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

Abdallah D. Al Zain, Phil W. Trinder, Greg J. Michaelson, and Hans-Wolfgang 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 (DSM) model 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 $GRID\text{-}GUM2$, 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 $GRID\text{-}GUM2$ on combinations of high/low and homogeneous/heterogeneous 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 $GRID\text{-}GUM2$'s performance scalability on medium-scale heterogeneous and high-latency computational GRIDs and analyze 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.

1 Introduction

Hardware price and performance ratios make cluster computing increasingly attractive. Moreover, emerging GRID technology [1] offers the potential of connecting 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, for example, under Condor [2] or the 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 synchronize.

Computational GRIDs are much harder to utilize effectively for parallelism than a classical high-performance computer (HPC). A classical HPC typically comprises a large number of homogeneous processing elements (PEs), communicating by 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, which is 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 levels of performance. Moreover, the interconnect is 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, for example, 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, for example, algorithmic skeletons [5], as detailed in Section 2.

We advocate specifying parallelism on computational GRIDs in a language with high-level coordination and a Distributed Shared Memory (DSM) model. 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 (RTE). The language investigated here is Glasgow parallel Haskell (GPH) [6], and its GUM RTE has been engineered to deliver good performance on classical HPCs and clusters [7]. We

- *A.D. Al Zain, P.W. Trinder, and G.J. Michaelson are with the School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland. E-mail: {ceeatia, trinder, greg}@macs.hw.ac.uk.*
- *H.-W. Loidl is with the Institut fur Informatik, Ludwig-Maximilians-Universitat Munchen, Germany. E-mail: hwloidl@tcs.ifi.lmu.de.*

Manuscript received 8 Sept. 2006; revised 9 Mar. 2007; accepted 15 May 2007; published online 28 June 2007.
Recommended for acceptance by G. Karypis.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0279-0906.
Digital Object Identifier no. 10.1109/TPDS.2007.70728.

have previously shown that a direct port of GUM to a the GRID, *GRID-GUM1*, only reliably provides good performance for low-latency homogeneous GRIDs and that load management limits the performance [8]. To overcome the limitations of *GRID-GUM1*, we have designed and implemented *GRID-GUM2* 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 homogeneous/heterogeneous computational GRIDs. That is, we present a systematic evaluation of the *GRID-GUM2* implementation of GPH, which is the first virtual shared-memory parallel language for computational GRIDs. The investigation uses six kernel parallel programs from a range of application areas, for example, 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 this paper is structured as follows: Section 2 describes related work. Section 3 describes the GPH language and its GUM RTE designed for a single HPC or cluster. Section 4 summarizes the design and performance of an initial port of GUM to the GRID *GRID-GUM1*. Section 5 outlines the design of *GRID-GUM2* with new load management mechanisms. Section 6 evaluates the performance of *GRID-GUM2* on a low-latency heterogeneous and homogeneous computational GRIDs. Section 7 evaluates the performance of *GRID-GUM2* on high-latency heterogeneous and homogeneous computational GRIDs. Section 8 investigates the performance scalability of the *GRID-GUM2* load distribution mechanisms on high-latency heterogeneous computational GRIDs. Section 9 analyzes the relative performance of all programs under *GRID-GUM1* and *GRID-GUM2* on combinations of low-latency/high-latency and homogeneous/heterogeneous computational GRIDs with respect to their communication behavior and degree of irregular parallelism. Section 10 concludes.

2 Related Work

2.1 High-Level Parallel Coordination

A parallel program must specify both *computation*, that is, a correct and efficient algorithm, and *coordination*, that is, how the computations across the PEs can be organized. Coordination typically includes aspects such as thread creation, placement, and synchronization. The computation aspect of a parallel program may be specified at a range of levels of abstraction, for example, relatively low level like assembler or C, or at a high level like SML or Haskell 98.

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 characterization of coordination abstraction levels from [9]. In a language with *explicit parallelism*, a programmer may explicitly create and place each thread and communicate and synchronize between threads. For example, the MPI [4] and PVM [10] libraries support coordination at this level. In languages with *semiexplicit parallelism* like GPH or Eden, the programmer specifies only a few key coordination aspects, for example, 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 does not specify coordination aspects, as the parallelism is implicit in the language semantics.

The great advantage of high-level, that is semiexplicit or implicit, parallel coordination is that it frees the programmer from specifying low-level coordination details. The disadvantages are that an 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 RTE 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 semiexplicit and implicit parallel languages than well-engineered implementations.

2.2 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 organizations [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 realizations 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 GRIDs*, which aggregate substantial computational resources to tackle problems that cannot be solved on a single system. A computational GRID typically comprises a number of HPCs, 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 a *hierarchy* of communication latencies, with communication to PEs at remote clusters being the slowest, to PEs at nearby clusters being slow, and to PEs within the same cluster being the fastest. Moreover, as the wide area network is shared, the 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], and 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 that we address is effectively executing 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.

The dominant parallelism paradigm for classical HPCs is 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 initialized 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.

2.3 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. Heterogeneous Adaptable Reconfigurable Networked Systems [22] (Harness) 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 personalized 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 customized 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 synchronize tasks. This reflects ConCert's distributed memory model implemented by mobile code units (chords). The DSM parallel graph rewriting enables a relatively simple denotational and operational semantics for GPH [24], whereas ConCert uses a modal lambda calculus [25].

To achieve good parallel performance on a variety of different machines, the Automated Empirical Optimization of Software (AEOS) paradigm 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 should be chosen, for example, to select cutoff values in recursive functions, depending on the processor speed. However, in the AEOS paradigm, the adoption of the software has to be done by a program and 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 parallel computing, as the architecture is too complex and dynamic for programmers to readily manage.

