Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs

A. D. Al Zain P. W. Trinder G. J. Michaelson H-W. Loidl

Abstract— **Computational Grids potentially offer low cost, readily available, and large-scale high-performance platforms. For the parallel execution of programs, however, computational GRIDs pose serious challenges: they are heterogeneous, and have hierarchical and often shared interconnects, with high and variable latencies between clusters.**

This paper investigates whether a programming language with high-level parallel coordination and a Distributed Shared Memory model (DSM) can deliver good, and scalable, performance on a range of computational GRID configurations. The high-level language, Glasgow parallel Haskell (GpH), abstracts over the architectural complexities of the computational GRID, and we have developed *GRID-GUM*2, a sophisticated grid-specific implementation of GpH, to produce the first high-level DSM parallel language implementation for computational GRIDs.

We report a systematic performance evaluation of *GRID-GUM*2 on combinations of high/low and homo/hetero-geneous computational GRIDs. We measure the performance of a small set of kernel parallel programs representing a variety of application areas, two parallel paradigms, and ranges of communication degree and parallel irregularity. We investigate *GRID-GUM*2's performance scalability on medium-scale heterogeneous and high-latency computational GRIDs, and analyse the performance with respect to the program characteristics of communication frequency and degree of irregular parallelism.

Index Terms— **Concurrent, distributed, and parallel languages; Grid Computing; Functional Languages;**

I. INTRODUCTION

HARDWARE price/performance ratios make cluster computing increasingly attractive. Moreover, emerging GRID technology [1] offers the potential to connect these ubiquitous clusters to form a computational GRID: a low-cost, yet large-scale high performance platform. Clusters and computational GRIDs are most commonly used to execute large numbers of independent sequential programs, e.g. under Condor [2] or LSF Platform [3]. For such systems the computational resource available to a single program is bounded by the most powerful machine in the network. In contrast we consider the *parallel* execution of a single program on a computational GRID, where the computational resource available to a program is the sum of all the resources on the network. The key technical distinction from high-throughput computing is the dependencies between the components of the parallel program: they 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. An SPMD model of parallel programming, supported by standard communication libraries like MPI [4] is the dominant parallel programming paradigm. 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 typically both high-latency and shared, and as a consequence communication latency may vary unpredictably during program execution. We argue that such an architecture is too complex and dynamic for programmers to readily manage at a relatively low-level, e.g. using SPMD.

Despite the challenges, the attraction of computational GRIDs as low cost, readily available and large-scale high-performance architecture has encouraged a number of groups to develop parallel execution environments. The most common approach is to specify the parallelism at a low level, although some higher-level parallel models have been used, e.g. algorithmic skeletons [5], as detailed in Section II.

We advocate specifying parallelism on computational GRIDs in a language with high-level coordination and a Distributed Shared Memory model (DSM). Such a language abstracts over the architectural complexities of a computational GRID: the programmer controls only a few key parallel coordination aspects, and the remaining coordination aspects and virtual shared memory are dynamically managed by a sophisticated runtime environment. The language investigated here is Glasgow parallel Haskell (GPH) [6], and its GUM runtime environment has been engineered to deliver good performance on classical HPCs and clusters [7]. We have previously shown that a direct port of GUM to a the GRID, *GRID-GUM*1, only reliably provides good performance for low-latency homogeneous GRIDs, and that load management limits performance [8]. To overcome the limitations of *GRID-GUM*1 we have designed and implemented *GRID-GUM*2 with novel dynamic load scheduling mechanisms that record and use both static and dynamic information about the computational GRID. We have also reported preliminary performance measurements on heterogeneous computational GRIDs [8].

This paper investigates whether a high-level DSM parallel programming paradigm can deliver good, scalable, performance for a variety of applications on combinations of high/low and homo/hetero-geneous computational GRIDs. That is, we present a systematic evaluation of the *GRID-GUM*2 implementation of GPH, the first virtual shared-memory parallel language for computational GRIDs. The investigation uses six kernel parallel programs from a range of application areas, e.g. AI and Symbolic Algebra, with data-parallel and divide-and-conquer parallel paradigms, and with a range of dynamic properties like communication frequency and degrees of irregular parallelism.

The remainder of the paper is structured as follows. Section II describes related work. Section III describes the GPH language and its GUM runtime environment designed for a single HPC or cluster. Section IV summarises the design and performance of an initial port of GUM to the GRID, *GRID-GUM*1. Section V outlines the design of *GRID-GUM*2 with new load management mechanisms. Section VI evaluates the performance of *GRID-GUM*2 on a low latency heterogeneous, and homogeneous, computational GRIDs. Section VII evaluates the performance of *GRID-GUM*2 on high latency heterogeneous, and homogeneous, computational GRIDs. Section VIII investigates the performance scalability of the *GRID-GUM*2 load distribution mechanisms on high-latency heterogeneous computational GRIDs. Section IX analyses the relative performance of all programs under *GRID-GUM*1 and *GRID-GUM*2 on combinations of low/high latency and homo/heterogeneous computational GRIDs with respect to their communications behaviour and degree of irregular parallelism. Section X concludes.

II. Related Work

A. High-level Parallel Coordination

A parallel program must specify both *computation* - a correct and efficient algorithm, and *coordination* - how to organise the computations across the PEs. Coordination typically includes aspects such as thread creation, placement, and synchronisation. The computation aspect of a parallel program may be specified at a range of levels of abstraction, e.g. relatively low level like assembler or C, or at a high level like SML or Haskell98.

Like the computation aspect, the coordination aspect of a parallel program may be specified at a range of levels of abstraction, and we use the characterisation of coordination abstraction levels from [9]. In a language with *explicit parallelism*, a programmer may explicitly create and place each thread, and communicate and synchronise between threads. For example, the MPI [4] and PVM [10] libraries support coordination at this level. In languages with *semi-explicit parallelism* like GPH or Eden, the programmer specifies only a few key coordination aspects, e.g. what threads to create, and the language implementation automatically manages the remaining coordination aspects. In an *implicitly-parallel* language like High Performance Fortran [11] or PMLS [12], the programmer specifies no coordination aspects as the parallelism is implicit in the language semantics.

The great advantage of high-level, i.e. semi-explicit or implicit, parallel coordination is that it frees the programmer from specifying low-level coordination details. The disadvantages are that automatic coordination management complicates the operational semantics, makes the performance of programs opaque, is hard to implement, and is frequently less effective than hand-crafted coordination. In these languages, the low-level coordination may be managed solely by the compiler as in PMLS [12], solely by the runtime environment as in GPH [13], or by both as in Eden [14]. Whichever mechanism is chosen, the implementation of sophisticated automatic coordination management is arduous, and there have been many more designs for semi-explicit and implicit parallel languages than well-engineered implementations.

B. Computational GRIDs

The GRID is an emerging large-scale distributed computing architecture that enables the collaborative use of computing resources owned and managed by multiple organisations [15]. Multiple networked machines, often from different administrative domains, are linked into a virtual architecture. The resources of any of the networked machines are available to computations on the virtual architecture, as governed by service level agreements. The architecture is hierarchical with a number of layers. The Globus [16] and Legion [17] projects have been the most important realisations of the GRID infrastructure.

GRIDs are used for a variety of purposes, including on-demand computing, and collaborative computing [18]. By this classification we employ *computational* GRID*s*, which aggregate substantial computational resources to tackle problems that cannot be solved on a single system. A computational GRID typically comprises a number of high performance computers, often clusters, connected by a shared wide area network. Such an architecture has a number of challenging properties. It is *heterogeneous* in that the number of PEs, and the speed of the PEs, in each cluster may be different. There is an *hierarchy* of communication latencies, with communication to PEs at remote clusters being slowest, to PEs at nearby clusters being slow, and to PEs within the same cluster being fastest. Moreover, as the wide area network is shared, communication latency may vary unpredictably during program execution.

Currently computational GRIDs are most commonly used to execute large numbers of independent sequential programs, supported by a number of systems including Condor [2], Maui [19], Legion [17]. In such systems the computational power available for a single program is bounded by the speed of the fastest PE in the GRID. Moreover, ASSIST and KOALA are prototype systems. 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 sum of all PEs in the GRID.

