Reliable Scalable Symbolic Computation: The Design of SymGridPar2

P. Maier[a,*], R. Stewart[a], P.W. Trinder[b]

[a]*School of Mathematical and Computing Sciences, Heriot-Watt University, Edinburgh, UK*
[b]*School of Computing Science, University of Glasgow, Glasgow, UK*

Abstract

Symbolic computation is an important area of both Mathematics and Computer Science, with many large computations that would benefit from parallel execution. Symbolic computations are, however, challenging to parallelise as they have complex data and control structures, and both dynamic and highly irregular parallelism. The SymGridPar framework has been developed to address these challenges on small-scale parallel architectures. However the multicore revolution means that the number of cores and the number of failures are growing exponentially, and that the communication topology is becoming increasingly complex. Hence an improved parallel symbolic computation framework is required.

This paper presents the design and initial evaluation of SymGridPar2 (SGP2), a successor to SymGridPar that is designed to provide scalability onto 10^5 cores, and hence also provide fault tolerance. We present the SGP2 design goals, principles and architecture. We describe how scalability is achieved using layering and by allowing the programmer to control task placement. We outline how fault tolerance is provided by supervising remote computations, and outline higher-level fault tolerance abstractions.

We describe the SGP2 implementation status and development plans. We report the scalability and efficiency, including weak scaling to about 32,000 cores, and investigate the overheads of tolerating faults for simple symbolic computations.

Keywords: parallel functional programming, locality control, fault tolerance

1. Introduction

Symbolic computation has underpinned key advances in Mathematics and Computer Science, for example in number theory, cryptography, and coding theory. Many symbolic problems are large, and the algorithms often exhibit a high degree of parallelism. However, parallelising symbolic computations poses challenges, as symbolic algorithms tend to employ complex data and control structures. Moreover, the parallelism is often both dynamically generated and highly irregular, e. g. the number and sizes of subtasks may

*Corresponding author
Email address: P.Maier@hw.ac.uk (P. Maier)

vary by several orders of magnitude. The SCIEnce project developed SymGridPar [21] as a standard framework for executing symbolic computations on small-scale parallel architectures (Section 2). SymGridPar uses OpenMath [29] as a lingua franca for communicating mathematical data structures, and dynamic load management for handling dynamic and irregular parallelism.

SymGridPar is not, however, designed for parallel architectures with large numbers of cores. The multicore revolution is driving the number of cores along an exponential curve, but interconnection technology does not scale that fast. Hence many anticipate that processor architectures will have ever deeper memory hierarchies, with memory access latencies varying by several orders of magnitude. The expectation is similar for large scale computing systems, where an increasing number of cores will lead to deeper interconnection networks, with relatively high communication latency between distant cores. Related to the exponential growth in the number of cores is a predicted exponential growth in core failures, as core reliability will remain constant, at best. These trends exacerbate the challenges of exploiting large scale architectures because they require the programmer to pay attention to locality and to guard against failures.

This paper presents the design and initial evaluation of SymGridPar2 (SGP2), a successor to SymGridPar that is designed to scale onto 10^5 cores by providing the programmer with high-level abstractions for locality control and fault tolerance. SGP2 is being developed as part of the UK EPSRC HPC-GAP project, which aims to scale the GAP computer algebra system to large scale clusters and HPC architectures.

The remainder of the paper is organised as follows. Section 2 surveys related work on parallel symbolic computation. Section 3 presents the SGP2 design goals, principles and architecture. A key implementation design decision is to coordinate the parallel computations in HdpH, a scalable fault tolerant domain specific language (Section 3.2).

We describe how scalability is achieved using layering and by allowing the programmer to control task placement on a distance-based abstraction of the communication topology of large architectures (Section 4). We outline how fault tolerance is provided by supervising remote computations. A fault tolerance API is presented, and we sketch higher-level fault tolerance abstractions, namely supervised skeletons (Section 5).

SGP2 is still under development, and we outline the current implementation and give preliminary scalability and fault tolerance results. Specifically, we investigate the scalability and efficiency of a layered task placement strategy on approximately 2000 cores of an HPC architecture (Section 6.2) , and we evaluate the overheads of a fault tolerant skeleton on a Beowulf cluster, both in the presence and absence of faults (Section 6.3).

Context.. This paper is an extension of a 2013 ACM Symposium on Applied Computing (SAC'13) paper [24]. It goes beyond the conference paper in the following ways.

It provides more detail on topology-aware work stealing in Section 4 and a more extensive discussion of SGP2 fault tolerance in Section 5, including a new fault tolerance work stealing protocol. Section 6 also presents new experimental results, including a demonstration of weak scaling of an SGP2 prototype up to 32K cores.

2. Related Work

2.1. Symbolic Computation and GAP

Symbolic Computation has played an important role in a number of notable mathematical developments, for example in the classification of finite simple groups. It is essential in several areas of mathematics which apply to computer science, such as formal languages, coding theory, or cryptography. Computational Algebra (CA) is an important class of Symbolic Computation (SC) where applications are typically characterised by complex and expensive computations that would benefit from parallel computation. Application developers are typically mathematicians or other domain experts, who may not possess parallel expertise or have the time/inclination to learn complicated parallel systems interfaces.

There are several Computational Algebra Systems (CAS) that often specialise in some mathematical area, for example Maple [6], Kant [10], or GAP [13]. GAP is a free-to-use, open source system for computational discrete algebra, which focuses on computational group theory. It provides a high-level domain-specific programming language, a library of algebraic functions, and libraries of common algebraic objects. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, and combinatorial structures.

2.2. Orchestrating CAS with SCSCP

The Symbolic Computation Software Composability Protocol (SCSCP) is a lightweight protocol for orchestrating CAS developed in the SCIEnce project [21]. In essence the protocol allows a CAS to make a remote procedure call to another CAS. Hence SCSCP compliant CAS may be combined to solve scientific problems that cannot be solved within a single CAS, or may be orchestrated for parallelism.

In SCSCP both data and instructions are represented as OpenMath objects. OpenMath is a standard markup language for specifying the meaning of mathematical formulae [29]. SCSCP has become a *de facto* standard, with implementations for 9 CAS and libraries for several languages including Java, C++, and Haskell [21].

2.3. Parallel Symbolic Computation

Some discrete mathematical problems, especially in number theory, exhibit *trivial parallelism*, where they can be partitioned into relatively large, totally independent pieces of predictable size. Mathematicians have for many years parallelised these computations by running different pieces on different computers. In extreme cases, the primitive steps are so simple and independent that they are amenable to internet-wide distributed computation as in the "Great Internet Mersenne Prime Search", which recently found a record-breaking prime number,with 12 978 189 digits.

Numerous authors have developed parallel algorithms and implementations of a variety of mathematical computations, and even developed general frameworks intended to simplify parallel programming for mathematical users e.g. [27, 18, 37]. Of particular relevance is the `ParGAP` system [8], which provided bindings to the MPI library in the GAP language. Most of these systems were specific to now obsolete hardware, and none has achieved wide usage.

In the recent SCIEnce project, a European consortium have investigated parallelising a range of algebraic computations in a Grid context. The consortium designed and

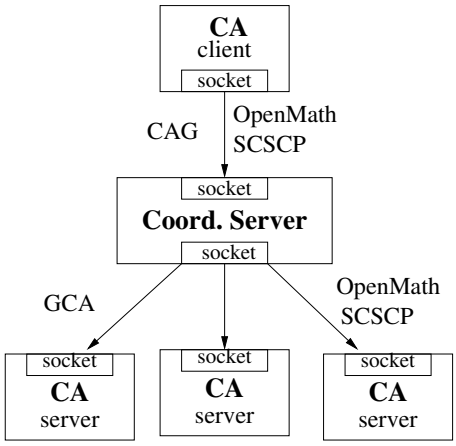


Figure 1: SymGridPar and SymGridPar2 Architecture

exploited the general-purpose skeleton-based *SymGridPar* framework outlined in the next section.

These, and other experiences, show that parallel algebraic computations pose additional and specific problems, as follows. Parallel algebraic computations exhibit high degrees of irregularity, with varying numbers and sizes of tasks. Some computations have both multiple levels of irregularity, and enormous (5 orders of magnitude) variation in task sizes [36]. They use complex user-defined data structures. They have complex control flows, often exploiting recursion. They make little, if any, use of floating-point operations.

This combination of irregularity, recursive structure and limited use of floating-point operations imply that computational algebra problems are unsuitable for relatively inflexible HPC acceleration techniques like vectorisation or FPGAs, rather they must use architectures based on general-purpose cores.

Moreover, explicit parallel paradigms are unlikely to deal effectively with the highly irregular computation structure, and this motivates our decision to develop a scheduling and management framework.