2.4 Distributed Shared Memory

One means of providing high-level coordination is to abstract over the memory architecture of a distributed system, that is, 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 the 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, that is, location, page, or object.

The key issue with DSM systems is efficiently maintaining a coherent view of the "shared" memory in the presence of concurrent updates on multiple PEs. A coherence protocol, which is chosen in accordance with some consistency models, 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 or 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, that is, 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, so far, only measured on small-scale GRIDs [30]. In contrast, GPH has the potential of utilizing large-scale computational GRIDs, and we report measurements on medium-scale GRIDs in Section 8.

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 RTE 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, for example, blocking any thread that demands the value of a graph node that is currently under evaluation.

Our GPH language has the potential of utilizing large-scale computational GRIDs, and we report measurements on medium-scale GRIDs in Section 8.

2.5 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 is 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, and 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 and the coordination in a single high-level language 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 a parallel GRID computation. Parallel applications are constructed by parameterizing a suitable skeleton with sequential functional units. Examples of this approach include work groups lead by Aldinucci and Danelutto [33], [34], Cole [35], and Alt et al. [5]. In contrast to the fixed set of skeletons, it is possible to define new coordination constructs in GPH, as outlined in Section 3.1.

Perhaps, the approach that is most closely related to ours is to port a high-level distributed programming language to the GRID. Both van Nieuwpoort et al [36] and Gorlatch et al. [37], [38] port Java to the GRID and use Remote Method Invocation (RMI) as the programming abstraction. Coordination in GPH is on a higher level than in RMI and is 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 by using high-level evaluation strategies, as outlined in Section 3.1. The remaining coordination aspects are dynamically managed by a sophisticated RTE *GRID-GUM2* specifically designed for computational GRIDs. GPH provides higher level coordination than the other GRID parallel programming languages described in the previous section.

3 GPH AND GUM

3.1 Glasgow Parallel Haskell (GPH)

GPH is a semiexplicit parallel functional language, enabling the programmer to specify parallelism with relatively little effort by using high-level parallel coordination constructs. It is a modest and conservative extension of Haskell 98, which is a nonstrict purely functional programming language [6]. GPH extends Haskell 98 with a parallel composition `par`, and an expression `e1 'par' e2 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 the evaluation of `e2`. Results from `e1`'s evaluation are available in `e2`, which shares subgraphs evaluated in `e1`, for example through common variables. GPH programs also sequence the evaluation of expressions by using the `seq` sequential composition. For example, a parallel naive `nfib` function, which is 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*, that is, 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 following `parMap` parallel map function 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 by using the `rnf` strategy. `parList` and `rnf` have 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 the 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 A.4 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 structure the `sumTotient` function appropriately and add the architecture-neutral evaluation strategy at the last line of the function. A thorough account of how one can engineer efficient parallel programs in GPH is given in [39]. The cost of providing the programmer with such a high-level abstraction is that GPH requires an elaborate RTE to dynamically manage parallel execution on complex architectures, and these are described next.

3.2 GUM: A Parallel Haskell Runtime Environment

GUM is a portable parallel 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 RTE. Here, we describe the key components for a GRID context, namely, program initialization and load distribution, for GUM 4.06 using the PVM communications library [10]. A full description of GUM is available in [13].

3.3 GUM Program Initialization

When a GPH program is launched under GUM, it initially creates a PVM manager task, whose job is to control the start-up 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 initializes itself: processing runtime arguments, allocating heap, etc. Once all PE tasks have initialized and been

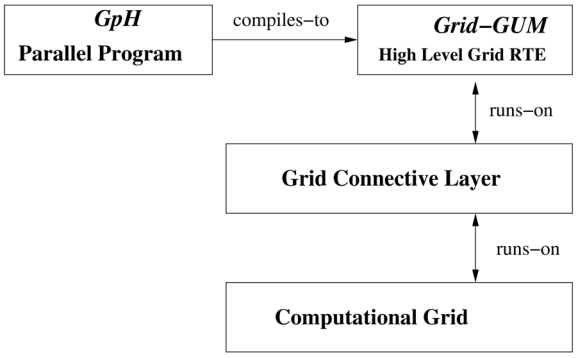

Fig. 1. *GRID-GUM1* system architecture.

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.

3.4 GUM Thread Management

The unit of computation in GUM is a lightweight thread, and each logical PE is an operating system process that coschedules multiple lightweight threads, as outlined below and detailed in [13]. Threads are automatically synchronized 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 reducible expression or *thunk* is a 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.

3.5 GUM Load Distribution

GUM uses dynamic, decentralized, and blind load management. The load distribution mechanism is designed for a flat architecture with uniform PE speed and communication latency and works as follows: 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 that was *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 that is again chosen at random. If a FISH recipient has a useful spark, it sends a SCHEDULE message to the PE that originated the FISH, containing the sparked thunk packaged with a 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.

3.6 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-memory 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, Loidl et al. [7] compare 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 Multiprecision library and the GNU C compiler. The sequential C program is five 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. However, the sizes of the GPH and C+PVM programs differ substantially: the C+PVM program is six times longer than the GPH program.

TABLE 1
GRID-GUM1 Speedups on a 16-PE Homogeneous Low-Latency Computational Grid

program	Runtime		Speedup
	Seq sec	16 PE sec	
`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

4 *GRID-GUM1*

4.1 *GRID-GUM1* Architecture

GRID-GUM1 is a port of GUM to the GRID [8]. The key part of the port is to utilize the MPICH-G2 communication library [40] in the GUM communication layer. MPICH-G2, in turn, uses the Globus Toolkit middleware, as illustrated in Fig. 1.