The dominant parallelism paradigm for classical high performance computers is the Single Program Multiple Data (SPMD) [20]. The paradigm is supported by standard communication libraries like MPI [4], giving portability between parallel architectures. In SPMD, and related paradigms like BSP [21], all PEs are initialised to the same set of invocable processes and no computations are transferred dynamically. Instead, the same effect is achieved by dynamically changing the patterns of process invocation across PEs.

An SPMD approach is impractical for computational GRIDs where, in principle, an arbitrary number of PEs may be available to a program. Populating all potential PEs is very wasteful. A first alternative is true dynamic process mobility, but the process granularity for an arbitrary program, especially in a typical coarse grained imperative parallel program, may not be suitable.

C. *Low-level* GRID *Parallelism*

A number of paradigms that are more dynamic than SPMD have been proposed for GRID Parallelism, the majority requiring the programmer to provide dynamic low level coordination. Harness [22] (Heterogeneous Adaptable Reconfigurable Networked Systems) focuses on dynamic, adaptive resource management and even provides facilities for dynamically splitting and merging of distributed virtual machines. Compared to the more classical use of static machine configurations over the lifetime of a parallel program, this approach provides increased scalability of the system, combining heterogeneous sets of machines. Within the context of GRID-computing, Harness supports defining a personalised subset of a GRID-infrastructure and treating it as a unified network. Furthermore, it is possible to use plug-ins for system components such as job scheduling or memory management, effectively generating instances of the virtual machine customised for the underlying architecture.

The ConCert system [23] has a similar philosophy to GPH, using ML as a high level computation language. The Hemlock compiler translates an ML subset to machine code for execution on a computational GRID. In contrast to our work, however, the parallel coordination in ConCert is largely explicit, with primitives to explicitly spawn and synchronise tasks. This reflects ConCert's distributed memory model implemented by mobile code units (chords). Where the DSM parallel graph rewriting enables a relatively simple denotational and operational semantics for GPH [24], ConCert uses a modal lambda calculus [25].

To achieve good parallel performance on a variety of different machines, the AEOS paradigm (Automated Empirical Optimisation of Software) has been proposed [26]. The essence of this paradigm is to provide several implementations of an operation, and to use empirical data such as runtime measurements, to decide which version to choose. For example, to select cut-off values in recursive functions depending on processor speed. However, in the AEOS paradigm the adaption of the software has to be done by a program, not automatically by the system, as we propose in our research.

We argue that low-level, or explicit, parallel programming paradigms are not appropriate for GRID parallelism computing as the architecture is too complex and dynamic for programmers to readily manage.

D. *Distributed Shared Memory*

One means of providing high-level coordination is to abstract over the memory architecture of a distributed system, i.e. to enable a thread at one PE to transparently access data residing on other PEs. Such a Distributed Shared Memory (DSM) model may be implemented in hardware, by the operating system, or by a programming language. There are a large number of research systems, and [27] gives a useful summary, classified by the unit of memory managed: i.e. location, page, or object.

The key issue with DSM systems is to efficiently maintain a coherent view of the 'shared' memory in the presence of concurrent updates on multiple PEs. A coherence protocol, chosen in accordance with some consistency model, maintains memory coherence. For example MESI is a simple and well-known coherence protocol, named after the memory object tags used: Modified, Exclusive, Shared and Invalid. Because declarative languages like GPH and single-assignment languages restrict where updates can occur, their coherence protocols can be far simpler than in conventional languages that allow unrestricted updates.

Because the costs of maintaining consistency rise with the number of PEs, DSM has previously been used mainly on clusters, i.e. relatively small scale systems. Example cluster DSM systems include Kerrighed [28] and TreadMarks [29]. Recently there has been considerable research interest in DSM systems for various types of GRIDs, including computational GRIDs. For example Teamster is a DSM system for computational GRIDs with rather low-level coordination, and thus far only measured on small-scale GRIDs [30]. In contrast GPH has the potential to utilise large scale computational GRIDs, and we report measurements on medium-scale GRIDs in section VIII.

Our GPH language supports a DSM model, and research contributions of this paper include proposing mechanisms for supporting DSM on the dynamic heterogeneous computational GRID architectures, and measuring how well such a DSM model scales on a computational GRID. A GPH program is represented as a graph that the GUM runtime environment maintains in distributed virtual shared-memory. Parallelism is introduced by rewriting multiple graph nodes simultaneously on multiple PEs. The coherence of the graph is maintained using specific graph-rewriting protocols, e.g. blocking any thread that demands the value of a graph node that is currently under evaluation.

Our GPH language has the potential to utilise large scale computational GRIDs, and we report measurements on medium-scale GRIDs in section VIII

E. *Other High-level* GRID *Parallel Paradigms*

Currently there is much interest in developing high-level paradigms that reduce the effort of GRID parallel programming. Much of the work is relatively immature, with systems currently under development, or being prototyped. A range of high-level paradigms are being explored, as outlined below.

High-level coordination languages/frameworks are being used to compose grid applications from large scale components, for example the ASSIST [31] and GrADS [32] 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) [6].

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 [33] [34], Cole [35] and Gorlatch [5]. In contrast to the fixed set of skeletons, it is possible to define new coordination constructs in GPH, as outlined in section III-A.

Perhaps the approach most closely related to ours is to port a high level distributed programming language to the GRID. Both Ibis [36] and Gorlatch's group [37], [38] 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.

Our approach is unique both in adopting a DSM model, and in specifying parallelism in a high-level language, GPH. GPH abstracts over the architectural complexities of a computational GRID. That is, the programmer controls only a few key parallel coordination aspects using high-level evaluation strategies, as outlined in section III-A. The remaining coordination aspects are dynamically managed by a sophisticated runtime environment, *GRID-GUM*2, specifically designed for computational GRIDs. GPH provides higher-level coordination than the other GRID parallel programming languages described in the previous section.

III. GPH AND GUM

A. *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 [6]. 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)
```

Higher-level coordination is provided using *evaluation strategies*: higher-order polymorphic functions that use `par` and `seq` combinators to introduce and control parallelism. For example, `using` applies a strategy to an expression to control its evaluation.

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

Hence the `parMap` parallel map function below applies the function `f` to all of the elements of the list `xs` in parallel. `parMap` is implemented using the `parList` and `rnf` strategies. The `parList` function evaluates the elements of a list in parallel to the degree specified by its argument, in this case, to normal form using the `rnf` strategy. `parList` and `rnf` have a straightforward implementations using `par` and `seq`.