2.4. The SymGridPar Framework

The SymGridPar middleware [21] orchestrates sequential SCSCP-compliant CAS into a parallel application. SymGridPar has been designed to achieve a high degree of flexibility in constructing a platform for high-performance, distributed symbolic computation, including multiple CAS. Although designed for distributed memory architectures, it also delivers good performance on shared memory architectures [2].

The SymGridPar architecture is shown in Figure 1, and has three main components.

The Client. The end user works in his/her own familiar programming environment, so avoiding the need to learn a new CAS, or a new language to exploit parallelism. The coordination layer is almost completely hidden from the CAS end user: they work exactly as they would with the CAS apart from calling some algorithmic skeletons to introduce parallelism [7]. This set of skeletons are the *CAG* interface to the coordination server.

Some CAG skeletons are generic, e.g. a `parMap` applies a function to every element of a list in parallel. Other skeletons are specific to the CA domain, e.g. a multiple homomorphic image skeleton solves each image in parallel.

The Coordination Server. This middleware provides parallelised services and parallel skeleton implementations. The skeleton implementations delegate work (usually calls to expensive computational algebra routines) to the Computation Server, via the *GCA* interface. The core of GCA are SCSCP remote procedure calls to SCSCP-compliant CAS instances. Currently the Coordination Server is implemented in Eden [22], a parallel Haskell dialect, allowing the user to exploit dynamic load management, polymorphism, and higher order functions for the effective implementation of high-performance parallelism.

The Computation Server. This component is a parallel machine with one or more CAS instances. For example a Beowulf cluster of 16 core nodes, and 16 instances of GAP on each node. Each server handles the requests that are sent to it, and returns the results to the coordination server. Finally, the coordination server may combine the results for returning to the client.

2.5. A Critique of SymGridPar for Impending Architectures

The multicore revolution is leading to the number of cores in commodity architectures growing exponentially. Many expect 100,000 core platforms to become commonplace. The data centre and High Performance Computing (HPC) communities already operate at this scale, and beyond. Indeed some mathematicians have access to architectures at this scale. Many more mathematicians have access to ubiquitous commodity clusters, e.g. University compute servers. While currently a typical cluster might have a few thousand cores, the exponential growth in cores will soon see this class of architecture reach a 100 000 core scale. Hence parallel systems must be designed for far greater scale than previously.

Moreover, hardware failures on architectures with 100 000 cores are relatively common, even hourly [4]. For example data centres operating at this scale require software frameworks like Hadoop or Google MapReduce that provide mechanisms for automatically recovering from faults [12]. So software designed for parallel systems need to be both scalable and fault tolerant.

SymGridPar has been evaluated on a range of architectures including a 32-node Beowulf cluster [36] using benchmarks similar to the ones reported in Section 6. Although SymGridPar achieved speedups up to 30 on the 32-core cluster it was never designed to scale to thousands of cores, let alone hundreds of thousands. Specifically its load management is unlikely to scale beyond a few hundred cores, nor does it provide any abstractions for controlling locality or tolerating failures. In part, these deficiencies are down to the SGP Coordination Server being implemented in Eden, which provides neither fault tolerance nor the locality control required by large architectures. The design presented in the following sections addresses these shortcomings.

3. SGP2 Architecture

3.1. SGP2 Design Goals and Principles

SymGridPar2 (SGP2) is designed as a successor to SymGridPar that shall meet the following four design goals.

1. *Scale symbolic computation to architectures with* 10^5 *cores.*
2. *Topology awareness* to cope with increasingly non-uniform communication topologies implied by scaling to 10^5 cores.
3. *Fault tolerance* to cope with increasingly frequent component failures implied by scaling to 10^5 cores.
4. Preserve the user experience of SGP, specifically the high-level *skeleton API*. That is, to the CAS user SGP2 will look like SGP, apart from a few new skeleton parameters for tuning locality control and/or fault tolerance.

Orthogonal to the above four design goals, SGP2 aims to embody the following design principles. First, the design of SGP2 is *layered*. That is, the most high-level abstractions, e. g. topology aware fault tolerant skeletons, are implemented in terms of simpler abstractions, e. g. plain skeletons, and simpler primitives.

Second, to support dynamic and irregular parallelism, task placement in SGP2 should avoid explicit choice wherever possible. Instead, choice should be semi-explicit, i. e. the programmer decides which tasks are suitable for parallel execution and possibly at what distance from the current processing element (PE) they should be executed. However, the actual decisions where to schedule work should be taken at runtime by the system rather than by the programmer.

3.2. SGP2 Architecture and HdpH

SGP2 retains the component architecture of SGP, as depicted in Figure 1, but provides a scalable fault tolerant Coordination Server component. The key implementation design decision is to realise the Coordination Server using the HdpH domain specific language (DSL) [25], designed to deliver scalable fault tolerant symbolic computation. HdpH (Haskell distributed-parallel Haskell) is a shallowly embedded parallel extension of Haskell that supports high-level semi-explicit parallelism on distributed-memory architectures. As suggested by the first "Haskell" in its name, HdpH is implemented in Haskell (with GHC extensions). Relying solely on the widely available GHC rather than requiring a bespoke low-level parallel runtime system makes HdpH portable and aids maintainability.

HdpH extends the `Par` monad DSL [26] for shared-memory parallelism to distributed memory. Figure 2 lists the HdpH primitives in so far as they are relevant for this paper.

HdpH focuses on task parallelism. A *task* computing a value of type `a` is an expression of type `Closure (Par (Closure a))`, i. e. a serialisable monadic computation that will deliver a serialisable value of type `a`. HdpH offers two modes of task distribution. The `spawnAt` primitive eagerly places a task on a named PE, where it is immediately executed. The `spawn` primitive, in contrast, places the task into a local task pool, from where it may be selected for execution by the current PE or stolen by other PEs looking for work; thus `spawn` provides on-demand (lazy) implicit task placement via distributed work stealing. Note that work stealing migrates tasks only prior to execution; tasks that have been selected for execution remain committed to the executing PE.

```
-- Par monad
type Par a  -- monadic computation returning type 'a'
eval :: a → Par a  -- strict evaluation in the the Par monad

-- distribution of tasks via explicit closures
type Task a = Closure (Par (Closure a))
type Closure a  -- explicit closure of type 'a', serialisable
spawn   ::       Task a → Par (Future a)  -- lazy & implicit placement
spawnAt :: PE → Task a → Par (Future a)  -- eager & explicit placement

-- communication of results via futures
type Future a = IVar (Closure a)
type IVar a  -- write-once buffer of type 'a'
probe :: Future a → Par Bool         -- local test, non-blocking
get   :: Future a → Par (Closure a)  -- local read, blocking
```

Figure 2: HdpH primitives.

Both `spawn` and `spawnAt` immediately return a *future* [17] of type `IVar (Closure a)` to the caller. Such a future is a write-once buffer (also known as an *IVar*) expecting the result of the task, an explicit closure of type `a`. A future can only be read by `get`, which will block until the closure is available, and tested by `probe`, which will indicate whether `get` would block. Futures are tied to the task that created them (by `spawn` or `spawnAt`); like tasks under execution, they are not serialisable and cannot migrate, hence they must be tested and read on the node where they were created. In this respect, futures are similiar to channels in Eden [22], supporting remote write but only local read.

As an aside, we note that HdpH supports a slightly richer set of primitives, separating the creation of futures from task distribution [25]. However, separating tasks and futures complicates the semantics of the language considerably, in particular when modeling fault-tolerance. Yet, the additional expressiveness of separate futures is rarely necessary to write task parallel skeletons — none of the skeletons in [25] used it.

The polymorphic `Closure` data type is central to communication in HdpH as only closures can be sent over the network. HdpH provides the following primitive operations: `unClosure` unwraps a `Closure a` and returns its value of type `a`; `toClosure` wraps a value of any serialisable type `a` into a `Closure a`. Additionally, the Template Haskell construct `$(mkClosure [| e |])` constructs a `Closure a` wrapping the unevaluated thunk `e` of type `a`; thus closures can wrap both values and computations. Efficient higher-level operations, like function closure application, are built on top of these primitives. Moreover HdpH provides a library of *algorithmic skeletons* [7], high-level abstractions for parallelism, built on top of the primitives.

A core feature of the HdpH system is its two-level work stealing scheduler, that combines local work stealing (from cores on the same node) with distributed work stealing (over the network) in a sophisticated way. This enables the HdpH coordination server to adapt to the irregular and dynamic parallelism exhibited by symbolic computations.