4.2 *GRID-GUM1* Performance

The following section summarizes the *GRID-GUM1* results from [8]. It reports measurements of the suite of programs characterized in Appendix A.2, where a key characteristic of the programs is the *communication degree*, that is, the number of messages transmitted per unit of execution time. The programs are measured on the collection of GRID-enabled Beowulf clusters specified in Appendix A.1.

Table 1 shows that for programs with a sufficiently large execution time, *GRID-GUM1* can deliver good speedups on homogeneous computational GRIDs with relatively low communication latency. The measurements are performed on the 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

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

TABLE 2
GRID-GUM1 Speedups on Heterogeneous Low-Latency Computational Grids

Config.	Mean CPU (MHz)	raytracer			queens(13)		
		Rtime	Speedup		Rtime	Speedup	
		Sec.	F	S	Sec.	F	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

degree programs like `queens`, and little speedup for high-communication degree programs like `raytracer`. Table 2 illustrates the impact of heterogeneity and shows that adding even a single slow machine to a five-PE cluster dramatically reduces the speedup, for example, from 4.0 to 2.8 for `queens`. The measurements use the relatively slow SBC S and relatively fast Edin3 F Beowulf clusters, as described in Table 14. The first column shows the GRID configuration. For example, 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 column 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 by using F's sequential runtime for `raytracer` and `queens`, respectively. The fourth and seventh columns record the speedup by using S's sequential runtime, and the fifth and last columns show the wall clock execution times.

Tables 3 and 4 show an example of the impact of a high communication latency interconnect. The measurements in both latency tables are undertaken on the Muni and Edin2 Beowulf clusters described in Table 14. Each Muni machine is labeled M, and each Edin2 machine is labeled E. Table 3 shows that low communication-degree programs `parFib` and `sumEuler` deliver good speedups on a range of high-latency computational GRIDs. However, Table 4 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; that is, the standard deviation of the inter-PE latencies is approximately 17 ms.

TABLE 3
Low Communication-Degree Programs: *GRID-GUM1* Speedups on Homogeneous High-Latency Computational Grids

Config.	Mean Latency (ms)	parFib(45)			sumEuler		
		Rtime	Speedup		Rtime	Speedup	
		Sec.	E	M	Sec.	E	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 4
High Communication-Degree Programs: *GRID-GUM1* Speedups on Homogeneous High-Latency Computational Grids

Config.	Mean Laten (ms)	raytracer			matMult			linSolv		
		Rtime	Spdup		Rtime	Spdup		Rtime	Spdup	
		Sec.	E	M	Sec.	E	M	Sec.	E	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

5 *GRID-GUM2*: An Adaptive RTE for Computational Grids

5.1 *GRID-GUM2* Design

To address the shortcomings of *GRID-GUM1*, we have designed and implemented a revised GPH RTE 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 that 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 RTE 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 the CPU speed of every PE, at program start-up and dynamic information throughout the execution. An example of dynamic information is the current load of every PE and the communication latency from this PE to every other PE. The dynamic information is time stamped and partial, and is cheaply propagated between PEs whenever they communicate.

The adaptive load distribution mechanisms utilize 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-communication-latency PE, the recipient sends additional work if possible, with the intention of offsetting the high

TABLE 5
*GRID-GUM*1 and *GRID-GUM*2 Performance Variation on 10 PEs

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%

latency, for example, between clusters, with bandwidth.

- *GRID-GUM*2 starts the computation in the "biggest" cluster, that is, 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, as well as 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; that is, 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, that is, to be most effective on 1) dedicated computational GRIDs where only one program is executed at a time and 2) a nonpreemptive environment, where each program executes to completion without interruption.

6 *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.

6.1 Low-Latency Homogeneous Performance

Section 4.2 showed that *GRID-GUM*1 already delivers good performance on low-latency homogeneous computational GRIDs [8]. Columns 2 and 5 in Table 5 show that *GRID-GUM*2 maintains this good performance and sometimes makes a small improvement. The remainder of this section compares the overhead and performance variability of *GRID-GUM*1 and *GRID-GUM*2.

6.1.1 Variability

The measurements in Table 5 have been performed on 10 PEs from the Edin1 cluster. In Table 5, 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.

Table 5 shows that *GRID-GUM*1 gives a highly variable performance, especially for programs with irregular parallelism. In Table 5, the programs with regular parallelism show less variations, for example, 23 percent in queens and 26.7 percent in parFib. However, the programs with irregular parallelism show greater variations, for example, 47 percent in sumEuler, 36.2 percent in linSolv, and 35.2 percent in raytracer.

The unpredictable behavior 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 Section 3.2. The performance is good when idle *GRID-GUM*1 PEs are "lucky" in their random selection of a PE to request work from. The 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 5, queens and parFib show improvements in percentage variance of 98 percent and 84 percent respectively. sumEuler, linSolv, and raytracer, which have irregular parallelism, show improvements of 63 percent, 66 percent, and 82 percent, respectively.

TABLE 6
*GRID-GUM*1 and *GRID-GUM*2 Overhead on 16 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

6.1.2 Overheads

Table 6 compares the overhead induced by *GRID-GUM*1 and *GRID-GUM*2 for the six programs. These measurements are made on 16 PEs from the Edin1 Beowulf cluster, and the runtimes reported are the median of three executions to ameliorate the impact of the 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 the averages over all processors for the maximal heap residency (that is, the maximum amount of heap that is alive at garbage collection time), the allocation rate (that is, the amount of local memory allocated per second of execution time), the communication degree (that is, the number of messages sent per second of execution time), and the average packet size (that is, the size of packets in bytes).