```
parMap f xs = map f xs `using` parList rnf
```

Specifying parallel coordination at such a high level substantially frees the programmer from considering specific aspects of teh underlying architecture. We argue that this is of great benefit for computational GRIDs where the architecture is very complex. As a more substantial example Appendix D shows the GPH `sumEuler` program used in the measurements in later sections. Here the programmer does not need to adapt the program to different computational GRID architectures, and only needs to add the architetcure neutral evaluation strategy in the last line of the `sumTotient` function. A thorough account of how to engineer efficient parallel programs in GPH is given in [39]. The cost of providing the programmer with such high-level abstraction is that GPH requires an elaborate runtime environment to dynamically manage parallel execution on complex architectures, and these are described next.

B. 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 [40] and [13].

C. GUM *Program Initialisation*

When a GPH program is launched under GUM, it initially creates a PVM manager task, whose job is to control startup and termination. This manager task then spawns the required number of logical PEs as PVM tasks, which PVM maps to the available processors. Each PE task then initialises itself: processing runtime arguments, allocating heap etc. Once all PE tasks have initialised, and been informed of each other's identity, one of the PE-tasks is nominated *at random* as the *main PE*. The main PE then begins executing the main thread of the Haskell program.

D. 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 [40] and [13]. 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.

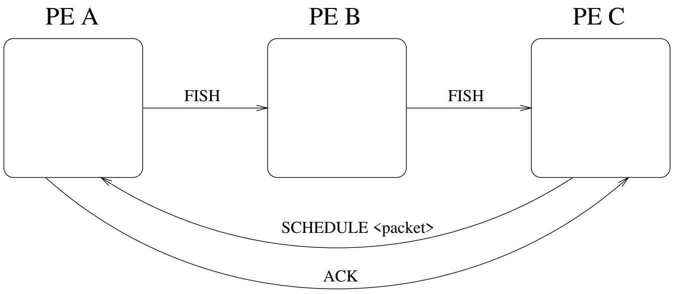

Fig. 1. Fish/Schedule/Ack Sequence

E. 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.

F. 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 [39]. GUM's performance is also comparable with other mature parallel functional languages [7].

GUM can also deliver comparable performance to conventional parallel paradigms. For example [7] 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.

program	Runtime Seq sec	Runtime 16 PE sec	Speedup
`parFib`	465.1	26.3	17.6
`sumEuler`	1598.1	188.1	8.4
`raytracer`	2782.7	301.7	9.2
`linSolv`	828.9	112.2	7.3
`matMult`	916.3	292.6	5.0
`queens`	2816.4	567.8	6.2

TABLE I
*GRID-GUM*1 SPEEDUPS ON A 16-PE HOMOGENEOUS LOW-LATENCY COMPUTATIONAL GRID

IV. *GRID-GUM*1

A. *GRID-GUM1 Architecture*

*GRID-GUM*1 is a port of GUM to the GRID [8]. The key part of the port is to utilise the MPICH-G2 communication library [41] in the GUM communication layer. MPICH-G2 in turn uses the Globus Toolkit middle-ware, as illustrated in Figure 2.

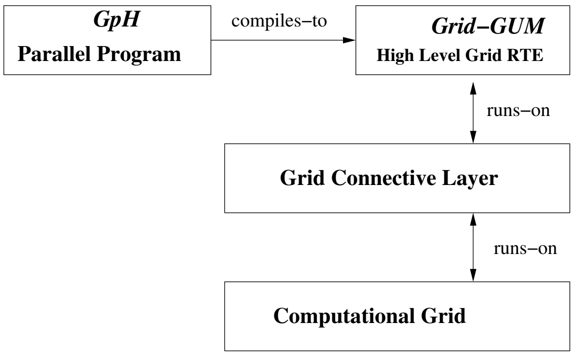

Fig. 2. *GRID-GUM*1 system architecture

B. *GRID-GUM1 Performance*

The following section summarises *GRID-GUM*1 results from [8]. It reports measurements of the suite of programs characterised in Appendix B, where a key characteristic of the programs is the *communication degree*, i.e. the number of messages transmitted per unit execution time. The programs are measured on the collection of GRID-enabled Beowulf clusters specified in Appendix A.

Table I shows that for programs with a sufficiently large execution time, *GRID-GUM*1 can deliver good speedups on homogeneous computational GRIDs with relatively low communication latency. The measurements are performed on the

Config.	Mean CPU (MHz)	raytracer Rtime Sec.	raytracer Speedup F	raytracer Speedup S	queens(13) Rtime Sec.	queens(13) Speedup F	queens(13) Speedup S
FFFFF	1816	376.7	4.0	12.9	181.1	4.0	12.8
FFFFS	1639	422.9	3.5	11.5	254.5	2.8	9.1
FFFSS	1462	519.2	2.9	9.4	544.9	1.3	4.2
FFSSS	1286	615.3	2.4	7.9	530.1	1.3	4.3
FSSSS	1109	755.6	1.9	6.4	577.7	1.2	4.0
SSSSF	1109	850.7	1.7	5.7	560.5	1.2	4.1
SSSFF	1286	786.0	1.8	6.2	474.3	1.5	4.9
SSFFF	1462	790.4	1.8	6.1	375.4	1.9	6.1
SFFFF	1639	747.6	1.9	6.5	316.5	2.2	7.3

TABLE II
GRID-GUM1 SPEEDUPS ON HETEROGENEOUS LOW-LATENCY COMPUTATIONAL GRIDS

Edin1 Beowulf cluster, and the fourth column records the relative speedup [1].

In contrast, on heterogeneous computational GRIDs, or those with high communication latency, GRID-GUM1 only delivers acceptable speedups for low-communication degree programs like `queens`, and little speedup for high-communication degree programs like `raytracer`. Table II illustrates the impact of heterogeneity, and shows that adding even a single slow machine to a 5-PE cluster dramatically reduces speedup, e.g. from 4.0 to 2.8 for `queens`. The measurements use the relatively slow SBC (S) and relatively fast Edin3 (F) Beowulf clusters described in Table XIV. The first column shows the GRID configuration e.g. FFSSS is a configuration with two fast machines and three slow machines. The first machine in the configuration string is where the program starts. The second columns shows the mean CPU speed of that configuration. As a measure of heterogeneity, the standard deviations of CPU speeds in all configurations is between 353 and 432 MHz. The third and the sixth columns record the speedup using F's sequential runtime for `raytracer` and `queens` respectively. The fourth and the seventh columns records the speedup using S's sequential runtime, and the fifth and the last columns show the wall-clock execution times.

Tables III and IV show an example of the impact of high communications interconnect. The measurements in both tables are undertaken on the Muni and Edin2 Beowulf clusters described in Table XIV. Each Muni machine is labeled M and each Edin2 machine is labeled E. Table III shows that low communication-degree programs `parFib` and `sumEuler` deliver good speedups on a range of high-latency computational GRIDs. However, Table IV shows that high communication-degree programs like `raytracer`, `matMult` and `linSolv` all deliver poor speedups. The columns in the table are as before, except that the second column reports the mean latency of the GRID configuration. The variation in latency is similar for all configurations, i.e. the standard deviation of the interPE latencies is approximately 17ms.

[1]Absolute speedup is defined with respect to sequential execution and relative speedup is defined with respect to execution of parallel code on a single processing element. Absolute and relative speedups for GUM, together with sequential and parallel efficiency measure are reported in [13].

Config.	Mean Latency (ms)	parFib(45) Rtime Sec.	parFib(45) Speedup E	parFib(45) Speedup M	sumEuler Rtime Sec.	sumEuler Speedup E	sumEuler Speedup M
MMMMM	0.13	205.9	5.0	4.2	523.2	6.1	5.9
EMMMM	14.3	212.7	5.0	4.0	544.0	5.9	5.7
EEMMM	21.5	226.2	4.7	3.8	553.7	5.8	5.6
EEEMM	21.5	224.7	4.7	3.8	620.8	5.1	5.0
EEEEM	14.4	234.0	4.5	3.7	588.4	5.4	5.3
EEEEE	0.15	251.3	4.2	3.4	570.8	5.6	5.4

TABLE III
LOW COMMUNICATION DEGREE PROGRAMS: GRID-GUM1 SPEEDUPS ON HOMOGENEOUS HIGH-LATENCY COMPUTATIONAL GRIDS

Config.	Mean Laten (ms)	raytracer Rtime Sec.	raytracer Spdup E	raytracer Spdup M	matMult Rtime Sec.	matMult Spdup E	matMult Spdup M	linSolv Rtime Sec.	linSolv Spdup E	linSolv Spdup M
MMMMM	0.13	287.8	3.5	3.1	108.6	2.3	2.4	104.4	2.8	2.7
EMMMM	14.3	473.8	2.1	1.9	290.8	0.8	0.9	147.0	2.0	1.9
EEMMM	21.5	413.7	2.4	2.1	228.8	1.1	1.1	142.1	2.1	2.0
EEEMM	21.5	378.7	2.7	2.3	150.9	1.7	1.7	104.9	2.8	2.7
EEEEM	14.4	329.9	3.1	2.7	125.1	2.0	2.1	107.7	2.7	2.7
EEEEE	0.15	279.8	3.6	3.2	95.9	2.7	2.7	102.9	2.9	2.8

TABLE IV
HIGH COMMUNICATION DEGREE PROGRAMS: GRID-GUM1 SPEEDUPS ON HOMOGENEOUS HIGH-LATENCY COMPUTATIONAL GRIDS

V. GRID-GUM2: AN ADAPTIVE RTE FOR COMPUTATIONAL GRIDS

A. GRID-GUM2 Design

To address the shortcomings of GRID-GUM1, we have designed and implemented a revised GPH runtime environment for computational GRIDs, GRID-GUM2. The GRID-GUM2 design is described in full in [8]. In GRID-GUM2 each PE dynamically maintains latency and load information to inform load management, so that work is only sought from PEs which are known to be relatively heavily loaded, and to give preference to local cluster resources. To propagate the necessary information, we augment the messages in GRID-GUM1 to carry dynamic information about latency and load between PEs, and hence between clusters. Such information is combined with static PE characteristics to determine relative loads. To the best of our knowledge, GRID-GUM2 is the first fully implemented virtual shared memory runtime environment on computational GRIDs.

