Reliable Scalable Symbolic Computation: The Design of SymGridPar2[*]

Patrick Maier
Heriot-Watt University, Edinburgh, UK
P.Maier@hw.ac.uk

Rob Stewart
Heriot-Watt University, Edinburgh, UK
R.Stewart@hw.ac.uk

Phil Trinder
Heriot-Watt University, Edinburgh, UK
P.W.Trinder@hw.ac.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 10^6 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 on approximately 2000 cores, and investigate the overheads of tolerating faults for simple symbolic computations.

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 vary by several orders of magnitude. The SCIEnce project developed SymGridPar [15] as a standard framework for executing symbolic computations on small-scale parallel architectures (Section 2). SymGridPar uses OpenMath [22] 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 10^6 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, and sketch higher-level fault tolerance abstractions like supervised workpools and 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).

[*]ACM, 2013. This is the authors version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in SAC'13.

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 [9], or GAP [10]. 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 [15]. 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 [22]. SCSCP has become a *de facto* standard, with implementations for 9 CAS and libraries for several languages including Java, C++, and Haskell [15].

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. [20, 14, 24]. 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 exploited the general-purpose skeleton-based **SymGrid-Par** 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 [1]. 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.

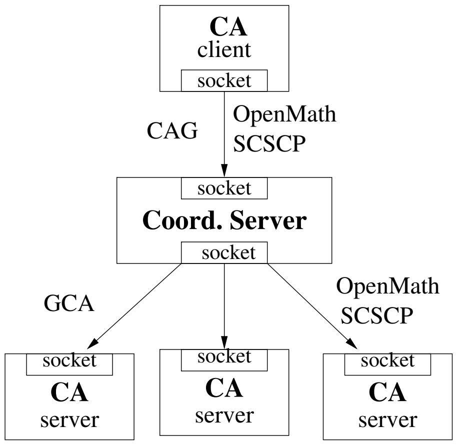


Figure 1: SymGridPar and SymGridPar2 Architecture

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 [15] 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]. Some 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, which is another SCSCP-compliant computer algebra system. Currently the Coordination Server is implemented in Eden [16], 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 following Moore's law, i.e. growing exponentially. Many expect 100,000 core platforms to become commonplace. Hence parallel systems must be designed for far greater scale than previously. Moreover, the best predictions are that core failures on such an architecture will become relatively common, perhaps one hour mean time between core failures. So parallel systems need to be both scalable and fault tolerant.

SymGridPar was never designed to scale to thousands of cores, let alone millions. Neither does its load management 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

The main goal in developing SymGridPar2 (SGP2) as a successor to SymGridPar is *scaling symbolic computation to architectures with* 10^6 *cores*. As argued above, this scale necessitates two further design goals: *topology awareness* and *fault tolerance*, to cope with increasingly non-uniform communication topologies and increasingly frequent component failures, respectively. Finally, the SGP2 design aims to 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) [18], designed to deliver scalable fault tolerant symbolic computation. HdpH is a shallowly embedded parallel extension of Haskell that supports high-level semi-explicit parallelism. To aid portability and maintainability, HdpH itself possesses a modular, layered architecture, and is implemented in Concurrent Haskell (with GHC extensions).

```
-- task distribution in the Par monad
type Par a       -- Par monad computation returning type 'a'
type Closure a   -- serialisable closure of type 'a'

pushTo :: PE -> Closure (Par ()) -> Par () -- eager explicit
spark  ::       Closure (Par ()) -> Par () -- lazy implicit

-- communication via IVars
type IVar a      -- write-once buffer of type 'a'
type GIVar a     -- global handle to an 'IVar a'

new   :: Par (IVar a)                          -- creation
glob  :: IVar a -> Par (GIVar a)               -- globalisation
rput  :: GIVar (Closure a) -> Closure a -> Par () -- remote write
probe :: IVar a -> Par Bool                    -- local test
get   :: IVar a -> Par a                       -- local read
```

Figure 2: HdpH primitives.