Table 6 shows that, except for the communication degree, *GRID-GUM*1 and *GRID-GUM*2 have similar overheads. *GRID-GUM*2 decreases the communication degree by using the information about the load, latencies, and CPU speeds to reduce the number of work-locating FISH messages.

TABLE 7
*GRID-GUM*1 and *GRID-GUM*2 Performance on Low-Latency Heterogeneous Computational Grids

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%

6.1.3 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 in Table 5).
- *GRID-GUM*2 programs exhibit far less performance variance than *GRID-GUM*1, reducing the variation by at least 63 percent for all programs measured (column 8 in Table 5).
- *GRID-GUM*2 retains a very light overhead, which does not effect the program's dynamic properties (Table 6).

6.2 Low-Latency Heterogeneous Performance

Table 7 reproduces measurements of the *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 four PEs from Edin1 and four PEs from the Edin2 Beowulf cluster. The runtimes reported are the median of three executions to ameliorate the impact of the operating system and shared network interaction.

Table 7 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 percent. 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, as reported in Table 1, 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 percent.

TABLE 8
Homogeneous High-Latency Computational Grids (`raytracer`)

Case	Confi g.	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%

6.2.1 Low-Latency Heterogeneous GRID Performance Summary

Table 7 shows the following points:

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

7 *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.

7.1 High-Latency Homogeneous Performance

Table 8 compares the performance of `raytracer` under *GRID-GUM*1 and *GRID-GUM*2 on all combinations of homogeneous GRIDs with up to five PEs. The configurations combine PEs from two very similar clusters with high-latency interconnect, namely, the Muni and Edin2 Beowulf clusters described in Tables 14 and 15. Each Edin2 machine is labeled E, and each Muni machine is labeled M.

In Table 8, the first and second columns show the case number and GRID configuration. The third column presents the mean communication latency. The fourth and fifth columns record the runtime (in seconds) for *GRID-GUM*1 (GG1) and *GRID-GUM*2 (GG2), respectively. The last column shows the percentage improvement in *GRID-GUM*2 runtime.

*GRID-GUM*2 improves the `raytracer` performance on each of the high-latency homogeneous GRID configurations in Table 8. 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, which is an E PE in this case. As a consequence, the larger number of remote M PEs must communicate with the main PE through the high-latency interconnect. In contrast, in this configuration, $GRID\text{-}GUM2$ selects the main PE from the remote group of M PEs, and hence, a smaller number of PE(s) required 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 Section 5.

In summary, Table 8 shows that $GRID\text{-}GUM2$ outperforms $GRID\text{-}GUM1$ on a high-latency homogeneous architecture for `raytracer`, a program with high communication degree and highly irregular parallelism.

7.1.1 Additional High-Latency Homogeneous Measurements

We have made similar measurements to those reported above for the `queens` and `sumEuler` programs [41]. $GRID\text{-}GUM2$ improves the performance on all high-latency homogeneous GRID configurations measured for both programs, with a maximum improvement of 30 percent for `sumEuler` and a maximum improvement of 9 percent for `queens`.

7.1.2 High-Latency Homogeneous GRID Performance Summary

- $GRID\text{-}GUM2$ outperforms $GRID\text{-}GUM1$ on all of the homogeneous high-latency computational GRID architectures for all three sensitive programs (Table 8).
- $GRID\text{-}GUM2$ improves the performance of programs with a range of parallel behaviors. `raytracer` with high communication degree shows an improvement of up to 37 percent (Table 8). `sumEuler` with low communication degree and irregular parallelism shows an improvement of up to 30 percent. `queens` with low communication degree and regular parallelism exhibits the least improvement of up to 9 percent (Section 7.1.1).

7.2 High-Latency Heterogeneous Performance

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

Table 9 compares the performance of `raytracer` under $GRID\text{-}GUM1$ and $GRID\text{-}GUM2$ on all nontrivial heterogeneous GRIDs with up to five PEs. The improvements are analyzed to identify the improvements due to the use of static and of dynamic information by using the $GRID\text{-}GUM1.1$ experimental RTE outlined in Appendix A.3. The measurements in Table 9 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, the Edin1 and Muni Beowulf clusters are connected over a high-latency interconnect, as detailed in Tables 14 and 15.

TABLE 9
`raytracer`: Heterogeneous High-Latency Computational Grid

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

In Table 9, each Edin1 machine is labeled E, and each Muni machine is labeled M. The first and second columns show the case number and different combinations of PEs from the 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 17 ms and 470 MHz, respectively. The fifth, sixth, and seventh columns record the runtime (in seconds) for $GRID\text{-}GUM1$ ($GG1$), $GRID\text{-}GUM1.1$ ($GG1.1$), and $GRID\text{-}GUM2$ ($GG2$), respectively. The seventh column shows the static information (CPU speed) contribution to the performance change under $GRID\text{-}GUM1.1$ in comparison with $GRID\text{-}GUM1$. The ninth column indicates the dynamic information (loads and latencies) contribution to the change under $GRID\text{-}GUM2$. The last column reports the total performance change by using both static and dynamic information in $GRID\text{-}GUM2$ in comparison with $GRID\text{-}GUM1$.

The additional static information enables a substantial improvement when there are more fast PEs M's than slow PEs E's, that is, cases 1, 2, and 3. For instance, in case 1, $GRID\text{-}GUM1.1$ reduces the runtime by 54 percent. However, the improvement due to static information is less when there are more slow PEs than fast PEs (cases 6, 8, 9, and 10) and may even degrade the performance. For instance, in case 10, $GRID\text{-}GUM1.1$ increases the runtime by 23 percent. This behavior of `raytracer` under $GRID\text{-}GUM1.1$ is related to the high-latency communication. In a configuration of the form $(xEyM)$, where $x > y$ (cases 7, 8, 9, and 10), $GRID\text{-}GUM1.1$ nominates the main PE 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 the $GRID\text{-}GUM1.1$ performance.