The new load distribution mechanism in GRID-GUM2 has two main components: information collection and adaptive load distribution. The information collection component obtains both static information, like CPU speed of every PE, at program startup, and dynamic information throughout the execution. Example dynamic information is the current load of every PE, and the communication latency from this PE to every other PE. The dynamic information is timestamped and partial, and is cheaply propagated between PEs whenever they communicate.

The adaptive load distribution mechanisms utilise the static and dynamic information, and the following are the key new policies.

- An idle PE only seeks work from (sends a FISH message 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.
- In response to a message seeking work (a FISH message) from a remote, or high communications latency, PE the recipient sends additional work if possible. The intention being to offset the high latency, e.g. between clusters, with bandwidth.
- *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.

In summary, *GRID-GUM*2 incorporates bespoke lightweight mechanisms for reducing communication, and measuring and managing load, rather than using generic GRID services. GRID connective layer services provide communication between, and authentication of, the PEs. *GRID-GUM*2 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 common 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.

VI. *GRID-GUM*2 ON LOW-LATENCY COMPUTATIONAL GRIDS

The following sections evaluate the performance of the new adaptive load distribution mechanism in *GRID-GUM*2 on low-latency heterogeneous, and homogeneous, computational GRIDS.

A. Low-Latency Homogeneous Performance

Section IV-B showed that *GRID-GUM*1 already delivers good performance on low-latency homogeneous computational Grids [8]. Columns 2 and 5 of Table V show that *GRID-GUM*2 maintains this good performance, and sometimes makes a small improvement. The remainder of this section compares the overheads and performance variability of *GRID-GUM*1 and *GRID-GUM*2.

1) Variability: The measurements in Table V have been performed on 10 PEs from the Edin1 cluster. In Table V, the second and fifth columns record the mean of 50 runs in seconds. The third and sixth columns show the variance of the 50 runs. The fourth and seventh columns present the percentage variance relative to the mean. The last column shows the percentage reduction in variance.

Program	*GRID-GUM*1			*GRID-GUM*2			Variance Reduction
	Mean Rtime (s)	Var	Var%	Mean Rtime (s)	Var	Var%	
`queens`	648.97	149.9	23.0%	649.59	2.62	0.4%	98%
`parFib`	84.91	22.68	26.7%	88.85	3.65	4.1%	84%
`linSolv`	176.21	63.82	36.2%	149.82	7.20	4.8%	86%
`sumEuler`	117.82	55.43	47.0%	116.28	20.33	17.4%	63%
`raytracer`	476.93	168.15	35.2%	448.53	27.93	6.2%	82%

TABLE V
*GRID-GUM*1 AND *GRID-GUM*2 PERFORMANCE VARIATION ON 10 PES

Program	RTE	No of Threads	Max Heap Resid. (KB)	Alloc Rate (MB/s)	Comm Degree (Msgs/s)	Aver. Pkt Size (Byte)
`parFib`	**GG**1	26595	5.12	55.3	15.55	5.6
	GG2	26595	5.12	43.2	14.87	5.6
`sumEuler`	**GG**1	82	62.4	52.8	2.09	90.3
	GG2	82	62.4	45.7	0.73	90.3
`raytracer`	**GG**1	350	538.6	60.0	62.72	321.8
	GG2	350	538.6	49.5	46.93	323.0
`linSolv`	**GG**1	242	437.2	40.3	5.50	290.7
	GG2	242	437.2	26.5	2.54	276.4
`matMult`	**GG**1	144	4.3	39.0	67.30	208.9
	GG2	144	4.3	40.0	31.29	209.4
`queens`	**GG**1	24	2.03	38.8	0.26	851.9
	GG2	24	2.03	34.0	0.13	846.2

TABLE VI
*GRID-GUM*1 AND *GRID-GUM*2 OVERHEADS ON 16 PES

Table V shows that *GRID-GUM*1 gives highly variable performance, especially for programs with irregular parallelism. In Table V, the programs with regular parallelism show less variation, e.g. 23% in `queens` and 26.7% in `parFib`. However, the programs with irregular parallelism show greater variation, e.g. 47% in `sumEuler`, 36.2% in `linSolv` and 35.2% in `raytracer`.

The unpredictable behaviour in *GRID-GUM*1 is due to its load distribution mechanism which is based on a naive, random and blind fishing mechanism, as discussed in III-B. Performance is good when idle *GRID-GUM*1 PEs are 'lucky' in their random selection of a PE to request work from. Performance is poor, however, if the idle PEs chose the wrong PE to request work from.

In contrast to *GRID-GUM*1, the adaptive mechanisms in *GRID-GUM*2 result in far less performance variation. In Table V, `queens` and `parFib` show improvements in percentage variance of 98% and 84% respectively. `sumEuler`, `linSolv` and `raytracer`, which have irregular parallelism, show improvements of 63%, 66% and 82% respectively.

2) Overheads: Table VI compares the overheads induced by *GRID-GUM*1 and *GRID-GUM*2 for the six programs. These measurements are made on 16 PEs from Edin1 Beowulf cluster and the runtimes reported are the median of three executions, to ameliorate the impact of operating system and shared network interaction. In the second column **GG**1 and **GG**2 stand for *GRID-GUM*1 and *GRID-GUM*2 respectively. The third column records the total number of threads generated during the execution. The remaining columns show averages over all processors for the maximal heap residency (i.e. the maximum amount of heap that is alive at garbage collection time) the allocation rate (i.e. the amount of local memory allocated per second of execution time) the communication degree (i.e. the number of massages sent per second of execution time) and the average packet size (i.e. the size of packet in Byte).

Table VI shows that, except for communication degree, *GRID-GUM*1 and *GRID-GUM*2 have similar overheads. *GRID-GUM*2 decreases the communication degree by using information about load, latencies and CPU speeds to reduce the number of work-locating FISH messages.

3) Low-latency Homogeneous GRID *Performance Summary:*

- *GRID-GUM*2 maintains this good performance of *GRID-GUM*1 on Low-latency Homogeneous GRIDs, and sometimes makes a small improvement (Columns 2 and 5 of Table V).
- *GRID-GUM*2 programs exhibit far less performance variance than *GRID-GUM*1: reducing variation by at least 63% for all programs measured (Column 8 of Table V).
- *GRID-GUM*2 retains a very light overhead which does not effect the program's dynamic properties (Table VI).

B. Low-Latency Heterogeneous Performance

Table VII reproduces measurements of *GRID-GUM*1 and *GRID-GUM*2 performance on heterogeneous computational GRIDs with moderate communication latency from [8]. The measurements compare runtimes on a small heterogeneous cluster formed from 4 PEs from Edin1 and and 4 PEs from Edin2 Beowulf clusters. The runtimes reported are the median of three executions to ameliorate the impact of operating system and shared network interaction.

Program	Run-time (s)		Improvement
	*GRID-GUM*1	*GRID-GUM*2	
`raytracer`	1340	572	57%
`queens`	668	310	53%
`sumEuler`	570	279	51%
`linSolv`	217	180	17%
`matMult`	94	86	9%
`parFib`	136	134	1%

TABLE VII
*GRID-GUM*1 AND *GRID-GUM*2 PERFORMANCE ON LOW-LATENCY HETEROGENEOUS COMPUTATIONAL GRIDS

Table VII shows that *GRID-GUM*2 outperforms *GRID-GUM*1 on low-latency heterogeneous computational GRIDs. `linSolv` scores a modest improvement under *GRID-GUM*2 of 17%. The limited irregular parallelism and the low-communication degree in `linSolv` helps *GRID-GUM*1 overcome the heterogeneous architecture without an adaptive load distribution mechanism. Due to this, the gains from using the adaptive load distribution of *GRID-GUM*2 to improve `linSolv` are limited.

*GRID-GUM*2 maintains the good parallel performance of `parFib` under *GRID-GUM*1 reported in Table I, but cannot significantly improve it. Likewise, *GRID-GUM*2 cannot significantly improve `matMult` due to inherent limitations on the parallelism [7].

Programs with a low degree of parallelism are most sensitive to a heterogeneous architecture, because an appropriate placement of the small number of threads is essential for good performance. Indeed the low parallelism-degree programs: `raytracer`, `queens` and `sumEuler`, show the greatest improvement under *GRID-GUM*2, each improving by more than 50%.