Sections 4 and 5 will extend HdpH with support for locality control and fault tolerance. The following glossary summarises how the central concepts, tasks and futures, need to be adapted to achieve this goal.

Term	Description
Future	A variable that initially holds no value. It will eventually be filled with the result (in the form of a *closure*) of its associated *task*.
Task	A (serialisable) suspended computation. It will eventually be evaluated and its result written to its corresponding *future*.
Bounded task	A task which can be scheduled only within a bounded *radius* around its creator.
Supervised future	Same as a *future*, with the additional guarantee of eventually being filled even in the presence of node failures.
Supervised task	A task whose current location is transparently tracked by its creator (the *supervisor*). Supervised tasks are transparently replicated and re-scheduled as needed to compensate node failures.

Table 1: HdpH Glossary: Tasks and Futures

4. SGP2 Locality Control

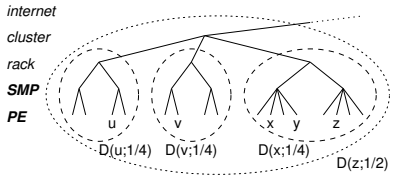
Historically many parallel architectures have had a flat communication topology: the communication latency between any pair of processors is approximately the same. Parallel programming models like MPI [28] exploit this simplified model.

As the number of cores grows however, we find that large scale architectures necessarily have a hierarchical communication topology. In a typical multicore cluster, cores can communicate more quickly with other cores on the same node than with cores on other nodes. To the programmer this manifests itself in significant differences in message passing latency. For example, latency to shared memory on modern commodity multicores is about 100 ns [19], whereas network latency between neighboring nodes is about 1 μs even in modern supercomputers [30]. Network latency increases further with every switch or router messages have to pass; at the same time, effective bandwidth decreases due to the risk of congestion along the way.

Large-scale parallel programming needs to be aware of the network topology, as both locality information (in the problem domain) and network topology information are necessary to efficiently schedule parallelism on large systems. The SGP2 design exposes the network topology as an abstract distance metric, and lets the programmer express locality in terms of these abstract distances. Thus, the distance metric enables locality control but avoids the temptation to code for a specific topology.

4.1. Distance Metric and Equidistant Bases

We take an abstract view of the network topology, modelling it as a hierarchy, as for example in Figure 3, i. e. an unordered tree whose leaves correspond to processing elements (PEs). Every subtree of the hierarchy forms a *virtual cluster*. The interpretation of these virtual clusters is not fixed. Figure 3 suggests the interpretation that a subtree of depth 1 represents a shared memory multicore (SMP) node, a subtree of depth 2 represents a



d	u	v	x	y	z
u	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
v	$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
x	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{8}$	$\frac{1}{4}$
y	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{8}$	0	$\frac{1}{4}$
z	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	0

Figure 3: Hierarchy, Distance Metric and Equidistant Partition

rack consisting of several multicores, a subtree of depth 3 represents a server room with several racks, and a subtree of depth 4 (i. e. the whole hierarchy in Figure 3) represents several clusters connected over the internet. Similar four-level hierarchies exist in data center networks, which typically connect thousands of multicore nodes via multi-rooted tree or fat tree topologies [1] with three layers of switches (edge, aggregate, and core).

The hierarchy is characterised by a *distance* function d on PEs, see Figure 3, which is defined by

$$d(p,q) = \begin{cases} 0 & \text{if } p = q \\ 2^{-n} & \text{if } p \neq q \text{ and } n = \text{length of longest common path from root to } p \text{ and } q. \end{cases}$$

Mathematically speaking, the distance function defines an *ultrametric space* on the set of PEs. That is, d is non-negative, symmetric, 0 on the diagonal, and satisfies the *strong triangle inequality*: $d(p_1,p_3) \leq \max\{d(p_1,p_2), d(p_2,p_3)\}$ for all PEs p_1, p_2, p_3. We observe that all non-zero distances are isolated points in the real interval $[0,1]$, and we denote the *set of distances* by $\mathit{range}\, d = \{2^{-n} \mid n = 0,1,\dots\} \cup \{0\}$.

The reason for defining d as above is precisely the fact that d is not just a metric but an ultrametric, which is essential for the existence of *equidistant bases*, an important concept to be defined below. Given a PE p and $r \geq 0$, define $D(p;r) = \{q \mid d(p,q) \leq r\}$ to be the *ball*[1] with center p and radius r. Balls correspond to virtual clusters in the hierarchy, see Figure 3 for a few examples. Balls have the following properties, which are straightforward consequences of d being an ultrametric.

(B1) Every PE inside a ball is its center. That is, for all p, q and r, $d(p,q) \leq r$ implies $D(p;r) = D(q;r)$.

(B2) Every ball of radius $r \in \mathit{range}\, d$ is uniquely partitioned by a set of balls of radius $\frac{1}{2}r$, the centers of which are pairwise spaced distance r apart. That is, $D(p;r)$ is partitioned by the set $\{D(q;\frac{1}{2}r) \mid q \in D(p;r)\}$, and $d(q,q') = r$ for any two distinct balls $D(q;\frac{1}{2}r)$ and $D(q';\frac{1}{2}r)$ in the partition.

We call the set $\{D(q;\frac{1}{2}r) \mid q \in D(p;r)\}$ the *equidistant partition* of $D(p;r)$. A set Q of PEs is an *equidistant basis* for $D(p;r)$ if Q contains exactly one center of each ball in the equidistant partition of $D(p;r)$. That is, $\{D(q;\frac{1}{2}r) \mid q \in Q\} = \{D(q;\frac{1}{2}r) \mid q \in D(p;r)\}$ and for all $q, q' \in Q$, $D(q;r) = D(q';r)$ implies $q = q'$. To illustrate, Figure 3 shows the equidistant partition of $D(z;\frac{1}{2})$, from which we can read off that $\{u,v,x\}$ is one equidistant basis.

[1] More accurately, $D(p;r)$ is known as a *closed ball* or *disk*.

Our abstract view means that the hierarchy need not exactly reflect the physical network topology. Rather, it presents a logical arrangement of the network into a hierarchy of clusters of manageable size. For example two small Ethernet clusters networked by a fast, high bandwidth WAN may be treated as a single cluster. However, since one motivation for topology awareness is to enable SGP2 to take communication costs into account, actual latencies should be reasonably compatible with the distance metric, i. e. with increasing distance actual latency should increase rather than decrease.

The remainder of this section describes how SGP2 will realise topology awareness by integrating the distance metric into both explicit work placement and work stealing primitives in HdpH. For ease of use SGP2 will provide topology aware skeletons implemented as HdpH skeletons.

4.2. Lazy Work Stealing

HdpH requires only a small change to allow the programmer to control the locality of tasks distributed via random stealing. HdpH will expose the set of distances, *range* d, as an abstract type, `Dist`, and add a *radius* parameter (of type `Dist`) to the `spawn` primitive:

```
boundedSpawn :: Dist → Task a → Par (Future a)
```

The radius r constrains how far a task can travel from the spawning PE p_0: it can be stolen precisely by the PEs in the closed ball $D(p_0; r)$. The corner cases deserve special attention.

- Radius $r = 1$ imposes no locality constraint at all, i. e. the task may be stolen by any PE.
- Radius $r = 0$ pins the task to p_0, i. e. it cannot be stolen at all. Thus $r = 0$ can express co-location of tasks.

The remainder of this subsection details aspects of HdpH's topology aware work stealing algorithm, including its task selection policy. Let p_0 be the current PE.

When p_0 executes the primitive `boundedSpawn r task`, it adds the pair `(task,r)` to its *task pool* data structure. We call the pair `(task,r)` a *bounded task* (with radius `r`).

When p_0 runs out of work, and its own task pool is non-empty, it uses the following *local task selection policy*: Pick a task with minimal radius; if there are several such tasks, other criteria can be considered, e. g. picking the youngest such task. Thus, HdpH prioritises tasks with small radius for local scheduling.

If, on the other hand, p_0 runs out of work with its own task pool empty then it will attempt to steal work by sending a FISH message to a random PE. In fact p_0 does not wait for its task pool to drain completely; to hide latency p_0 will attempt to steal work when the pool hits a minimum number, the so-called *low water mark*.