HdpH extends the `Par` monad DSL [19] for shared-memory parallelism to distributed memory. Figure 2 lists the HdpH primitives. There are two modes of task distribution, both taking a task of type `Par ()` wrapped as a serialisable `Closure`. The `pushTo` primitive eagerly places the task on a named PE, where it is immediately executed. In contrast the `spark` primitive places the task into a local spark pool, from where it may be stolen by any PE looking for work. That is, `spark` provides on-demand (lazy) implicit task placement via distributed work stealing.

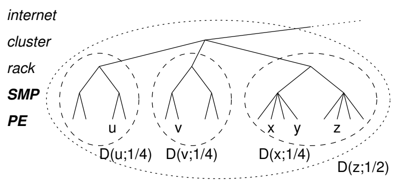
Tasks synchronise and communicate by write-once buffers called `IVar`. An IVar is created by `new`, and globalised by `glob`, resulting in a `GIVar`, a global handle to the IVar. Via the global handle, `rput` can transparently write a serialisable closure to a remote IVar. An IVar can be read by `get`, which will block until a value is available, and tested by `probe`, which will indicate whether `get` would block; note that `probe` and `get` operate locally — they can only test and read IVars on the node they were created on.

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 t` and returns its value of type `t`; `toClosure` wraps a value of any serialisable type `t` into a `Closure t`. Additionally, the Template Haskell construct `$(mkClosure [| e |])` constructs a `Closure t` wrapping the unevaluated thunk `e` of type `t`; 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*, 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.

4. SGP2 LOCALITY CONTROL

Historically many parallel architectures have had a flat communi-



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

cation topology: the communication latency between any pair of processors is approximately the same. Parallel programming models like MPI [21] exploit this simplified model.

As the number of cores grows however, we find that large scale architectures necessarily have a hierarchical communication topology. For example in a typical cluster a core in an shared-memory node can communicate more quickly with a core in the same node than with a core in a remote shared-memory node. To the programmer this manifests itself in differences in latency as messages to far-away processors have to hop across more switches and routers. Due to network contention, effective bandwidth typically decreases with increasing latency.

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 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.

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

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

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 $\mathit{range}\, d = \{2^{-n} \mid n = 0, 1, \dots\} \cup \{0\}$.

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 (due to 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 $D(p; r) = D(q; r)$.

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

(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.

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 `spark` primitive:

```
spark :: Dist -> Closure (Par ()) -> Par ()
```

The radius r constrains how far a task can travel from the sparking 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 `spark r task`, it adds the pair `(task,r)` to its *spark pool* data structure. We call the pair `(task,r)` a *bounded spark* (with radius `r`).

When p_0 runs out of work, and its own spark pool is non-empty, it uses the following *local spark selection policy*: Pick the youngest of the sparks with minimal radius and schedule it for execution. Thus, p_0 prioritises sparks with small radius for local scheduling. If, on the other hand, p_0 runs out of work with its own spark pool empty then it will send a message requesting work to a random PE.[2]

When p_0 receives a request for work from another PE p, it tries to find a suitable spark using the following *remote spark selection policy*: Pick a spark with minimal radius from the set of sparks

[2]Actually, p_0 does not wait for the spark pool to drain completely; to hide latency p_0 will send a request for work already when the pool hits a minimum number, the so-called *low water mark*.

whose radius is greater or equal to $d(p_0, p)$; if there are several such sparks, pick the oldest one. Thus for remote scheduling, p_0 prioritises sparks whose radii match the distance to the PE requesting work. If this policy does not yield a suitable spark then p_0 forwards p's request for work to a random PE. If, however, the remote spark selection policy does yield a spark `(task,r)` then p_0 sends the spark to the requesting PE p, which will put it into its own spark pool, from where it will either be scheduled for local execution, or sent to yet another PE requesting work. Note that due to property (B1), $D(p_0; \texttt{r}) = D(p; \texttt{r})$, i. e., both p_0 and p are centers of the same ball of PEs eligible to execute the spark `(task,r)`.