1) Low-latency Heterogeneous GRID *Performance Summary:* Table VII shows the following points.

- Compared with *GRID-GUM*1, *GRID-GUM*2 improves the performance of 5 of the 6 programs, and maintains the good performance of the 6th (`parFib`).
- Only certain programs are sensitive to low-latency heterogeneous computational GRIDs: some like `parFib` already give good performance, while others like `matMult` are already at some performance bound.
- *GRID-GUM*2 improves the performance of low parallelism-degree programs by more than 50%.

VII. *GRID-GUM*2 ON HIGH-LATENCY COMPUTATIONAL GRIDS

The following sections evaluate the performance of *GRID-GUM*2 on high-latency heterogeneous, and homogeneous, computational GRIDs.

A. High-Latency Homogeneous Performance

Table VIII compares the performance of `raytracer` under *GRID-GUM*1 and *GRID-GUM*2 on all combinations of homogeneous GRIDs with up to 5 PEs. The configurations combine PEs from two very similar clusters with high-latency interconnect, namely the Muni and Edin2 Beowulf clusters described in Tables XIV and XV. Each Edin2 machine is labeled E and each Muni machine is labeled M.

In Table VIII, the first and second columns show case number and GRID configuration. The third column presents the mean communication latency. The fourth and fifth columns record the run-time in seconds for *GRID-GUM*1 (GG1) and *GRID-GUM*2 (GG2) respectively. The last column shows the percentage improvement in *GRID-GUM*2 run-time.

Case	Config.	Mean Latency (ms)	Runtimes		Impr%
			GG1	GG2	
1	1E4M	14.4	995	617	38%
2	1E3M	17.9	1066	728	32%
3	2E3M	21.5	911	703	23%
4	1E2M	23.9	1088	892	18%
5	2E2M	23.9	952	843	11%
6	2E1M	23.9	1007	926	8%
7	1E1M	35.8	1842	1687	8%
8	3E1M	18.0	852	786	8%
9	4E1M	14.4	668	642	4%
10	3E2M	21.5	772	754	2%

TABLE VIII
HOMOGENEOUS HIGH-LATENCY COMPUTATIONAL GRIDS (`raytracer`)

*GRID-GUM*2 improves `raytracer` performance on each of the high-latency homogeneous GRID configurations in Table VIII. For `raytracer`, as for `sumEuler` and `queens`, *GRID-GUM*2 has the greatest improvement against *GRID-GUM*1 on configurations of the form $xEyM$, where $x < y$. This is because in *GRID-GUM*1, the first PE is selected as the main PE, an E PE in this case. In consequence the larger number of remote M PEs must communicate with the main PE through the high-latency interconnect. In contrast, in this configuration *GRID-GUM*2 selects the main PE from the remote group of M PEs, and hence a smaller number of PE(s) require to obtain work through the high-latency interconnect. Moreover when a

FISH is sent over the high-latency interconnect, more work is returned as described in V.

In summary, Table VIII shows that, *GRID-GUM*2 outperforms *GRID-GUM*1 on high-latency homogeneous architecture for `raytracer`, a program with high-communication degree and highly irregular parallelism.

1) Additional High Latency Homogeneous Measurements: We have made similar measurements to those reported above for the `queens` and `sumEuler` programs [42]. *GRID-GUM*2 improves performance on all high-latency homogeneous GRID configurations measured for both programs, with a maximum improvement of 30% for `sumEuler` and a maximum improvement of 9% for `queens`.

2) High-latency Homogeneous GRID *Performance Summary:*

- *GRID-GUM*2 outperforms *GRID-GUM*1 on all of the homogeneous high-latency computational GRID architectures for all three sensitive programs (Table VIII,Section VII-A.1).
- *GRID-GUM*2 improves the performance of programs with a range of parallel behaviours. `raytracer`, with high-communication degree, shows an improvement of up to 37% (Table VIII). `sumEuler`, with low-communication degree and irregular parallelism, shows an improvement of up to 30%. `queens`, with low-communication degree and regular parallelism exhibits least improvement of up to 9% (Section VII-A.1).

B. High-Latency Heterogeneous Performance

High-Latency Heterogeneous computational GRIDs are the most challenging architecture. As the previous section showed that `raytracer`, `queens` and `sumEuler` are the programs that are sensitive to heterogeneity, this section investigates the behaviour of these programs on heterogeneous computational GRIDs.

Table IX compares the performance of `raytracer` under *GRID-GUM*1 and *GRID-GUM*2 on all non-trivial heterogeneous GRIDs with up to 5 PEs. The improvements are analysed to identify the improvements due to the use of static and of dynamic information, using the *GRID-GUM*1.1 experimental run-time environment outlined in Appendix C. The measurements in Table IX are performed on two heterogeneous Beowulf clusters, Edin1 and Muni. PEs in the Edin1 Beowulf cluster have slower CPU speed than those in the Muni Beowulf cluster. Moreover, Edin1 and Muni Beowulf clusters are connected over a high-latency interconnect as detailed in Tables XIV and XV.

In Table IX each Edin1 machine is labeled E and each Muni machine is labeled M. The first and second columns show case number and different combination of PEs from Edin1 and Muni Beowulf clusters respectively. The third and fourth columns report the mean CPU speed and mean latency for the configuration. As before, the variation in latency and CPU speeds is similar for all configurations, with standard deviations of approximately 17ms and 470MHz respectively. The fifth, sixth and seventh columns record the run-time in seconds for *GRID-GUM*1 ($GG1$), *GRID-GUM*1.1 ($GG1.1$) and *GRID-GUM*2 ($GG2$) respectively. The seventh column shows the static information (CPU speed) contribution to the performance change under *GRID-GUM*1.1 in comparison with *GRID-GUM*1. The ninth column indicates the dynamic information (loads and latencies) contribution to the change under *GRID-GUM*2. The last column reports the total performance change using both static and dynamic information in *GRID-GUM*2 in comparison with *GRID-GUM*1.

Case	Config.	Mean Laten.	Mean CPU Spd	GG1 Rtime	GG1.1 Rtime	Static Impr	GG2 Rtime	Dynamic Impr	Total Impr
1	1E4M	14.4	1330.0	1490	689	53%	583	7%	60%
2	1E3M	17.9	1280.5	1658	748	54%	716	2%	56%
3	1E2M	23.9	1197.3	1607	975	39%	848	8%	47%
4	2E3M	21.5	1131.0	1223	745	39%	716	2%	41%
5	2E2M	23.9	1031.5	1396	965	30%	909	4%	34%
6	2E1M	23.9	865.7	1778	1687	5%	1326	20%	25%
7	3E2M	21.5	932.0	1254	983	21%	961	2%	23%
8	1E1M	35.8	1031.5	1934	1678	13%	1689	0%	13%
9	3E1M	18.0	782.7	1495	1832	-22%	1305	34%	12%
10	4E1M	14.4	733.0	1296	1597	-23%	1236	27%	4%

TABLE IX
`raytracer`: HETEROGENEOUS HIGH-LATENCY COMPUTATIONAL GRID

The additional static information enables a substantial improvement when there are more fast PEs (M) than slow PEs (E), i.e. cases 1, 2, 3, 4. For instance, in case 1 *GRID-GUM*1.1 reduces runtime by 54%. However, the improvement due to static information is less when there are more slow PEs than fast PEs, cases (6, 8, 9, 10), and may even degrade performance. For instance, in case 10 *GRID-GUM*1.1 increases the run-time by 23%. This behaviour of `raytracer` under *GRID-GUM*1.1 is related to the high-latency communication. In a configuration of the form ($xEyM$), where $x > y$, cases (6, 8, 9, 10), *GRID-GUM*1.1 nominates the mainPE from M PEs. In this case, E PEs have to seek work during the course of the execution from M PE(s) through high-latency interconnect. Hence for programs with a relatively high-communication degree like `raytracer`, high-latency communication has a major impact on *GRID-GUM*1.1 performance.

The seventh column of Table IX shows that the use of dynamic load and latency information in *GRID-GUM*2 improves performance an all of the GRID configurations. The improvement varies according to the number of remote and local PEs and their CPU speed. If there are fewer slow PEs than fast (($xEyM$), where $x < y$), the dynamic information makes a limited contribution to the performance. For instance, in case 1 the dynamic information improves performance by only 7%. In contrast, if there are more slow PEs than fast (($xEyM$), where $x > y$), the dynamic information has a greater contribution to the performance. For instance, in case 9 the dynamic information improves performance by 34%. In this case the dynamic information is used to nominate the mainPE from among the E PEs, decreasing the number of PEs required to seek work over the high-latency interconnect, and load information is used to transfer larger amounts of work over the high-latency interconnect, thereby reducing the number of messages.