When p_0 receives a FISH message from another PE p, it tries to find a suitable task using the following *remote task selection policy*: Pick a task with minimal radius from the set of tasks whose radius is greater or equal to $d(p_0, p)$; if there are several such tasks, pick the oldest one. Thus for remote scheduling, HdpH prioritises tasks whose radii match the distance to the PE requesting work. The following equation formalizes remote task selection; it uses the radius-indexed notation $pool_r$ to denote the set of bounded tasks of radius r in the task *pool*.

```
handle_FISH (thief :: PE) = do
  let pool = my task pool
  let r = d(me,thief)
  if select(pool,r) = (task,r')
    then send SCHEDULE(task,r') to thief
    else do
      let victim = random PE such that d(victim,thief) ≥ r
      send FISH(thief) to victim
```

Figure 4: Work stealing algorithm (pseudo code)

$$select(pool, r) = \begin{cases} (task, r') & \text{if } (task, r') \text{ is the oldest task in } pool_{r'} \text{ and} \\ & \quad r' \geq r \text{ is the least radius such that } pool_{r'} \neq \emptyset \\ \emptyset & \text{if } pool_{r'} = \emptyset \text{ for all } r' \geq r \end{cases}$$

If this policy does not yield a suitable task then p_0 forwards p's FISH message to a random PE. If, however, the remote task selection policy does yield a `(task,r)` then p_0 sends it in a SCHEDULE message to the requesting PE p, which will place the task in its task pool, from where it will either be scheduled for local execution, or sent to yet another PE looking for work. Note that due to property (B1), $D(p_0; \mathtt{r}) = D(p; \mathtt{r})$, i. e. both p_0 and p are centers of the same ball of PEs eligible to execute `(task,r)`.

To prioritise local stealing, the work search algorithm is not always random. When p_0 first sends a FISH it targets a random PE *nearby*. And when p_0 forwards a request for work from another PE p, it will forward to a random PE at a distance greater or equal to $d(p_0, p)$, see Figure 4. Note that the distribution used for random selection is not uniform but skewed to favour PEs that minimise the distance constraint. Thus, a request for work targets nearby PEs first, looking for local work, and then travels further and further afield in search for work. To prevent the network being swamped with requests for work at times when there is little, FISH messages expire after being forwarded a number of times. Upon expiry, the last PE targeted by the FISH message sends a NOWORK message to the requesting PE p_0, which, upon receiving NOWORK, backs off for some time before repeating the request.

The task selection policies can be summed up as: Tasks with small radii are preferred for local execution, and the bigger the radius, the further a task should travel. This design is consistent with the findings of [16], which investigated scheduling strategies based on granularity information (asserted by the program annotations) and found that the optimal strategy schedules small tasks locally and large tasks remotely. With this scheduling strategy in mind, task radii can be given a less operational meaning. Instead of as distance bounds constraining locality, radii may be viewed as rough estimates of granularity — the larger a task's radius, the larger the task.

Note that the `boundedSpawn` primitive still falls into the class of semi-explicit parallel programming interfaces. It is not an explicit interface because it does not expose locations, and because it leaves the actual scheduling decisions to the runtime system's work stealing algorithm. The task radii only allow the programmer to constrain the runtime system's choices to better take locality into account.

4.3. Eager Work Placement

Random work stealing performs well with irregular parallelism. However, it tends to under-utilise large scale architectures at the beginning of the computation. To combat this drawback, SGP2 complements random stealing with explicit placement. Explicit placement differs from random stealing in several dimensions:

- Placement is mandatory and explicitly controlled by the programmer, i. e. concrete locations are exposed.
- Placement is eager, i. e. an explicitly placed task will be scheduled for execution immediately, taking priority over any stolen tasks.

HdpH already supports eager work placement via `spawnAt`. In order to support topology aware placement, HdpH has to be made fully location aware by exposing the following additional primitives:

```
dist :: PE → PE → Dist
equiDist :: Dist → Par [(PE, Int)]
```

The function `dist` is the reification of the distance metric d. The primitive `equiDist` takes a radius r and returns a size-enriched equidistant basis for $D(p_0; r)$, where p_0 is the current PE. More precisely, it returns a non-empty list `[(`q_0`,`n_0`),(`q_1`,`n_1`),...]` such that

- n_i is the size of $D(q_i; \frac{1}{2}r)$, i. e. n_i equals the number of PEs clustered up to distance $\frac{1}{2}r$ around q_i, and
- the q_i form an equidistant basis for $D(p_0; r)$.

By convention, the first PE q_0 is always the current PE p_0, which can be used to discover the identity of the current PE programmatically.

Note that the primitive `equiDist` is well-defined thanks to the properties (B1) and (B2) of ultrametric spaces mentioned in Section (4.1). (B2) guarantees the existence of equidistant bases. However, due to (B1), these bases are not unique, so the runtime system may pick one of many alternatives at random. For example, given the network topology in Figure 3, calling `equiDist` $\frac{1}{2}$ on PE `u` might return `[(u,4),(v,4),(x,8)]` or `[(u,4),(v,4),(y,8)]` or `[(u,4),(v,4),(z,8)]`.

The sizes n_i in an equidistant basis are intended to measure the compute power clustered around the PEs q_i, respectively, and hence assume that all PEs are homogeneous. The homogeneity requirement can be relaxed by reporting "sizes" n_i relating to the actual compute power (e. g. measured by benchmarking) of the PEs clustered around the q_i rather than just the number of such PEs.

HdpH does not expose a primitive returning the set of all PEs as it would be prohibitively expensive on any large architecture. Instead, HdpH only maintains a distance-indexed table of the bases returned by `equiDist`, and the space required to store this table typically scales logarithmically with the number of cores. The set of all PEs can be computed from the equidistant bases by a (costly) distributed gather operation.

```
parMapLocal  -- bounded work stealing parallel map skeleton
  :: Dist              -- bounding radius
  → Closure (a → b)    -- function closure
  → [Closure a]        -- input list
  → Par [Closure b]    -- output list
parMapLocal r c_f cs = mapM apply_c_f cs >>= mapM get
  where
    apply_c_f c = boundedSpawn r $(mkClosure [|eval $ toClosure (unClosure c_f $ unClosure c)|])

parMap2Level, parMap2LevelRelaxed  -- two-level bounded parallel map skeletons
  :: Dist              -- bounding radius
  → Closure (a → b)    -- function closure
  → [Closure a]        -- input list
  → Par [Closure b]    -- output list
parMap2Level r c_f cs = do
  qs ← equiDist r
  let qcs = chunk qs cs
  futs ← mapM spawnParMap qcs
  concat <$> mapM (λ fut → unClosure <$> get fut) futs
    where
      spawnParMap (q,cs_q) = spawnAt q $(mkClosure [|toClosure <$> parMapLocal (r/2) c_f cs_q|])

parMap2LevelRelaxed r f_f cs = do
  -- same as parMap2Level
    where
      spawnParMap (q,cs_q) = spawnAt q $(mkClosure [|toClosure <$> parMapLocal r c_f cs_q|])
```

Figure 5: Topology Aware Algorithmic Skeletons

4.4. Topology Aware Skeletons

HdpH provides a library of *topology aware algorithmic skeletons* that abstract over the topology aware primitives. For example Figure 5 shows three versions of a parallel map over a list. All skeletons take an extra radius parameter for locality control. Note that for distribution over the network HdpH requires the function argument and the list elements to be `Closures`. Skeletons similar to these are benchmarked in Section 6.2.

`parMapLocal` creates tasks bounded by radius `r`, resulting in a lazy distribution of the parallel work across the network to PEs no further than distance `r` from the PE calling `parMapLocal`. PEs beyond this distance will receive no tasks from this skeleton as their communication latency is expected to outweigh the benefit of the additional parallelism.

`parMap2Level` uses a combination of eager and lazy work distribution. It obtains an equidistant basis `qs` with radius `r` and splits the input list into chunks, one per basis PE, taking into account the size information present in the basis `qs`. Then the skeleton eagerly pushes big tasks to the basis PEs, one per PE. Each big task in turn calls `parMapLocal` on its chunk of the input list, restricting the radius to $\frac{1}{2}$`r`. This results is a quick distribution of *big* tasks to PEs far from the caller, and these PEs then act as local coordinators by spawning *small* tasks to be evaluated in their vicinity. Thanks to bounded task creation and equidistance of the coordinators, it is guaranteed that the small tasks spawned by one local coordinator stay in its vicinity; in particular, they cannot travel to a PE in the vicinity of another local coordinator.

Finally, `parMap2LevelRelaxed` is a variant of `parMap2Level` that only differs from the latter in the bound it imposes on small tasks. `parMap2LevelRelaxed` relaxes this

bound to from $\frac{1}{2}$`r` to `r`. The effect is to allow the stealing of small tasks even between previously isolated local coordinators, which can help mitigate imbalances in task distribution arising from irregular parallelism. Due to the work stealing algorithm's preference for local work, the additional stealing due to the relaxed bound tends to be a last resort, occurring mostly in the final stages of a computation when work is drying up. Our results in Section 6.2 demonstrate the benefits of this skeleton.