To prioritise local stealing, the work search algorithm is not uniformly random. When p_0 initiates a request for work, it will send its request to 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 $d(p_0, p)$. 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 work, requests expire after being forwarded a number of times. In this case the requesting PE backs off for some time before repeating the request.

The task selection policies can be summed up as: Sparks with small radii are preferred for local execution, and the bigger the radius, the further a spark should travel. This design is consistent with the findings of [13], which investigated scheduling strategies based on granularity information (which loosely corresponds to radii) and found that the optimal strategy schedules small tasks locally and large tasks remotely.

Note that the bounded `spark` 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 RTS's work stealing algorithm. The spark radii only allow the programmer to constrain the RTS'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 tasked will be scheduled for execution immediately, taking priority over any stolen tasks.

HdpH already supports eager work placement via `pushTo`. In order to support topology aware placement, HdpH has to 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 $D(p_0; r)$.

```
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 spawn cs >>= mapM get where
  spawn c = do
    v <- new
    gv <- glob v
    spark r $(mkClosure
                [|rput gv $
                  toClosure (unClosure c_f $ unClosure c)|])
    return v

parMap2Level  -- two-level bounded parallel map skeleton
  :: 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
  vs <- mapM spawn qcs
  concat <$> mapM ( v -> unClosure <$> get v) vs
    where spawn (q,cs_q) = do
            v <- new
            gv <- glob
            pushTo q $(mkClosure
                        [|parMapLocal (r/2) c_f cs_q >>=
                          rput gv . toClosure|])
            return v
```

Figure 4: Topology Aware Algorithmic Skeletons

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.

Property (B2) guarantees that equidistant bases exist. However, due to (B1), these bases are not unique. 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.

4.4 Topology aware Skeletons

HdpH provides a library of *topology aware algorithmic skeletons* that abstract over the topology aware primitives. For example Figure 4 shows two versions of a parallel map over a list. Both 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 measured in Section 6.2.

`parMapLocal` creates sparks 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 in a quick distribution of *big* tasks to PEs far from the caller, and these PEs then act as local coordinators by sparking *small* tasks to be evaluated in their vicinity. Thanks to bounded sparks and equidistance of the coordinators, it is guaranteed that the small tasks sparked by one local coordinator stay in its vicinity; in particular, they cannot travel to a PE in the vicinity of another local coordinator.

We stress that both of the above skeletons implement a semi-explicit interface in that both 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.

5. SGP2 FAULT TOLERANCE

Fault tolerance is a means to unlock SymGridPar2 scalability ambitions for reliable long-running computations on massively parallel systems.

Most existing fault tolerant approaches in distributed architectures follow a rollback-recovery approach, often involving checkpointing and synchronisation phases. New opportunities are being explored as alternative and more scalable possibilities, and both language and non-language based techniques have been proposed. At the highest level, fault oblivious and self stabilising algorithms have been developed [5] 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 necessarily 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 [11], and 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 mechanisms in SGP2 exploit the loosely coupled design of HdpH, which separates remote tasks and values. They are integrated with the topology aware task placement mechanisms described in Section 4.

5.1 Fault Tolerant Primitives

Key to SGP2 fault tolerance is the *supervision* and *replication* of remote tasks in HdpH: The PE spawning a supervised task monitors the PE executing task, and replicates task on another PE should it find that the monitored PE has failed before completing task. Note that this has implications for side effects: Supervised tasks may only perform idempotent side effects, i. e., side effects whose repetition cannot be observed.