Broadly speaking, both static and dynamic information contribute to the GRID-GUM2 performance gains for a program like `raytracer` with relatively high-communication degree and irregular parallelism. For instance, in case 1, to finish the computation of `raytracer` in five PEs ($1E4M$), GRID-GUM1 requires 1490s. However, GRID-GUM2 requires only 583s, an improvement of 60%.

1) Additional High Latency Heterogeneous Measurements: We have made similar measurements to those reported above for the `queens` and `sumEuler` programs [42]. For `sumEuler` there is a maximum total improvement of GRID-GUM2 over GRID-GUM1 of 32%, and maximum static and dynamic improvements of 27% and 16% respectively. For `queens` there is a maximum total improvement of GRID-GUM2 over GRID-GUM1 of 35%, and maximum static and dynamic improvements of 23% and 12% respectively.

2) High-latency Heterogeneous GRID *Performance Summary:*

- Compared with GRID-GUM1, GRID-GUM2 improves the performance of all three programs on all heterogeneous high-latency GRID configurations measured (Column 10 of Table IX, Section VII-B.1).
- GRID-GUM2's static information gives substantial improvements when there are more fast PEs than slow PEs, but less when there are more slow PEs than fast PEs (Column 7 of Table IX).
- GRID-GUM2's dynamic load and latency information improves performance on all of the heterogeneous high-latency GRID configurations measured. The improvement is greater if there are more slow PEs than fast, and less if there are more fast machines than slow (Column 7 of Table IX).
- For a program with a high-communication degree, `raytracer`, GRID-GUM2 delivers a substantial maximum improvement of 60%, whereas for both programs with relatively low-communication degree (`sumEuler` and `queens`) more modest improvements of 31% and 35% (Section VII-B.1).

VIII. SCALABILITY

This subsection investigates the performance scalability 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 XIV and XV. Because of the relative cluster sizes, many configurations have 6 Edin1 PEs for every Muni PE.

The experiments have the following limitations:

- The programs in Table XVI were designed for smaller scale high performance computers, and only some of them generate sufficient parallelism to utilise medium-scale, and large-scale, computational GRIDs.
- The number of PEs available for these experiments at the cooperating sites were limited: Edin1 (E) 30 PEs, Edin2 (E_2) 5 PEs, and Muni (M) 6 PEs.

A. `raytracer`

The `raytracer` is a realistic parallel program with limited amounts of highly-irregular parallelism and a relatively high communication degree (Table XVI). Table X 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.

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 X, the `parFib` from the $6E1M$ GRID-GUM2 row of Table XII.

Table XI 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. Section IV suggests that better speedups and efficiency would be obtained on either an homogeneous GRID, or a low latency GRID. Moreover Table XII reports rather better efficiency for a less challenging program.

B. `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 XVI). Table XII 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

Case	GUM			GRID-GUM1		
	Config.	Rtime	Spdup	Config.	Rtime	Spdup
1	7E	2609	7	6E1M	2530	7
2	14E	2168	8	12E2M	2185	8
3	21E	1860	10	18E3M	1824	10
4	28E	1771	10	24E4M	1776	10
5	30E	1762	10			
6				30E5M	1666	11
7				$5E_2$30E6M	1652	11

TABLE X

GUM AND GRID-GUM1 SCALABILITY (raytracer)

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 XI

GRID-GUM1 AND GRID-GUM2 SCALABILITY (raytracer)

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 IV 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 XII demonstrates that sophisticated load distribution is not required for parFib. That the GRID-GUM2 performance is so similar to the GRID-GUM1 performance shows that even on medium-scale computational GRIDs, the overheads of GRID-GUM2's load distribution mechanism remain minimal.

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 XII

GRID-GUM1 AND GRID-GUM2 SCALABILITY (parFib)

C. Scalability Summary

- The experiments in this subsection show that emerging GRID technology offers the opportunity to improve performance by integrating remote heterogeneous clusters into a computational GRID (Table X).
- The measurements in Tables XI and XII show that the parallel performance of GRID-GUM2 scales to medium scale heterogeneous high-latency computational GRIDs: 41 PEs in three clusters, and (Table XI) continues to deliver significant performance benefits over GRID-GUM1 for a realistic program.
- The measurements of parFib, a program with near-ideal parallel behaviour, show show that the overheads of GRID-GUM2 load management are relatively low, even on medium scale computational GRIDs (Table XII).

Program	Regul	Comm	GRID Config.		GG1	GG1.1	GG2	Case
			Lat.	Archit.				
raytracer	Irreg	High	High	Hetr	1	2	3	1
			High	Hom	1	1	2	2
			Low	Hetr	1	2	3	3
			Low	Hom	1	1	2	4
sumEuler	Irreg	Low	High	Hetr	1	2	3	5
			High	Hom	1	1	2	6
			Low	Hetr	1	2	3	7
			Low	Hom	1	1	2	8
linSolv	Irreg	Low	High	Hetr	1	2	3	9
			High	Hom	1	1	2	10
			Low	Hetr	1	2	3	11
			Low	Hom	1	1	2	12
matMult	Reg	High	High	Hetr	1	2	2	13
			High	Hom	1	1	2	14
			Low	Hetr	1	2	2	15
			Low	Hom	1	1	1	16
queens	Reg	Low	High	Hetr	1	2	2	17
			High	Hom	1	1	1	18
			Low	Hetr	1	2	2	19
			Low	Hom	1	1	1	20
parFib	Reg	Low	High	Hetr	1	2	2	21
			High	Hom	1	1	1	22
			Low	Hetr	1	2	2	23
			Low	Hom	1	1	1	24

TABLE XIII

COMPARATIVE PERFORMANCE SUMMARY: GRID-GUM1, GRID-GUM1.1 AND GRID-GUM2

IX. GRID-GUM2 PERFORMANCE ANALYSIS

This section analyses the performance of the benchmark programs under GRID-GUM1, GRID-GUM1.1 and GRID-GUM2 on combinations of high/low and homo/hetero-geneous computational GRIDs with respect to their communications behaviour and degree of irregular parallelism. In table XIII, the second and third columns present the program characteristics, parallelism regularity and communication degree, respectively. The fourth and fifth columns give the GRID latency and homo/hetero-geneity. The sixth, seventh and eighth columns rank the performance of GRID-GUM1 ($GG1$), GRID-GUM1.1 ($GG1.1$) and GRID-GUM2 ($GG2$) respectively from 3 (best) to 1 (worst). The last column presents the case number.

We make the following conclusions from Table XIII.

- GRID-GUM2's dynamic adaptive load management techniques are effective: they improve or maintain the performance of all the benchmark programs on all GRID configurations (Column 8).
- Sophisticated load management is not required to effectively parallelise regularly-parallel programs on homogeneous computational GRIDs. That is, GRID-GUM2 does

not reliably improve the performance of these programs (Cases 14,16,18,20,22 and 24).

- Static information is the key to effectively parallelising regularly-parallel programs on heterogeneous computational GRIDs: *GRID-GUM*2 shows the same improvement as *GRID-GUM*1.1 for these programs (Cases 13,15,17,19,21 and 23).

In summary, not only does the adaptive load distribution of *GRID-GUM*2 deliver more predictable performance than *GRID-GUM*1 as shown in Table V, but it also reduces the runtime of all programs.

X. CONCLUSION

A. Summary

We have presented a systematic evaluation of the performance of GPH, the first DSM language with high-level parallel coordination on computational GRIDs. We report both absolute performance and performance relative to *GRID-GUM*1 and GUM, and the latter has previously been compared with conventional parallel technology (C with PVM). In essence we have demonstrated that a high-level DSM parallel programming paradigm can deliver good parallel performance for a variety of applications on a range of high/low latency and homo/hetero-geneous computational GRIDs. Moreover, the performance scales to medium-scale computational GRIDs. The core of our approach to achieving good performance from this class of parallel language is a sophisticated runtime environment with aggressive and dynamic load management mechanism.