We stress that all three of the above skeletons implement a semi-explicit interface in that they allow for tuning of locality via a single radius parameter, without ever exposing locations to the programmer. This abstract locality control is intended to facilitate performance portability between parallel architectures.

4.5. Topology Aware Related Work

We have investigated adding semi-explicit topology awareness in a Glasgow parallel Haskell (GpH) [34] implementation [3], effectively prototyping SGP2 topology awareness. In architecture-aware GpH the *levels* of the architecture are exposed, rather than distances. Spawning primitives, variants of the `par` combinator, specify a minimum, a maximum, or both minimum and maximum levels that the task (spark) may travel to. Like SGP2 GpH uses lazy work stealing informed by the level information associated with tasks. Architecture-aware GpH provides skeletons that are parameterised by communication level, analogous to SGP2 topology aware skeletons. We briefly outline the performance of topology awareness in GpH when describing initial SGP2 performance in Section 6.2.

5. SGP2 Fault Tolerance

Fault tolerance is a means to unlock SymGridPar2 scalability ambitions for reliable long-running computations on massively parallel systems. Many fault tolerant approaches in distributed architectures follow a rollback-recovery approach, often involving checkpointing and synchronisation phases. As the bandwidth per core falls, it is becoming increasingly hard to checkpoint large HPC systems, and alternatives are being sought. At the highest level, fault oblivious and self stabilising algorithms [31] have been developed for imprecise applications such as stochastic simulations, which often involve a balance between precision and reliability. Such a trade-off is impossible in the SGP2 design, where the symbolic computing domain requires solutions to be exact.

A lower level and popular approach for achieving fault tolerance in HPC systems is to adopt a resilient MPI communication layer. Thorough comparisons of fault tolerant MPI approaches and implementations have been made [14]. These include checkpointing the state of computation, or extending the semantics of the MPI standard. In either case, the onus is often on the user to handle faults programmatically.

The fault tolerance model in SGP2 exploits the loosely coupled design of HdpH, which separates remote tasks and futures, and is described in Section 5.1. Two new primitives for fault tolerance are added to the HdpH API (Section 5.2). They use a reliability extension of the scheduler that guarantees the execution of supervised tasks. Section 5.3 describes the reliable scheduling extension, introducing a fault tolerant fishing protocol. Section 5.4 returns to the parallel skeletons introduced in Section 4.4, and outlines fault tolerant parallelism abstractions.

5.1. Fault Tolerance Model

The key to SGP2 fault tolerance is the *supervision* and *replication* of remote tasks in HdpH. The API of the fault tolerant work distribution primitives Section 5.2 is similar to the non-fault tolerant APIs in Sections 3.2 and 4.2. Spawning a computation creates a local empty future, and a task that may be lazily scheduled as before (Section 4.2). This allows users to easily opt-in to reliable scheduling. Failure recovery is provided by the HdpH scheduler (Section 5.1.3). The user does not need to handle failure in the program, in contrast to most fault tolerant MPI approaches.

The reliable scheduler in HdpH guarantees that supervised futures will be filled eventually, even in the presence of node failures, provided the node hosting the supervised future (i. e. the *supervisor*) does not die. Supervision may be nested; for instance, a supervised task may spawn further supervised tasks. This results in a tree of supervised tasks and futures, with the root node functioning as the top level supervisor.

5.1.1. Task Tracking

The work stealing architecture (Section 4.2) of HdpH migrates tasks from busy nodes to idle nodes. When the supervisor detects faults (Section 5.1.2), the location of *supervised* tasks needs to be known in order to recover potentially lost supervised tasks.

The reliable scheduler in HdpH uses a more elaborate fishing protocol (Section 5.3) to the default fault *oblivious* protocol in Section 4.2. Additional messages (NOTIFY and ACK) let supervisors track the locations of their supervised tasks.

5.1.2. Failure Detection

Failures are identified by detecting lost connections. There are numerous possible causes for a connection loss, including loss of power, corrupt hard drive, operating system crash, or a networking failure. Yet the cause does not matter, the consequence of failure is the same. That is, the node affected by the failure can no longer participate in the execution of a distributed program.

Connection loss detection in HdpH uses a network abstraction layer [9] that propagates failure events. The fault recovery (Section 5.1.3) procedure is initiated by recipients of these node failure messages.

5.1.3. Failure Recovery

The reliable scheduler replicates any supervised tasks that may have been lost (i. e. that have not yet filled their associated future) as a result of the detected failure. Candidates for replication are either supervised tasks that were explicitly placed on the failed node, or supervised tasks that had migrated to the failed node via work stealing.

Pessimistic Replication. A supervisor recovers from failure by ensuring the liveness of supervised tasks that *may* have been lost as a consequence of failure. For this purpose, the supervisor remembers for each supervised task: the task itself, the associated future, and a *task location record*. The latter is a set of locations where the supervisor believes the supervised task to be, based on task tracking messages it has received. Note that the supervisor cannot always have perfect knowledge of the location of a task; if a task is in flight, migrating from one node to another, the supervisor's task location record will indicate that the task is on both nodes, even though at any given point in time, the

task is actually only on one of the two nodes. While the supervisor cannot have perfect knowledge of task locations, the task tracking protocol guarantees that it is sufficient to record at most two locations.

Supervised task recovery is *pessimistic*. If the supervisor learns of a node failure, it will replicate all tasks it believes to be on the failed node. As a consequence the supervisor may replicate tasks that have not failed.

Duplicating Tasks. The pessimistic replication strategy may lead to multiple copies of a task. An in-flight supervised task may "survive" a node failure, provided it was stolen from the failed node in time. This has implications for side effects: Supervised tasks may only perform *idempotent* side effects i. e. side effects whose repetition cannot be observed.

Having multiple copies of the same task complicates tracking task locations. To simplify matters, the scheduler will track only the locations of the most recent copy of a task. To this end, each task is tagged with a *replication counter*, and task migration messages include a field with the replication counter value. The supervisor ignores all migration messages whose replication counter field is less than the current counter value stored by the supervisor. The supervisor increments the replication counter only when replicating a task.

Note that the fault tolerant scheduler does not kill obsolete copies of tasks. In fact, an obsolete copy may complete and write its result to its associated future ahead of the most recent copy. Thanks to idempotence, this scenario is indistinguishable from the one where the obsolete copy has died.

Simultaneous Failure. There are failure scenarios that result in simultaneous loss of physical connectivity between the root node and multiple other nodes. For example, network failures may lead to *partitioning* [11] where nodes in a partition may continue to communicate with each other, but no communication can occur between sites in different partitions. If a network is split in to two or more partitions, the valid partition is determined by which contains the root node.

However, when a network failure results in simultaneous connection losses, the underlying network abstraction layer nevertheless delivers failure event messages *sequentially*. Thus, the fault tolerant scheduler will have to react to a rapid sequence of individual node failures rather than to a simultaneous failure event.

5.2. Fault Tolerant Primitives

The fault tolerant primitives match the non-fault tolerant HdpH variants in Section 3.2, meaning that opting in to reliable scheduling is straightforward. The fault tolerant primitives explicitly create supervised futures, and implicitly create supervised tasks for the reliable scheduler to execute. The operations on a `SupervisedFuture` are the same as they are for a `Future` i.e. `get` and `probe`.

```
supervisedSpawn :: Dist → Task a → Par (SupervisedFuture a)
supervisedSpawnAt :: PE → Task a → Par (SupervisedFuture a)
```

The call `supervisedSpawn r task` behaves like the call `boundedSpawn r task` (Section 4.2), i. e. the bounded task `(task,r)` is stored in the caller's task pool, from where it may be stolen or selected for local execution. The reliable scheduler ensures that at least one copy of `task` is eventually evaluated.

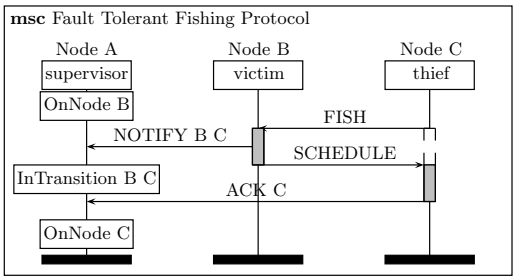


Figure 6: Supervised work-stealing in the absence of faults

The call `supervisedSpawnAt p task` behaves like the call `spawnAt p task`, i. e. `task` is placed on node `p` and eagerly executed. On detecting failure of `p`, the supervisor will replicate `task`, provided its associated future has not yet been filled, and eagerly execute `task` itself.