At the lowest level, HdpH will expose *supervised* variants of the task distribution primitives spark and pushTo that guarantee completion of a single task in the presence of failures.

```
supervisedSpark  :: Par Bool -> Dist -> Closure (Par ()) -> Par ()
supervisedPushTo :: Par Bool -> [PE] -> Closure (Par ()) -> Par ()
```

Both primitives take an additional argument, an action probing the IVar that is to receive the result of the supervised task, which they perform to determine whether the task has completed.

```
parMapLocal
 :: Dist                 -- bounding radius
 -> Closure (a -> b)     -- function closure
 -> [Closure a]          -- input list
 -> Par [Closure b]      -- output list

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 5: Fault Tolerant Topology Aware Skeletons

The call supervisedSpark done r task behaves like the call spark r task (Section 4.2). However, behind the scenes, the sparking PE monitors whichever PE has stolen task, and re-sparks task should that PE fail. Eventually, the re-sparked task will be stolen by another PE, and the monitoring starts afresh.

The call supervisedPushTo done ps task behaves similar to pushTo (Section 3.2) except that supervisedPushTo takes a list of target PEs. First, task is pushed to the head of the list ps for execution. If the pushing PE finds the executing PE failed, it will re-push task to the next PE on the list, and so on, until either task is completed, or the list ps is exhausted; in the latter case the pushing PE will execute task itself.

5.2 Fault Tolerant Workpools and Skeletons

The above fault tolerant primitives guarantee the completion of a single task. Higher-level abstractions, built on top of the primitives, guarantee the completion of a set of tasks.

Supervised Workpools.

The simplest abstraction is a supervised workpool, which takes as input a list of tasks and a list of actions probing the IVars associated with the input tasks. The supervised workpool guarantees the completion of all input tasks, provided the PE hosting the workpool does not fail. Thus the supervised workpool reduces a distributed architecture to one with a single point of failure.

The above fault tolerant primitives are used to implement several flavours of supervised workpools, depending on whether tasks are placed explicitly (via supervisedPushTo), implicitly (via supervisedSpark), or according to some combination thereof.

Fault Tolerant Algorithmic Skeletons.

Supervised workpools are still relatively low-level since they require the programmer to explicitly create all tasks and IVars beforehand. Higher-level abstractions, like *fault tolerant algorithmic skeletons*, hide all these details by creating tasks and IVars dynamically. Figure 5 shows the signatures of two fault tolerant skeletons, a parallel map and a parallel divide and conquer. Under the hood, these skeletons use supervised workpools or the above fault tolerant primitives to integrate remote task supervision and replication with topology aware task stealing.

Note how the signature of parMapLocal matches exactly that of the corresponding non-fault tolerant skeletons in Figure 4; the same is true for parDivideAndConquer, though we do not show the non fault tolerant implementation due to lack of space. Thus, depending on whether the architecture requires it and whether its overheads are bearable, 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 [17], 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. We expect that the restricted data structure interface will give rise to simple and efficient algorithms for keeping 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 [12]. 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 [21] 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 4 and 5, 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 workpools and 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 an irregular data parallel problem where the irregularity stems from computing ϕ on smaller or larger numbers. The HdpH implementation uses a layered two-level skeleton combining eager explicit task placement with lazy work stealing very similar to the `parMap2Level` skeleton described in Section 4.4. The main difference being that it does not control locality on the inner work stealing layer.