We have summarised earlier work outlining *GRID-GUM*1, a port of the GUM runtime environment for GPH, originally designed for a single high performance computer, to computational GRIDs; showing that *GRID-GUM*1 only reliably delivered good performance on low-latency homogeneous computational GRIDs; that poor load management limits *GRID-GUM*1 performance; and outlining the design of *GRID-GUM*2, a new runtime environment incorporating new adaptive load management techniques.

The evaluation of *GRID-GUM*2 performance covers combinations of high/low latency, and homo/hetero-geneous computational GRIDs, with the results outlined in the paragraphs below. Unsurprisingly *GRID-GUM*2 gives greatest performance improvements on the most challenging combination: a 60% improvement on a heterogeneous high latency computational GRID (Table IX).

On low latency homogeneous computational GRIDs, Table V shows how *GRID-GUM*2 maintains the good performance of *GRID-GUM*1 reported in Table I (Section VI-A). On low latency heterogeneous computational GRIDs *GRID-GUM*2 improves the performance of 5 out of 6 programs and maintains the good performance of the 6th, although only certain programs are sensitive to heterogeneity (Section VI-B). On high latency homogeneous computational GRIDs *GRID-GUM*2 improves the performance of all three programs on all GRID configurations measured (Section VII-A). On high latency heterogeneous computational GRIDs *GRID-GUM*2 improves the performance of all three sensitive programs on all GRID configurations measured (Section VII-B).

The scalability measurements consider the most challenging, but most common, computational GRIDs: heterogeneous high-latency GRIDs. The results show that *GRID-GUM*2 performance scales to medium scale heterogeneous high-latency computational GRIDs, e.g. delivering a speedup of 28 on 41 PEs in three clusters, although effiency falls to just 65% on this challenging architecture (Section VIII). The relative performance of the programs on all combinations of low/high latency and homo/heterogeneous computational GRIDs has been analysed with respect to their communications behaviour and degree of irregular parallelism. The analysis shows that *GRID-GUM*2's dynamic adaptive load management techniques are effective as they improve or maintain the performance of all the benchmark programs on all GRID configurations (Section IX).

B. Limitations and Future Work

The current work has the following limitations. The parallel programs measured are small and medium-scale kernels. The scalability of *GRID-GUM*2 has only been measured on medium-scale computational GRIDs. *GRID-GUM*2 inherits limited and user-authentication biased security mechanisms from Globus Toolkit. *GRID-GUM*2 inherits a restriction to closed systems, i.e. executing on a fixed set of PEs, from the MPICH-G2 communications library. Currently *GRID-GUM*2 has no fault tolerance mechanisms: if any PE or communication link fails then the entire computation may fail.

There are several avenues to extend this research and address the limitations. One avenue is to implement larger parallel programs, and our current work entails parallelising large computer algebra computations as part of the *SCIEnce* project (Symbolic Computation Infrastructure for Europe EU FP VI I3-026133). A second research avenue is to investigate the scalability of *GRID-GUM*2 on large-scale 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.

Another future research avenue is to implement a program-based security mechanism to analyse program behaviour to decide whether to permit execution of the code. For example, to enhance program based security, certificates of bounded resource consumption could be attached to the code sent between PEs in the network and checked by a resource protection component before executing the code. Using a communication library other than MPICH-G2 would enable *GRID-GUM*2 to support open systems, and possibilities include using an optimised version of PVM for computational GRIDs, e.g. [43].

A rather more challenging task would be to tackle the problem of fault tolerant parallel execution on computational GRIDs. Here *GRID-GUM*2 could benefit from the statelessness of functional programs. Statefulness amounts to updating the global program state and its absence means that the damage caused by a failing computation is confined. Moreover, if an error is detected, pure computations can be automatically restarted without the danger of making multiple updates. A second potential benefit of high-level language technology is that fault tolerance is a global property affecting all operations of the virtual machine underlying a language, and enforcing such property is easier with a high level virtual machine like GRID-GUM2. Indeed, runtime environment level fault tolerance has been proposed for GUM [44].

REFERENCES

[1] I. Foster and C. Kesselman, "Computational Grids," *The Grid: Blueprint for a Future Computing Infrastructure*, 1998.

[2] J. Basney and M. Livny, *High Performance Cluster Computing*. Prentice-Hall, 1999, vol. 1, ch. Deploying a High Throughput Computing Cluster.

[3] S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *Software - Practise and Experience*, vol. 23, no. 12, pp. 1305–1336, 1993.

[4] MPI-Forum, "MPI: A message passing intrface standard," *International Journal of Supercomputer Application*, vol. 8, no. 3–4, pp. 165–414, 1994.

[5] 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.

[6] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones, "Algorithm + Strategy = Parallelism," *J. of Functional Programming*, vol. 8, no. 1, pp. 23–60, January 1998. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/strategies.ps.gz

[7] H.-W. Loidl, F. Rubio Diez, N. Scaife, K. Hammond, U. Klusik, R. Loogen, G. Michaelson, S. Horiguchi, R. Pena Mari, S. Priebe, A. Rebon Portillo, and P. Trinder, "Comparing Parallel Functional Languages: Programming and Performance," *Higher-order and Symbolic Computation*, vol. 16, no. 3, pp. 203–251, 2003.

[8] 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*, vol. 6, no. 4, 2006.

[9] R. Loogen, "Programming Language Constructs," in *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds. Springer-Verlag, 1999, pp. 63–91.

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

[11] D. B. Loveman, "High performance fortran," *IEEE Parallel Distrib. Technol.*, vol. 1, no. 1, pp. 25–42, 1993.

[12] G. Michaelson, N. Scaife, P. Bristow, and P. King, "Nested Algorithmic Skeletons from Higher Order Functions," *Parallel Algorithms and Applications*, vol. 16, pp. 181–206, 2001.

[13] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones, "GUM: a Portable Parallel Implementation of Haskell," in *PLDI'96 — Programming Languages Design and Implementation*, Philadelphia, PA, USA, May 1996, pp. 79–88. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/gum.ps.gz

[14] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí, "Eden — The Paradise of Functional Concurrent Programming," in *EuroPar'96 — European Conf. on Parallel Processing*, ser. LNCS 1123. Lyon, France: Springer, 1996, pp. 710–713.

[15] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann, 1999.

[16] Globus, 2005, <URL:`http://www.globus.org/toolkit/`>.

[17] A. Grimshaw and W. Wulf, "The Legion Vision of a World-Wide Virtual Computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, 1997.

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

[19] D. Jackson, "Advanced Scheduling of Linux Clusters using Maui," in *USENIX'99*, 1999.

[20] E. Smirni and E. Rosti, "Modelling Speedup of SPMD Applications on the Intel Paragon: A Case Study," in *HPCN'95 High Performance Computing and Networks, Languages and Computer Architecture*, Milan, Italy, 1995.

[21] L. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–, Aug. 1990.

[22] M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, M. Kohl, J.a nd Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam, "HARNESS: A Next Generation Distributed Virtual Machine," *Future Generation Computer Systems*, vol. 15, no. 5/6, pp. 571–582, Oct. 1999, special Issue on Metacomputing.

[23] 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*, vol. 2536 of LNCS. Springer-Verlag, 2001.