The seventh column in Table 9 shows that the use of dynamic load and latency information in $GRID\text{-}GUM2$ improves the performance on 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 ones ($xEyM$, where $x < y$), the dynamic information makes a limited contribution to the performance. For instance, in case 1, the dynamic informa-

tion improves the performance by only 7 percent. In contrast, if there are more slow PEs than fast ones ($xEyM$, where $x > y$), the dynamic information has a greater contribution to the performance. For instance, in case 9, the dynamic information improves the performance by 34 percent. In this case, the dynamic information is used to nominate the main PE 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 a 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 1,490 s. However, *GRID-GUM2* requires only 583 s, which is an improvement of 60 percent.

7.2.1 Additional High-Latency Heterogeneous Measurements

We have made similar measurements to those reported above for the `queens` and `sumEuler` programs [41]. For `sumEuler`, there is a maximum total improvement of *GRID-GUM2* over *GRID-GUM1* of 32 percent, and maximum static and dynamic improvements of 27 percent and 16 percent, respectively. For `queens`, there is a maximum total improvement of *GRID-GUM2* over *GRID-GUM1* of 35 percent, and maximum static and dynamic improvements of 23 percent and 12 percent, respectively.

7.2.2 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 in Table 9).
- *GRID-GUM2*'s static information gives substantial improvements when there are more fast PEs than slow PEs, but this is less when there are more slow PEs than fast PEs (column 7 in Table 9).
- *GRID-GUM2*'s dynamic load and latency information improve the performance on all of the heterogeneous high-latency GRID configurations measured. The improvement is greater if there are more slow PEs than fast ones, and this is less if there are more fast machines than slow ones (column 7 in Table 9).
- For a program with a high communication degree, that is, `raytracer`, *GRID-GUM2* delivers a substantial maximum improvement of 60 percent, whereas for both programs with a relatively low communication degree (`sumEuler` and `queens`), there are more modest improvements of 31 percent and 35 percent (Section 7.2.1).

TABLE 10
GUM and *GRID-GUM1* Scalability (`raytracer`)

Case	No PEs	GUM			*GRID-GUM1*		
		Confi g.	Rtime	Spdup	Confi g.	Rtime	Spdup
1	7	7E	2609	7	6E1M	2530	7
2	14	14E	2168	8	12E2M	2185	8
3	21	21E	1860	10	18E3M	1824	10
4	28	28E	1771	10	24E4M	1776	10
5	30	30E	1762	10			
6	35				30E5M	1666	11
7	41				$5E_2$30E6M	1652	11

8 SCALABILITY

This section 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, which were connected over a low-latency interconnect, and Muni, which was connected with the other two clusters over a high-latency interconnect, as specified in Tables 14 and 15. Because of the relative cluster sizes, many configurations have six Edin1 PEs for every Muni PE.

The experiments have the following limitations:

- The programs in Table 16 were designed for smaller scale HPCs, and only some of them generate sufficient parallelism to utilize medium-scale and large-scale computational GRIDs.
- The numbers of PEs available for these experiments at the cooperating sites were limited: Edin1 E has 30 PEs, Edin2 E_2 has 5 PEs, and Muni M has 6 PEs.

8.1 `raytracer`

The `raytracer` is a realistic parallel program with limited amounts of highly irregular parallelism and a relatively high communication degree (Table 16). Table 10 compares the scalability of the `raytracer` program under GUM and *GRID-GUM1*. The table shows that GUM and *GRID-GUM1* deliver very similar performance levels up to 28 PEs, 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 by using PEs in a remote cluster.

The GRID configurations measured in this section have very similar mean CPU speeds and latencies, namely, 676 MHz and approximately 9.2 ms. Likewise, the configurations have very similar variations in CPU speed and communication latencies, namely, approximately 360 MHz and 15.5 ms, 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 seven-PE configuration, that is, the `raytracer` runtime on the $7E$ row in Table 10 and the `parFib` on the $6E1M$ *GRID-GUM2* row in Table 12.

TABLE 11
*GRID-GUM*1 and *GRID-GUM*2 Scalability (`raytracer`)

Case	No PEs	Confi g.	*GRID-GUM*1 Rtime	Spdup	Eff.	*GRID-GUM*2 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 12
*GRID-GUM*1 and *GRID-GUM*2 Scalability (`parFib`)

Case	No PEs	Confi g.	*GRID-GUM*1 Rtime	Spdup	Eff.	*GRID-GUM*2 Rtime	Spdup	Eff.	Impr%
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 11 compares the scalability and parallel efficiency of the `raytracer` program under *GRID-GUM*1 and *GRID-GUM*2 on a high-latency heterogeneous computational GRID. The efficiency comparison of the two cluster results relies on the similarity of the architectures, that is, six Edinburgh PEs for every Munich PE, and obviates the requirement for a sophisticated calculation of heterogeneous efficiency. The table shows that *GRID-GUM*2 always improves on *GRID-GUM*1 performance. Moreover, although the speedup improvement is modest on small GRIDS, it increases with the GRID size. For example, on the largest, that is, 41-PE, configuration, *GRID-GUM*2 gives a 46 percent improvement, that is, a runtime of 1,133 s compared with 1,652 s for *GRID-GUM*1.

