%	0%
2	14	12E2M	1993	14	93%	2003	14	93%	0%
3	21	18E3M	1545	18	80%	1494	19	83%	5%
4	28	24E4M	1237	23	75%	1276	22	73%	-4%
5	35	30E5M	1142	24	65%	1147	24	65%	0%
6	41	$5E_2$30E6M	1040	27		1004	28		4%

Table 4. `parFib`**: Scalability**

shows that even on medium-scale computational Grids, the overheads of `Grid-GUM2`'s load distribution mechanism remain minimal.

5 Related Work

Currently computational Grids are most commonly used to execute large numbers of independent sequential programs, and a number of systems exist to support this model including Condor [7], Maui [13], Legion [11]. In such systems the computational power available for a single program is bounded by the speed of the fastest PE in the Grid. In contrast the challenge we address is to effectively execute components of a single program in parallel on a computational Grid. Under parallel evaluation the computational power available to a program is bounded by the sum of all PEs in the Grid.

High-level coordination languages/frameworks are being used to compose Grid applications from large scale components, for example the ASSIST [4] and GrADS [8] projects. The key idea is that the coordination language or framework automatically manages the Grid complexities like resource heterogeneity, availability, network latency. The components, which may be sequential or parallel, require minimal changes to be deployed on the Grid. In contrast our approach describes the computation, as well as the coordination in a single high level language, Glasgow parallel Haskell (GpH) [17].

Algorithmic skeletons are being used to provide high-level parallelism on computational Grids. The essence of the idea is to provide a library of higher-order functions that encapsulate common patterns of parallel Grid computation. Parallel applications are constructed by parameterising a suitable skeleton with sequential functional units. Examples of this approach include work groups lead by Danelutto [5], Cole [9] and Gorlatch [6]. In contrast to the fixed set of skeletons, it is possible to define new coordination constructs in GpH, as outlined in section 2.1.

Perhaps the approach most closely related to ours is to port a high level distributed programming language to the Grid. Both Ibis [19] and Gorlatch's group [6] port Java to the Grid and use Remote Method Invocation (RMI) as the programming abstraction. Coordination in GpH is higher-level than RMI and more extensible.

An early design of `Grid-GUM2` has been published in [1]. A systematic evaluation of the performance of `Grid-GUM2` and `Grid-GUM1` in combinations of high/low latency, and homo/hetero-geneous computational Grids appear in [3] and [2] respectively.

6 Conclusion

We have presented the design of `Grid-GUM2`, a sophisticated Grid-specific runtime system for the GpH high-level parallel language. The key elements of the design are monitoring mechanisms that collect static and partial dynamic information, and new, adaptive load management mechanisms. These provably improve performance on Grid architectures, while maintaining the original virtual shared memory programming paradigm and using agressive optimisations on the sequential code. Both monitoring and load management are bespoke lightweight mechanisms that do not use generic Grid services. However communication between, and authentication of, the PEs is provided by Grid connective layer services, namely MPICH-G2, Globus TK2, and RSL.

Measurements of `Grid-GUM2` show that it delivers greatest performance improvements on the most challenging architectures, e.g. a 60% improvement on a heterogeneous high latency computational Grid [3]. On low latency homogeneous computational Grids, `Grid-GUM2` delivers an excellent relative speedup of 17.6 on 16 PEs for a simple program (parallel factorial). On low latency heterogeneous computational Grids `Grid-GUM2` improves the performance most of our benchmark programs On high latency homogeneous and heterogenous computational Grids `Grid-GUM2` improves the performance of all our benchmark programs on all Grid configurations measured. In short, `Grid-GUM2`'s dynamic adaptive load management techniques are effective as they improve or maintain the performance of all the benchmark programs on all Grid configurations.

Grid-GUM2 has a number of limitations. It is designed to work in a *closed* computational Grid, i.e. it is not possible for other machines to join the computation after it has started. Moreover it is tuned for a classical high-performance setup, i.e. to be most effective on: *a)* dedicated computational Grid where only one program is executed at a time, and *b)* a non-preemptive environment: each program executes to completion without interruption.

There are several avenues to extend this research. One avenue is to implement larger parallel programs, and our current work entails parallelising large computer algebra computations as part of the *SCIEnce* project. A second research avenue is to investigate the scalability of Grid-GUM2 on large computational Grids, e.g. with 100s of PEs. Such a Grid is likely to be heterogeneous and high-latency, and we hope to make these measurements in the SCIEnce project.

Acknowledgement

We would like to thank Kevin Hammond for the constructive feedback.

References

[1] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell . In *International Conference on Computational Science (ICCS 2005)*, LNCS. Springer, 2005.

[2] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *Journal of Scalable Computing: Practice and Experience*, 7(3):9–26, 2006.

[3] A. D. Al Zain, P. W. Trinder, G. J. Michaelson, and H.-W. Loidl. Evaluating a High-Level Parallel Language (GpH) for Computational Grids. *IEEE Transactions on Parallel and Distributed Systems*, 2007. TO APPEAR.

[4] M. Aldinucci and M. Danelutto. Advanced skeleton programming systems. *Parallel Computing*, 2006. to appear.

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

[6] M. Alt and S. Gorlatch. Adapting java rmi for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2005.

[7] J. Basney and M. Livny. *High Performance Cluster Computing*, volume 1, chapter Deploying a High Throughput Computing Cluster. Prentice-Hall, 1999.

[8] F. Berman, A. Chien, J Cooper, K.and Dongarra, I Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, and L.and WolskiMatteo R. Torczon. The GrADS Project: Software Support for High-Level Grid Application Development. *Int. Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[9] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[10] Al Geist, Adam Beguelin, Jack Dongerra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT, 1994.

[11] A.S. Grimshaw and W.A. Wulf. The Legion Vision of a World-Wide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.

[12] K. Hammond, J.S. Mattson Jr., A.S Partridge, S.L. Peyton Jones, and P.W. Trinder. GUM: a Portable Parallel Implementation of Haskell. In *IFL'95 — Intl Workshop on the Parallel Implementation of Functional Languages*, September 1995.

[13] D.B. Jackson. Advanced Scheduling of Linux Clusters using Maui. In *USENIX'99*, 1999.

[14] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: a grid-enabled implementation of the message passing interface. *Journal Parallel Distributed Computing*, 63(5):551–563, 2003.

[15] H-W. Loidl, F. Rubio Diez, N.R. Scaife, K. Hammond, U. Klusik, R. Loogen, G.J. Michaelson, S. Horiguchi, R. Pena Mari, S.M. Priebe, A.J. Rebon Portillo, and P.W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3):203–251, 2003.

[16] H-W. Loidl, P. W. Trinder, K. Hammond, S. B. Junaidu, R. G. Morgan, and S. L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.

[17] 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.

[18] 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 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.

[19] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.

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