5.3. Fault Tolerant Work Stealing

5.3.1. Fault Tolerant Fishing Protocol

The fault tolerant scheduler provides task tracking (Section 5.1.1), and failure recovery (Section 5.1.3). These requirements necessitate additional runtime system messages for supervised work stealing. Before SCHEDULEing a task to the thief, the victim must inform the task's supervisor of the impending task migration, which it does with NOTIFY. This message informs the supervisor that the task is *in transit*, travelling between the victim and the thief. Should the failure of either the victim or thief be detected, the task is replicated. Once the thief receives the task, it informs the supervisor of the completed migration with an ACK message. Only if the thief is then detected to have failed is the task replicated, i.e. failure of previous victims no longer pose a threat to the task's existence.

The UML message sequence chart for the fault tolerant work stealing protocol is shown in Figure 6. The protocol involves the *supervisor* (the task creator), in an observational role, allowing it to track supervised task movements. Intuitively, there is a performance trade-off when using `supervisedSpawn` and `supervisedSpawnAt`. The gain is the reliability guarantee that the task will be completed in the presence of failure, provided that the node hosting the supervised future does not fail or become disconnected from the root node. The expense is a runtime performance overhead in the absence of faults, due to two additional messages (NOTIFY and ACK) transmitted for each task migration.

5.3.2. Guarding Against Migration Trace Race Conditions

Due to the asynchronous nature of communication channels in distributed systems task tracking messages may arrive out of order at the supervisor. Causal ordering [20] is only preserved on FIFO channels (i. e. TCP connections) between any two nodes. However, the network transport layer does not guarantee any causal ordering between three or more nodes. Hence, the fault tolerant fishing protocol must handle the possibility of task tracking messages "over taking" each other.

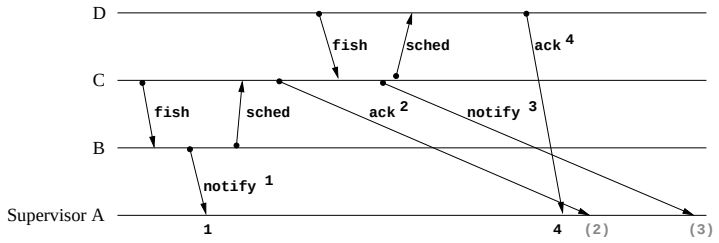


Figure 7: Supervisor Ignores Migration Message Overtaking

In order to cope with ACK messages "over taking" NOTIFY messages or vice versa, supervised tasks are equipped with an additional age counter. This counter is incremented each time a node sends a NOTIFY or an ACK message about the supervised task. The supervisor records only the location indicated by the most recent NOTIFY or ACK; it simply ignores task tracking messages whose age counter value is less than the value stored in the supervisor's task location record.

An example of message overtaking is shown in Figure 7. The task is supervised by node A, and has been stolen by node B. Node C fishes from B. Node B sends a NOTIFY to A, increments the age counter of the task, and sends the task in a SCHEDULE message to C. Node C sends an ACK to A and increments the age again, though due to network congestion, A does not receive this message until much later. In the meantime, D fishes from C. C sends a NOTIFY to A (also delayed in transmission), increments the age counter and SCHEDULEs the task to D. Node D sends an ACK message, along with the age tag, which is now 4. When node A receives task tracking messages of ages 2 and 3 from node C, they are simply ignored.

5.4. Fault Tolerant Skeletons

The `supervisedSpawn` and `supervisedSpawnAt` primitives guarantee the evaluation of a single supervised task. Higher-level abstractions built on top of these primitives guarantee the completion of a set of tasks. These abstractions hide lower level details by creating supervised tasks and futures dynamically behind the scenes.

There is a fault tolerant counterpart for each parallel skeleton in Section 4.4. Lazy work stealing skeletons are replaced with *supervised* lazy work stealing versions implemented using `supervisedSpawn`. Fault tolerant skeletons that use eager task placement are implemented with `supervisedSpawnAt`. As an example Figure 8 shows the implementation of the fault tolerant `parMapLocal`, in contrast to the non-fault tolerant version in Figure 5.

Divide and Conquer is a more elaborate, recursive parallel pattern, and a `parDivideAndConquer` skeleton is included in the fault tolerant and non-fault tolerant HdpH APIs. It is a parallel skeleton that allows a problem to be decomposed into sub-problems until they are sufficiently small, and then reassembled with a combining function.

In the fault tolerant `parDivideAndConquer` skeleton shown in Figure 9 supervisors are hierarchically nested. Decomposing a large problem into many smaller problems in HdpH will saturate the distributed environment with a tree of supervised tasks. That

```
parMapLocal  -- fault tolerant, bounded work stealing parallel map skeleton
  :: Dist              -- bounding radius
  → Closure (a → b)    -- function closure
  → [Closure a]        -- input list
  → Par [Closure b]    -- output list
parMapLocal r c_f cs = mapM apply_c_f cs >>= mapM get
  where
    apply_c_f c = supervisedSpawn r $(mkClosure [|eval $ toClosure (unClosure c_f $ unClosure c)|])
```

Figure 8: Fault Tolerant Topology Aware `parMap` Implementation

```
parDivideAndConquer
 :: Closure (Closure a → Dist)                      -- problem radius
 → Closure (Closure a → Par (Closure b))            -- seq solver
 → Closure (Closure a → [Closure a])                -- decompose
 → Closure (Closure a → [Closure b] → Closure b)    -- combine
 → Closure a                                        -- problem
 → Par (Closure b)                                  -- output
```

Figure 9: Fault Tolerant Topology Aware `parDivideAndConquer` API

is, a node may evaluate a supervised task that spawns several subtasks, to be supervised by their parent task.

The advantage of divide and conquer style parallelism is twofold. First, the skeleton recursively decomposes large tasks into smaller ones so that granularity is sufficiently small to fully utilise all compute nodes. Second, it removes a bottleneck in the supervised work stealing protocol (Section 5.3). Multiple supervisors means that there is no one node receiving all NOTIFY and ACK messages for all supervised tasks. One drawback of this approach is substantial recovery costs when a supervisor with many children is lost. One approach to minimise the recovery costs is to apply memoization techniques to computed functions to avoid re-evaluating all child tasks once again, as demonstrated in [35].

Note how the signature of `parMapLocal` in figure 8 matches exactly that of the corresponding non-fault tolerant skeleton in Figure 5; the same is true for `parDivideAndConquer`, though we do not show the non-fault tolerant implementation due to lack of space. Thus the programmer can enable or disable fault tolerance with minimal effort, by simply switching skeletons.

So far, the fault tolerance approach of SGP2 focuses on the replication of computations lost due to failure, rather than on the replication of distributed state. However, some symbolic applications, such as the orbit calculation [23], require large distributed data structures. SGP2 plans to support this class of applications by offering distributed data structures, like for instance *distributed hash tables*, with a restricted interface, e. g. no deletions. In this case, fault tolerance will be achieved by replicating distributed state and keeping the replicas in sync.

6. Initial Evaluation

This section outlines the SGP2 implementation status before demonstrating the capabilities of the current implementation snapshot by presenting some scalability and fault

tolerance measurements.

6.1. Implementation Status

The HdpH DSL outlined in Section 3.2 is implemented both for Beowulf clusters and HPC platforms, and is available online [15]. The cluster implementation uses TCP communication and a full set of Unix utilities, and is demonstrated in Section 6.3. The HPC implementation is more challenging as it must use MPI [28] for communication and a restricted set of Unix utilities, e.g. no sockets. Section 6.2 demonstrates HdpH scalability on HECToR, at present the UK's largest HPC with approximately 90 000 cores.

HdpH implements most of the generic SGP CAG skeletons including those in Figures 5, 8 and 9, and others like task-farms. Not all of the locality control and fault tolerance features described in Sections 4 and 5 are fully realised. So far, HdpH has limited locality control: it can distinguish between PEs on the same multicore node and on other nodes. Only explicit placement variants of the fault tolerant skeletons outlined in Section 5 have been implemented so far.

We are currently completing the HdpH implementation, integrating locality control and fault tolerance with work stealing. We are also developing the key components required to complete the SGP2 implementation, namely (1) a high-performance link (with overheads much lower than SCSCP) between the HdpH coordination server and GAP CAS servers, (2) HdpH skeletons specific to the symbolic computation domain, and (3) GAP bindings to the HdpH skeletons to implement the CAG interface.

6.2. Scalability

The scalability of HdpH has been investigated on HECToR with the *SumEuler* symbolic benchmark that sums Euler's totient function ϕ over long lists of integers. SumEuler is a moderately irregular data parallel problem where the irregularity stems from computing ϕ on smaller or larger numbers.