Although *GRID-GUM*2 is always more efficient than *GRID-GUM*1, the absolute efficiency of *GRID-GUM*2 falls significantly to just 38 percent on a 35-PE cluster. Although some of the loss of efficiency is attributable to the high-level DSM programming model, readers should recall that `raytracer` is a challenging program, that is, exhibiting highly irregular parallelism and high levels of communication, executing on a challenging architecture, that is, a high-latency heterogeneous GRID. Section 4 suggests that better speedups and efficiency would be obtained on either a homogeneous GRID or a low-latency GRID. Moreover, Table 12 reports a rather better efficiency for a less challenging program.

8.2 `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 16). Table 12 compares the scalability and efficiency of `parFib` under *GRID-GUM*1 and *GRID-GUM*2 on a high-latency heterogeneous computational GRID. It shows that both *GRID-GUM*1 and *GRID-GUM*2 deliver good and very similar speedups. The speedup is excellent up to 21 PEs, but declines thereafter. The speedup is still increasing, even between 35 and 41 PEs, with a maximum speedup of at least 27 on 41 PEs. *GRID-GUM*2 is again always more efficient than *GRID-GUM*1. Moreover, although the drop in the absolute efficiency to 65 percent on 35 PEs is substantial, it is far less than for the challenging `raytracer`. Section 4 suggests that even better speedups and efficiency would be obtained on either a homogeneous GRID or a low-latency GRID.

The good *GRID-GUM*1 performance reported in Table 12 demonstrates that a sophisticated load distribution is not required for `parFib`. That the *GRID-GUM*2 performance is so similar to the *GRID-GUM*1 performance shows that even on medium-scale computational GRIDs, the overhead of *GRID-GUM*2's load distribution mechanism remains minimal.

8.3 Scalability Summary

- The experiments in this section show that the emerging GRID technology offers the opportunity of improving the performance by integrating remote heterogeneous clusters into a computational GRID (Table 10).
- The measurements in Tables 11 and 12 show that the parallel performance of *GRID-GUM*2 scales to medium-scale heterogeneous high-latency computational GRIDs, that is, 41 PEs in three clusters, and (as shown in Table 11) continues to deliver significant performance benefits over *GRID-GUM*1 for a realistic program.
- The measurements of `parFib`, a program with near-ideal parallel behavior, show that the overhead of the *GRID-GUM*2 load management is relatively low, even on medium-scale computational GRIDs (Table 12).

9 *GRID-GUM*2 Performance Analysis

This section analyzes the performance of the benchmark programs under *GRID-GUM*1, *GRID-GUM*1.1, and *GRID-GUM*2 on combinations of high/low and homogeneous/heterogeneous computational GRIDs with respect to their communication behavior and degree of irregular parallelism. In Table 13, 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 homogeneity/heterogeneity. The sixth, seventh, and eighth columns rank the performance of *GRID-GUM*1 (*GG*1), *GRID-GUM*1.1 (*GG*1.1), and *GRID-GUM*2 (*GG*2), respectively, from 3 (best) to 1 (worst). The last column presents the case number.

We make the following conclusions based on Table 13:

- *GRID-GUM*2'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 parallelize regularly parallel programs on homogeneous computational GRIDs. That is, *GRID-GUM*2 does not reliably improve the

TABLE 13
Comparative Performance Summary: *GRID-GUM*1, *GRID-GUM*1.1, and *GRID-GUM*2

Program	Regul	Comm	GRID Config. Lat.	GRID Config. Archit.	GG1	GG1.1	GG2	Case
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

performance of these programs (cases 14, 16, 18, 20, 22, and 24).

- Static information is the key to effectively parallelizing 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, the adaptive load distribution of *GRID-GUM*2 not only delivers a more predictable performance than *GRID-GUM*1, as shown in Table 5, but also reduces the runtime of all programs.

10 Conclusion

10.1 Summary

We have presented a systematic evaluation of the performance of GPH, the first DSM language with a high-level parallel coordination on computational GRIDs. We report both the absolute performance and the 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-latency/low-latency and homogeneous/heterogeneous 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 RTE with an aggressive and dynamic load management mechanism.

We have summarized an earlier work outlining *GRID-GUM*1, a port of the GUM RTE for GPH originally designed for a single HPC, to computational GRIDs. It showed that *GRID-GUM*1 only reliably delivered good performance on low-latency homogeneous computational GRIDs and that poor load management limits the *GRID-GUM*1 performance. It also outlined the design of *GRID-GUM*2, which is a new RTE incorporating new adaptive load management techniques.

The evaluation of the *GRID-GUM*2 performance covers combinations of high-latency/low-latency and homogeneous/heterogeneous computational GRIDs, with the results outlined in the following paragraphs. Unsurprisingly, *GRID-GUM*2 gives the greatest performance improvements on the most challenging combination: a 60 percent improvement on a heterogeneous high-latency computational GRID (Table 9).

On low-latency homogeneous computational GRIDs, Table 5 shows how *GRID-GUM*2 maintains the good performance of *GRID-GUM*1 reported in Table 1 (Section 6.1). On low-latency heterogeneous computational GRIDS, *GRID-GUM*2 improves the performance of five out of six programs and maintains the good performance of the sixth, although only certain programs are sensitive to heterogeneity (Section 6.2). On high-latency homogeneous computational GRIDS, *GRID-GUM*2 improves the performance of all three programs on all the GRID configurations measured (Section 7.1). On high-latency heterogeneous computational GRIDS, *GRID-GUM*2 improves the performance of all three sensitive programs on all the GRID configurations measured (Section 7.2).

The scalability measurements consider the most challenging but most common computational GRIDs, that is, heterogeneous high-latency GRIDs. The results show that the *GRID-GUM*2 performance scales to medium-scale heterogeneous high-latency computational GRIDs, for example, delivering a speedup of 28 on 41 PEs in three clusters, although the efficiency falls to just 65 percent on this challenging architecture (Section 8). The relative performance of the programs on all combinations of low-latency/high-latency and homogeneous/heterogeneous computational GRIDs has been analyzed with respect to their communication behavior and the 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 9).