[24] C. Baker-Finch, D. King, J. Hall, and P. Trinder, "An Operational Semantics for Parallel Lazy Evaluation," in *ICFP'00 — International Conference on Functional Programming*. Montreal, Canada: ACM Press, Sept. 2000, pp. 162–173.

[25] T. Murphy VII, K. Crary, and R. Harper, "Distributed Control Flow with Classical Modal Logic," in *Proceedings of 19th International Workshop on Computer Science Logic (CSL 2005)*, ser. LNCS 3634. Springer, July 2005, pp. 51–69.

[26] R. Whaley, A. Petitet, and J. Dongarra, "Automated Empirical Optimisations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, pp. 3–35, 2001.

[27] "Distributed Shared Memory Home Pages," WWW page, 2006, `http://www.ics.uci.edu/ javid/dsm.html/`.

[28] C. Morin, P. Gallard, R. Lottiaux, and G. Valle, "Design and implementations of ninf: towards a global computing infrastructure," *Future Gener. Comput. Syst.*, vol. 20, no. 2, 2004.

[29] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, pp. 1512–1530, 2000.

[30] T.-Y. Liang, C.-Y. Wu, J.-B. Chang, and C.-K. Shieh, "Teamster-G: a grid-enabled software DSM system," in *CCGRID 2005*, 2005, pp. 905–912.

[31] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "ASSIST as a research framework for high-performance Grid programming environments," in *Grid Computing: Software environments and Tools*, J. C. Cunha and O. F. Rana, Eds. Springer, Jan. 2006.

[32] 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. W. R. Torczon, "The GrADS Project: Software Support for High-Level Grid Application Development," *Int. Journal of High Performance Computing Applications*, vol. 15, no. 4, pp. 327–344, 2001.

[33] 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.

[34] M. Aldinucci and M. Danelutto, "Advanced skeleton programming systems," *Parallel Computing*, 2006, to appear. [Online]. Available: http://www.di.unipi.it/ aldinuc/papers.html

[35] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, 2004.

[36] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a flexible and efficient Java based grid programming environment," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 7-8, pp. 1079–1107, June 2005.

[37] J. Dunnweber, M. Alt, and S. Gorlatch, "Apis for grid programming using higher order components," in *GGF12 - The Twelfth Global Grid Forum, Brussels, Belgium*, T. Kielmann, S. Pickles, and S. J. Cox, Eds., September 2004. [Online]. Available: http://pvs.uni-muenster.de/pvs/mitarbeiter/jan/adgggf04.html

[38] M. Alt and S. Gorlatch, "Adapting java rmi for grid computing," *Future Generation Computer Systems*, vol. 21, no. 5, pp. 699–707, 2005. [Online]. Available: http://pvs.uni-muenster.de/pvs/publikationen/

[39] 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*, vol. 11, pp. 701–752, 1999. [Online]. Available: http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/cpe-gph.ps.gz

[40] K. Hammond, J. Mattson Jr., A. Partridge, S. Peyton Jones, and P. Trinder, "GUM: a Portable Parallel Implementation of Haskell," in *IFL'95 — Intl Workshop on the Parallel Implementation of Functional Languages*, Sept. 1995. [Online]. Available: ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Simon_Peyton_Jones/gum-ifl95.ps.gz

[41] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: a grid-enabled implementation of the message passing interface," *Journal Parallel Distributed Computing*, vol. 63, no. 5, pp. 551–563, 2003.

[42] A. Al Zain, "Implementing High-Level Parallelism on Computational GRIDs," Ph.D. dissertation, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland, UK, April 2006. [Online]. Available: http://www.macs.hw.ac.uk/ trinder/theses/AlZainAbstract.html

[43] G. Sipos and P. Kacsuk, "Executing and Monitoring PVM Programs in Computational Grids with Jini." in *PVM/MPI*, ser. Lecture Notes in Computer Science, J. Dongarra, D. Laforenza, and S. Orlando, Eds., vol. 2840. Springer, 2003, pp. 570–576, http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2840&spage=570.

[44] P. Trinder, R. Pointon, and H.-W. Loidl, "Towards Runtime System Level Fault Tolerance for a Distributed Functional Language," in *SFP'00 — Scottish Functional Programming Workshop*, ser. Trends in Functional Programming, vol. 2. St Andrews, Scotland, Jul 26–28: Intellect, 2000, pp. 103–113.

APPENDIX

A. Hardware Apparatus

Beowulfs	CPU Speed MHz	Cache kB	Memory Total kB	PEs
Edin1	534	128	254856	32
Edin2	1395	256	191164	6
Edin3	1816	512	247816	10
Muni	1529	256	515500	7
SBC	933	256	110292	4

TABLE XIV
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
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

TABLE XV
APPROXIMATE INTER-CLUSTER LATENCIES (MS)

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 Borders campus(*SBC*); see Tables XIV and XV for the characteristic of these Beowulfs.

B. Software Apparatus

Table XVI summarises the characteristics of the six programs measured. `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 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.

Three of the programs have regular parallelism `queens`, `parFib` and `matMult`; three programs have irregular parallelism `sumEuler`, `linSolv` and `raytracer`. Programs with regular parallelism generate threads which have approximately the same cost of computation. Programs with irregular parallelism generate threads with varying cost of computation. Moreover, irregular-parallel programs generate threads at different stages through the course of execution. Of the programs, `queens`, `sumEuler` and `linSolv` have relatively low-communication degrees, i.e. perform relatively little communication per unit execution time, whereas `parFib`, `matMult` and `raytracer` have relatively high-communication degree, as shown in column 6 of Table VI.

Program	Application Area	Paradigm	Regularity	Comm Degree PKT/S	Source Lines Code (SLOC)
`queens`	AI	Div-. Conq.	Regul	0.2 low	21
`parFib`	Numeric	Div-Conq.	Regul	65.5 high	22
`linSolv`	Symbolic Algebra	Data Par.	Limit Irreg.	5.5 low	121
`sumEuler`	Numerical Analysis	Data Par.	Irreg.	2.09 low	31
`matMult`	Numeric	Div-Conq.	Irreg.	67.3 high	43
`raytracer`	Vision	Data Par.	High irreg.	46.7 high	80

TABLE XVI
PROGRAM CHARACTERISTICS

C. GRID-GUM1.1

A special implementation of GRID-GUM2, GRID-GUM1.1, is used to study the performance impact of the static information, namely the CPU speed of every PE in the GRID. GRID-GUM1.1 uses CPU speed information to choose a fast PE as the mainPE where the program starts, and to prevent slow PEs from extracting work from faster PEs unless the latter is the mainPE. Unlike GRID-GUM2, GRID-GUM1.1 does not collect or use dynamic information on PE loads and latencies.

D. GPH *Example:* `sumEuler`

As a non-trivial example of the GPH language, the complete code for the `sumEuler` program outlined in Table XVI is given below. The only evaluation strategy required to parallelise the program is in the last line of the `sumTotient` function.

```
-------------------------------------------------
-- This program calculates the sum of Euler
-- totients between a lower and an  upper limit,
-- using fixed precision integers.
-------------------------------------------------
module Main(main) where

import System(getArgs)
import Strategies
-------------------------------------------------
-- Primary Function, sumTotient
-------------------------------------------------
sumTotient :: Int -> Int -> Int -> Int
sumTotient lower upper c =
  sum ( map (sum . map euler)
       (splitAtN c [upper, upper-1 .. lower])
       `using` parList rnf)

-------------------------------------------------
-- euler
```

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

-------------------------------------------------
-- Auxiliary Functions
-------------------------------------------------
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

--------------------------------------------------
-- Interface Section
--------------------------------------------------
main = do args <- getArgs
          let
            lower = read (args!!0) :: Int
            upper = read (args!!1) :: Int
            c     = read (args!!2) :: Int
          putStrLn ("Sum of Totients between [" ++
                    (show lower) ++ ".." ++
                    (show upper) ++ "] is " ++
                     show (sumTotient
                           lower upper c))
```