Strong scaling experiments. We investigate how SumEuler over a fixed list of 96000 integers performs on two different platforms, a commodity Beowulf cluster (up to 30 nodes, 240 cores) and HECToR (up to 8 nodes, 256 cores). In this microbenchmark, the actual computation of Euler's ϕ is performed by a naive algorithm coded in Haskell. The sequential runtimes of this problem are 941.7 seconds on the Beowulf and 780.2 seconds on HECToR.[2]

Figure 10 plots, separately for each architecture and on a logarithmic scale, the runtimes of two topology-aware skeletons, `parMap2Level` and `parMap2LevelRelaxed`, and of two unaware skeletons, one doing plain work stealing, the other plain round-robin placement. On both architectures, plain work stealing has the highest runtime and the highest variance. On the Beowulf, plain work stealing performs dramatically worse than the other skeletons, yet plain round-robin scheduling initially appears to perform almost as well as the topology-aware skeletons. On HECToR, the performance of plain work stealing isn't as far off as on the Beowulf, probably due to its low-latency interconnect. Moreover, on HECToR the topology-aware skeletons clearly outperform plain round-robin placement.

[2]Reported runtimes are averages of at least 5 runs.

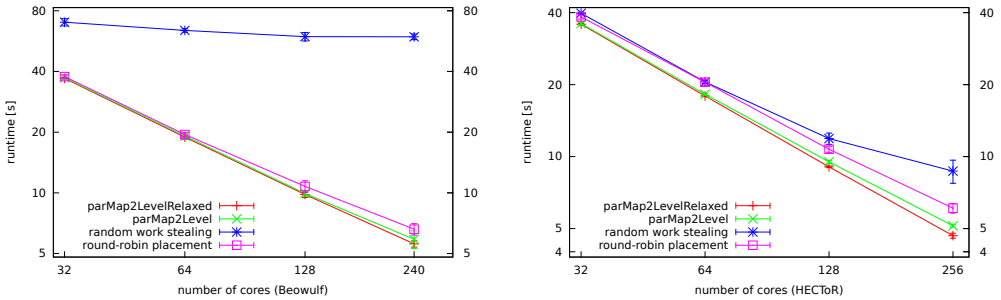


Figure 10: SumEuler Runtime (log scale) of Topology-aware and -unaware Skeletons.

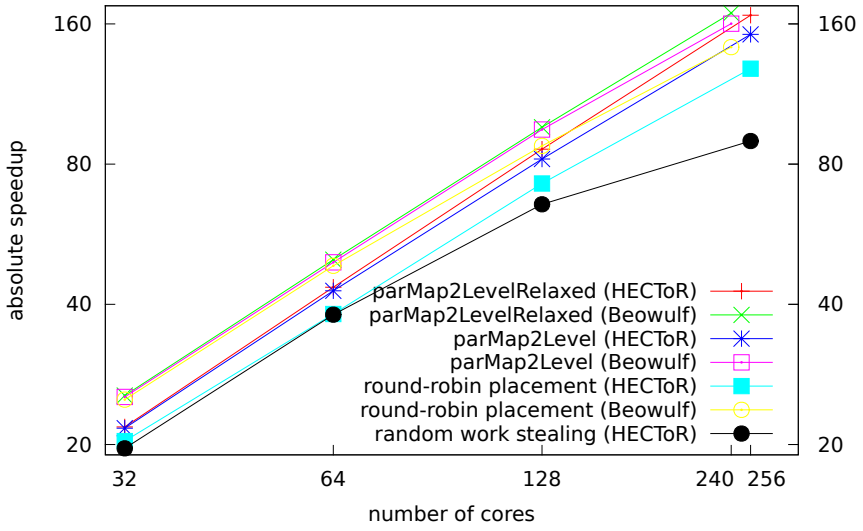


Figure 11: SumEuler Speedup (log scale) of Topology-aware and -unaware Skeletons.

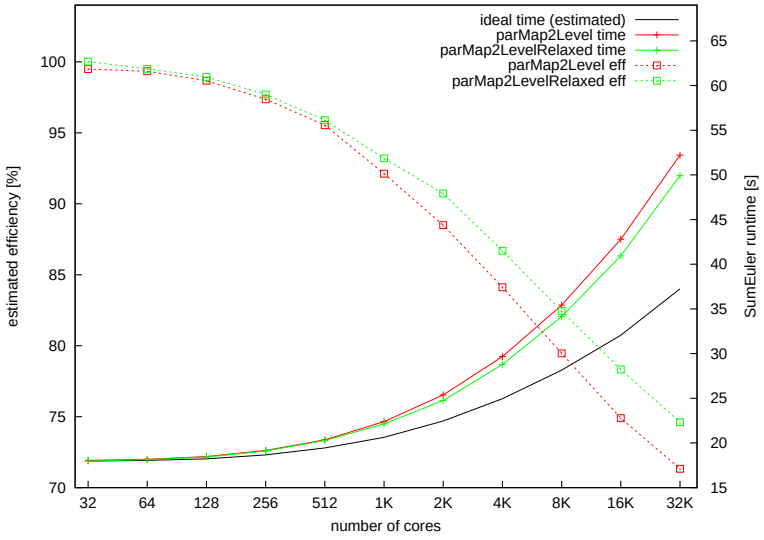


Figure 12: SumEuler — Weak Scaling of 2 Two-level Skeletons to 32K Cores

Figure 11 compares the speedups (on a logarithmic scale) of all skeletons on both platforms. Unsurprisingly, random work stealing performs worst on both platforms,[3] followed by round-robin placement. The `parMap2LevelRelaxed` skeleton wins the contest on both platforms, eventually reaching absolute speedups of 169 on the Beowulf and 167 on HECToR, This corresponds to an efficiency of 70% on the Beowulf and 65% on HECToR.

Weak scaling experiment. Figure 12 reports weak scaling of two variants of the SumEuler benchmark from 1 to 1024 nodes (i. e. 32 to 32K cores) on HECToR. The two variants differ only in their use of the underlying skeleton: One uses `parMap2Level`, the other `parMap2LevelRelaxed`. Both benchmarks rely on GAP to compute Euler's ϕ, SGP2 performs only coordination (and the final summation).

The benchmarks start out with an input interval of 6.25 million integers on 32 cores, doubling the size of the interval when doubling the number of cores; on 32K cores the interval is 6.4 billion integers long. Doubling the size of the input interval increases the amount of work by more than a factor of 2; by sampling the sequential runtime of small subintervals, we estimate a runtime curve for ideal scaling. The runtime graphs in Figure 12 show that the two benchmarks do not scale perfectly. However, even on 32K cores their runtimes are still within a factor of 1.5 times of the ideal.

The efficiency graphs (computed with respect to estimated ideal scaling) show that efficiency is steadily declining, yet remains above 70% even on 32K cores. These graphs also

[3]The speedup curve for random work stealing on the Beowulf is not shown as it remains below 20.

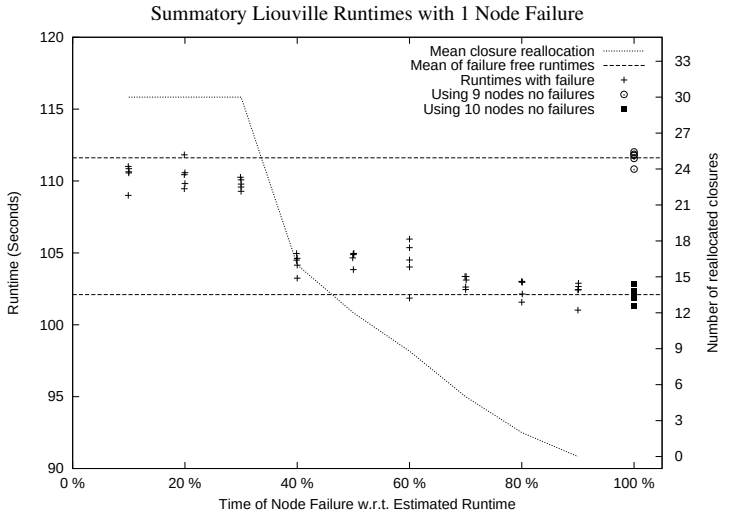


Figure 13: Summatory Liouville — Recovery Times after a Node Failure

show that the skeleton `parMap2LevelRelaxed` offers a small efficiency advantage (of 3 to 4 percentage points) over `parMap2Level`. This is likely because `parMap2LevelRelaxed` is coping better with the irregularity of the benchmark in the final stages of the computation, as it can utilise nodes that have run out of work early.

An earlier investigation of the impact of topology awareness on the performance of work stealing [3] is outlined in Section 4.5. Some key performance results are as follows. Topology awareness reduces variability dramatically: the topology information helps to avoid making bad scheduling decisions. For a small set of kernel benchmarks topology awareness increases speedup, in some cases by an order of magnitude, even on small architectures, e.g. a Beowulf cluster of multicore nodes [3].