10.2 Limitations and Future Work

The current work has the following limitations. The parallel programs measured are small-scale 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 the Globus Toolkit. *GRID-GUM*2 inherits a restriction to closed systems, that is, executing on a fixed set of PEs, from the MPICH-G2 communications library. Currently, *GRID-GUM*2 has no fault-tolerant 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 parallelizing large computer algebra computations as part of the Symbolic Computation Infrastructure for Europe (*SCIEnce*)

TABLE 14
Beowulf Cluster Architectures

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 15
Approximate Intercluster Latencies (in Milliseconds)

	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

Project EU FP VI I3-026133. A second research avenue is to investigate the scalability of *GRID-GUM2* on large-scale computational GRIDs, for example, with hundreds of PEs. Such a GRID is likely to be heterogeneous and have 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 analyze the program behavior to decide whether the execution of the code should be permitted. 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-GUM2* to support open systems, and possibilities include using an optimized version of PVM for computational GRIDs, for example, as done in [42].

A rather more challenging task would be to tackle the problem of fault-tolerant parallel execution on computational GRIDs. Here, *GRID-GUM2* 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 automatically be 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 a property is easier with a high-level virtual machine like *GRID-GUM2*. Indeed, the RTE-level fault tolerance has been proposed for GUM [43].

APPENDIX A

A.1 Hardware Apparatus

The measurements have been performed on five Beowulf clusters: three located at the Heriot-Watt Riccarton campus (*Edin1, Edin2,* and *Edin3*), a cluster located at the Ludwig-Maximilians University, Munich (*Muni*), and a cluster located at the Heriot-Watt Borders campus (*SBC*). See Tables 14 and 15 for the characteristics of these Beowulfs.

A.2 Software Apparatus

Table 16 summarizes 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` program 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` program multiples two matrices. The `linSolv` program finds an exact solution of a linear system of equations.

Three of the programs have regular parallelism, that is, `queens`, `parFib`, and `matMult`, whereas three programs have irregular parallelism, that is, `sumEuler`, `linSolv`, and `raytracer`. Programs with regular parallelism generate threads that 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. Among the programs, `queens`, `sumEuler`, and `linSolv` have relatively low communication degrees, that is, perform relatively little communication per unit of execution time, whereas `parFib`, `matMult`, and `raytracer` have relatively high communication degree, as shown under column 6 in Table 6.

A.3 *GRID-GUM1.1*

A special implementation of *GRID-GUM2*, that is, *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 the CPU speed information to choose a fast PE as the main PE, where the program starts, and to prevent slow PEs from extracting work from faster PEs, unless the latter is the main PE. Unlike *GRID-GUM2*, *GRID-GUM1.1* does not collect or use dynamic information on PE loads and latencies.

A.4 GPH Example: `sumEuler`

As a nontrivial example of the GPH language, the complete code for the `sumEuler` program outlined in Table 16 is given as follows, and the only evaluation strategy required to parallelize the program is at the last line of the `sumTotient` function:

TABLE 16
Program Characteristics

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

```
----------------------------------------
-- 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 Functions: sumTotient & Euler
----------------------------------------
sumTotient :: Int -> Int -> Int -> Int
sumTotient lower upper c =
  sum (map (sum . map Euler)
      (splitAtN c upper, upper-1 .. lower])
      "using" parList rnf)

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

REFERENCES

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

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

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

[4] "MPI: A Message Passing Interface Standard," *Int'l J. Supercomputer Application,* vol. 8, nos. 3-4, pp. 165-414, 1994.

[5] M. Alt, H. Bischof, and S. Gorlatch, "Program Development for Computational Grids Using Skeletons and Performance Prediction," *Proc. Third Int'l Workshop Constructive Methods for Parallel Programming (CMPP '02),* June 2002.

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

[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," *J. Scalable Computing: Practice and Experience,* vol. 6, no. 4, 2006.

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

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

[11] D.B. Loveman, "High Performance Fortran," *IEEE Parallel and Distributed Technology,* 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," *Proc. ACM Conf. Programming Languages Design and Implementation (PLDI '96),* pp. 79-88, http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/gum.ps.gz, May 1996.

[14] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Peña Marí, "Eden: The Paradise of Functional Concurrent Programming," *Proc. European Conf. Parallel Processing (EuroPar '96),* pp. 710-713, 1996.

[15] *The Grid: Blueprint for a New Computing Infrastructure,* I. Foster and C. Kesselman, eds., Morgan Kaufmann, 1999.

[16] Globus, http://www.globus.org/toolkit/, 2005.

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

[18] F. Berman, G. Fox, and T. Hey, "The Grid: Past, Present, Future," *Grid Computing: Making the Global Infrastructure a Reality,* F. Berman, G. Fox, and A. Hey, eds. John Wiley & Sons, pp. 9-50, 2003.

[19] D. Jackson, "Advanced Scheduling of Linux Clusters Using Maui," *Proc. Usenix Ann. Technical Conf. (Usenix '99),* 1999.

[20] E. Smirni and E. Rosti, "Modelling Speedup of SPMD Applications on the Intel Paragon: A Case Study," *Proc. Int'l Conf. and Exhibition High-Performance Computing and Networks, Languages and Computer Architecture (HPCN '95),* 1995.

[21] L. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* vol. 33, no. 8, p. 103, Aug. 1990.

[22] M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, M. Kohl, J. Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam, "HARNESS: A Next Generation Distributed Virtual Machine," *Future Generation Computer Systems,* special issue on metacomputing, vol. 15, nos. 5-6, pp. 571-582, Oct. 1999.

[23] B.-Y. Evan Chang, K. Crary, M. DeLap, R. Harper, J. Liszka, T. Murphy VII, and F. Pfenning, "Trustless Grid Computing in ConCert," *Proc. Third Int'l Workshop Grid Computing (GRID '02),* 2002.

[24] C. Baker-Finch, D. King, J. Hall, and P. Trinder, "An Operational Semantics for Parallel Lazy Evaluation," *Proc. Fifth Int'l Conf. Functional Programming (ICFP '00),* pp. 162-173, Sept. 2000.

[25] T. Murphy VII, K. Crary, and R. Harper, "Distributed Control Flow with Classical Modal Logic," *Proc. 19th Int'l Workshop Computer Science Logic (CSL '05),* pp. 51-69, July 2005.

[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, http://www.ics.uci.edu/ javid/dsm.html/, 2006.
[28] C. Morin, P. Gallard, R. Lottiaux, and G. Valle, "Design and Implementations of NINF: Towards a Global Computing Infrastructure," *Future Generation Computer Systems,* vol. 20, no. 2, 2004.
[29] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs," *J. 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," *Proc. Fifth IEEE Symp. Cluster Computing and the Grid (CCGrid '05),* pp. 905-912, 2005.
[31] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "ASSIST as a Research Framework for High-Performance Grid Programming Environments," *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. 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'l J. High-Performance Computing Applications,* vol. 15, no. 4, pp. 327-344, 2001.
[33] M. Aldinucci, M. Danelutto, and J. Dünnweber, "Optimization Techniques for Implementing Parallel Sckeletons in Grid Environments," *Proc. Fourth Int'l Workshop Constructive Methods for Parallel Programming (CMPP '04),* July 2004.
[34] M. Aldinucci and M. Danelutto, "Advanced Skeleton Programming Systems," *Parallel Computing,* http://www.di.unipi.it/aldinuc/papers.html, 2006.
[35] M. Cole, "Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming," *Parallel Computing,* 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, nos. 7-8, pp. 1079-1107, June 2005.
[37] J. Dünnweber, M. Alt, and S. Gorlatch, "Apis for Grid Programming Using Higher Order Components," *Proc. 12th Global Grid Forum (GGF '04),* http://pvs.uni-muenster.de/pvs/mitarbeiter/jan/adgggf04.html, Sept. 2004.
[38] M. Alt and S. Gorlatch, "Adapting Java RMI for Grid Computing," *Future Generation Computer Systems,* vol. 21, no. 5, pp. 699-707, http://pvs.uni-muenster.de/pvs/publikationen/, 2005.
[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, http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/cpe-gph.ps.gz, 1999.
[40] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface," *J. Parallel Distributed Computing,* vol. 63, no. 5, pp. 551-563, 2003.
[41] A. Al Zain, "Implementing High-Level Parallelism on Computational GRIDs," PhD dissertation, School of Math. and Computer Sciences, Heriot-Watt Univ., http://www.macs.hw.ac.uk/trinder/theses/AlZainAbstract.html, Apr. 2006.
[42] G. Sipos and P. Kacsuk, *Executing and Monitoring PVM Programs in Computational Grids with Jini,* LNCS 2840, J. Dongarra, D. Laforenza, and S. Orlando, eds. Springer, pp. 570-576, http://springerlink. metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2840&spage=570, 2003.
[43] P. Trinder, R. Pointon, and H.-W. Loidl, "Towards Runtime System Level Fault Tolerance for a Distributed Functional Language," *Proc. Second Scottish Functional Programming Workshop (SFP '00),* vol. 2, pp. 103-113, July 2000.

Abdallah D. Al Zain received the BSc degree in computer science from the Applied Science University, Jordan, in 1998 and the PhD degree from the Heriot-Watt University, Edinburgh, in April 2006. His PhD research was implementing high-level parallelism on computational grids. He has a two-year working experience on databases. He is currently a postdoctoral researcher with the EU-funded Symbolic Computation Infrastructure for Europe (SCIEnce) Project that utilizes the Grid-enabled parallel Haskell implementation (Grid-GUM), which he implemented during his PhD research.

Phil W. Trinder received the BSc degree (with honors) from Rhodes University, South Africa, in 1983 and the DPhil degree from Oxford University in 1989. His PhD research was on parallel functional databases. He is currently a senior lecturer of Computer Science at the Heriot-Watt University. He has a strong record of research in the design, implementation, and evaluation of high-level parallel and distributed programming languages. He has held six major UK, EU, and industrial grants to support this activity and has more than 60 publications in refereed journal and conference proceedings publications.

Greg J. Michaelson is the head of the Department of Computer Science, Heriot-Watt University. He has a strong record of research in formally motivated software engineering, with particular expertise in the design and implementation of functional languages for parallel, distributed, mobile, and embedded deployment. He has held seven major UK, EU, and industrial grants to support this activity and has more than 60 publications in refereed journal and conference proceedings. He is a Fellow of the British Computer Society.

Hans-Wolfgang Loidl received the MSc (Dipl-Ing) degree from the Johannes Kepler University, Austria, in 1992 and the PhD degree from the University of Glasgow in 1998. His PhD research was the parallel implementation of functional languages. From 1999 to 2002, he was a postdoctoral research fellow in the Austrian Academy of Sciences, Heriot-Watt University, Edinburgh, working on architecture-independent parallelism. He is currently a postdoctoral researcher with the Theoretical Computer Science Group, Ludwig-Maximilians University, Munich, working on the EU-funded Embounded Project.