6.3. Fault Tolerance

We have previously reported the performance, and failure-recovery overheads, of an early fault tolerant version of HdpH [33]. We investigated both divide-and-conquer and data-parallel fault tolerant skeletons that use eager random and eager round robin scheduling respectively. The results show runtime overheads of between 2.5% and 7% using a eager map skeleton in the absence of faults. A thorough evaluation of the fault tolerant scheduler in SymGridPar2 will appear in [32].

We illustrate the fault tolerance mechanisms here by considering the Summatory Liouville task-parallel Computational Algebra problem [5]. It is parallelised with a variant of the `parMap` skeleton that uses eager round robin task placement. The Liouville function $\lambda(n)$ is the completely multiplicative function defined by $\lambda(p) = -1$ for each prime p. $L(n)$ denotes the sum of the values of the Liouville function $\lambda(n)$ up to n, where $L(n) := \sum_{k=1}^{n} \lambda(k)$.

The computation measured is $L(3 \cdot 10^8)$ with a chunk size of 10^6, which is initially deployed on 10 nodes, generating 300 closures in total and distributing 30 closure to each node.

The results in Figure 13 show the runtime of computing $L(3 \cdot 10^8)$ when a single node failure occurs at approximately 10%,20%..90% of expected failure-free execution time. Each failure experiment was repeated 5 times, shown as 5 dots in the graph. The figure also shows two horizontal lines indicating the failure-free runtimes (averaged over 5 runs) on 9 and 10 nodes, respectively. Lastly, the figure also shows the mean number of closures that were reallocated to compensate the node failure.

Up to 30% of expected total runtime, all 30 closures need to be reallocated. However, at 40%, 14 closure have already been evaluated, and only the 16 remaining closures need to be reallocated. The number of reallocated closures continues to fall and reaches zero at 90% of expected total runtime.

We see that when a node fails early on, i.e. in the first 30% of estimated total runtime, the performance of the remaining 9 nodes is comparable to that of a failure-free run on 9 nodes. Moreover, node failure occurring near the end of a run, e.g. at 90% of estimated runtime, does not impact runtime performance, i.e. matches that of a 10 node cluster that experiences no failures at all.

Based on our earlier work, and the example above we are hopeful that the overheads of HdpH fault tolerance in the absence of failures are low, i.e. under 7% in the 2 programs measured [33], and the cost of recovery is negligible.

7. Conclusion

We have presented the design and initial evaluation of SymGridPar2 (SGP2), a framework designed to execute symbolic computations on large (10^5 core) architectures. We have outlined the SGP2 design goals, principles and architecture, including the key decision to coordinate the parallel computations in the HdpH domain specific language (Section 3). We have described how scalability is achieved using layering and by allowing the programmer to control task placement (Section 4). We have outlined how fault tolerance is provided by supervising remote computations, and shown how higher-level fault tolerance abstractions can be constructed (Section 5). We have outlined the current implementation and report encouraging preliminary scalability and fault tolerance results on architectures with up to 32,000 cores (Section 6).

The implementation of SGP2 is ongoing, and Section 6.1 discusses our plans to complete the CAG interface to GAP, to better integrate the fault tolerance and work distribution, and to improve locality control. Larger scale evaluations that approach our design aim of 10^5 cores will be possible as HPC platforms grow and become more affordable. We are simultaneously developing an HdpH profiler to aid programmers. Alongside the implementation effort we also plan to investigate SGP2's effectiveness on challenge symbolic computations. One such challenge application, to be developed within HPC-GAP, will involve solving large "standard base" problems that arise in representation theory.

8. Acknowledgements

This work is supported by the EPSRC HPC-GAP (EP/G05553X), EU FP6 SCIEnce (RII3-CT-2005-026133) and EU FP7 RELEASE (IST-2011-287510) grants.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008, Seattle, WA, USA*, pages 63–74. ACM, 2008.

[2] A. Al Zain, K. Hammond, J. Berthold, P. W. Trinder, G. Michaelson, and M. Aswad. Low-pain, high-gain multicore programming in Haskell: coordinating irregular symbolic computations on multicore architectures. In *DAMP 2009, Savannah, GA, USA*, pages 25–36. ACM Press, 2009.

[3] M. Aswad, P. W. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow Parallel Haskell (GPH). *Procedia CS*, 9:1807–1816, 2012.

[4] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, Ieee*, 23(2):22–28, 2003.

[5] P. B. Borwein, R. Ferguson, and M. J. Mossinghoff. Sign changes in sums of the Liouville function. *Math. Comput.*, 77(263):1681–1694, 2008.

[6] B. W. Char et al. *Maple V Language Reference Manual*. Maple Publishing, Waterloo Canada, 1991.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[8] G. Cooperman. Parallel GAP: mature interactive parallel computing. In *Groups and computation, III, Columbus, OH, 1999*, pages 123–138. De Gruyter, 2001.

[9] D. Coutts, N. Wu, and E. de Vries. Haskell library: `network-transport` package. A network abstraction layer API. `http://hackage.haskell.org/package/network-transport`.

[10] M. Daberkow, C. Fieker, J. Klüners, M. Pohst, K. Roegner, M. Schörnig, and K. Wildanger. Kant v4. *J. Symb. Comput.*, 24(3/4):267–283, 1997.

[11] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Survey*, 17(3):341–370, 1985.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] GAP Group. GAP – Groups, Algorithms, and Programming, 2007. `http://www.gap-system.org`.

[14] W. Gropp and E. Lusk. Fault tolerance in MPI programs. *Journal High Performance Computing Applications*, 18:363–372, 2002.

[15] Haskell distributed parallel Haskell. Repository `http://hackage.haskell.org/package/hdph`.

[16] V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform Grids. In *CCGrid 2010, Melbourne, Australia*, pages 123–134. IEEE, 2010.

[17] R. H. H. Jr. Multilisp: A Language for Concurrent Symbolic Computation. *Journal: ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[18] W. Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In *AAECC-8, Tokyo, Japan*, LNCS 508, pages 341–353. Springer, 1991.

[19] C. Lameter. An overview of non-uniform memory access. *Commun. ACM*, 56(9):59–54, 2013.

[20] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[21] S. Linton, K. Hammond, A. Konovalov, C. Brown, P. Trinder., and H.-W. Loidl. Easy composition of symbolic computation software using SCSCP: A new lingua franca for symbolic computation. *Journal of Symbolic Computation*, 49:95–119, 2013. To appear.

[22] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[23] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math.*, 10:197–205, 2001.

[24] P. Maier, R. J. Stewart, and P. W. Trinder. Reliable scalable symbolic computation: the design of SymGridPar2. In *SAC 2013, Coimbra, Portugal*, pages 1674–1681. ACM, 2013.

[25] P. Maier and P. Trinder. Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL 2011, Lawrence, Kansas, USA*, LNCS 7257, pages 35–50. Springer, 2012.

[26] S. Marlow, R. Newton, and S. L. Peyton-Jones. A monad for deterministic parallelism. In *Haskell 2011, Tokyo, Japan*, pages 71–82. ACM Press, 2011.

[27] M. M. Maza and S. M. Watt, editors. *PASCO '07: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, London, Ontario, Canada*. ACM Press, 2007.

[28] MPI-Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Application*, 8(3–4):165–414, 1994.

[29] The OpenMath Standard, Version 2.0, 2012. `http://www.openmath.org/`.

[30] K. P. Saravanan, P. M. Carpenter, and A. Ramirez. Power/performance evaluation of energy

efficient ethernet (eee) for high performance computing. In *ISPASS 2013, Austin, TX, USA*, pages 205–214. IEEE, 2013.

[31] M. Schneider. Self-Stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[32] R. Stewart. *Reliable Massive Parallel Computing: Fault Tolerance for a Distributed Haskell*. PhD thesis, Mathematical and Computer Sciences, Heriot Watt University, 2013.

[33] R. Stewart, P. Trinder, and P. Maier. Supervised Workpools for Reliable Parallel Computing. In *TFP 2012, St Andrews, UK*. Springer, 2013.

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

[35] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Fault-Tolerant Scheduling of Fine-Grained Tasks in Grid Environments. *International Journal of High Performance Applications*, 20(1):103–114, 2006.

[36] A. A. Zain, P. W. Trinder, and K. Hammond. Orchestrating computational algebra components into a high-performance parallel system. *Int. J. of High Performance Computing and Networking*, 7(2):76–86, 2012.

[37] R. E. Zippel, editor. *Computer Algebra and Parallelism*, LNCS 584. Springer, 1992.