Figure 6 reports the runtime and speedup of the SumEuler benchmark on two input lists with 160k and 240k elements respectively. The benchmark demonstrates strong scaling from 1 to 64 nodes (i. e., 32 to 2048 cores) on HECToR. Note that the absolute speedup is extrapolated because the problem is too big to be run sequentially; the extrapolation is based on an estimated absolute speedup of 21 on 32 cores, an estimate that was confirmed on smaller inputs. Efficiency ranges from 66% on 32 cores to about 63% on 512 cores, but drops on 2048 cores to 55% and 45% for the larger and smaller inputs, respectively.

The benchmark demonstrates HdpH scaling well to 2048 cores,

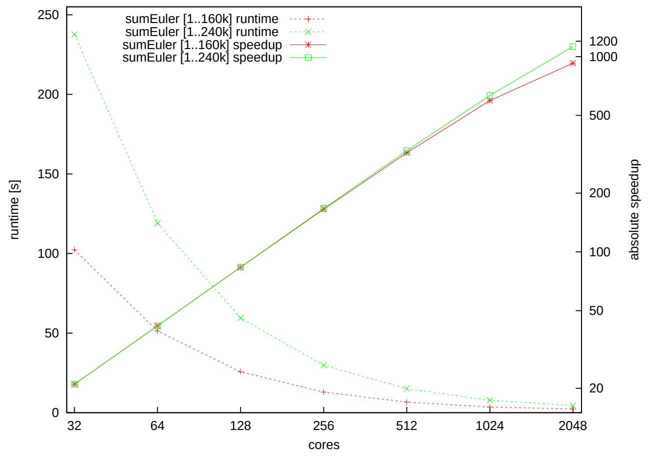


Figure 6: Strong Scaling of a Two-level Skeleton to 2048 Cores

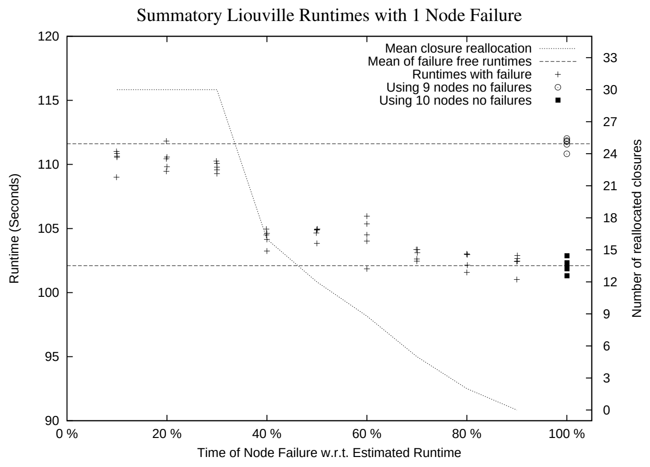


Figure 7: Workpool Recovery Times after a Node Failure

and very well to about 512 cores. Beyond 512 cores, the speedup curves deviate slightly from their linear progression. We believe this is due to the problem size being too small, with runtimes falling below 5 seconds, rather than an inherent limit on the scalability of HdpH. The obvious next step is to investigate weak scaling of HdpH for larger data sets, and on larger HECToR configurations.

The impact of topology awareness on the performance of work stealing has also been investigated in [3], demonstrating that topology awareness reduces variability and increases speedup, in some cases by an order of magnitude, already on small Beowulf clusters.

6.3 Fault Tolerance

Performance and failure-recovery overheads have been reported for divide-and-conquer and task-parallel fault tolerant skeletons [23]. The example used here is Summatory Liouville, a task-parallel Computational Algebra problem [4]. 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, and distributing 30 to each node.

The results in Figure 7 show the runtime of computing $L(3 \cdot 10^8)$ when node failure occurs at approximately 10%,20%..90% of expected execution time, and 5 runtimes are observed at each timing point. The mean number of closures that are reallocated relative to when node failure occurs is also shown. Lastly, it shows 5 runtimes using 10 nodes when *no* failures occur, and additionally 5 runtimes using 9 nodes, again with no failures.

Fully evaluated closure values are first seen at 40% of estimated total runtime, where only 16 (of 30) are reallocated. This continues to fall until 90% of the predicted runtime, when 0 closures are reallocated, indicating that all closures had already been fully evaluated on the responsible node, prior to failing.

When a node dies early on, i.e. in the first 30% of estimated total runtime, the performance of the remaining 9 nodes is comparable with 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.

7. CONCLUSION

We have presented the design and initial evaluation of SymGrid-Par2 (SGP2), a framework for executing symbolic computations on large (10^6 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 scalability and fault tolerance results on architectures with up to 2000 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. 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.

9. REFERENCES

[1] A. Al Zain, J. Berthold, K. Hammond, and P. W. Trinder. Orchestrating production computer algebra components into portable parallel programs. In *2008 Open Source Grid and Cluster Conference, Oakland, CA, USA*, pages 257–266, 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] P. B. Borwein, R. Ferguson, and M. J. Mossinghoff. Sign changes in sums of the Liouville function. *Math. Comput.*, 77(263):1681–1694, 2008.

[5] F. Cappello, A. Geist, B. Gropp, L. V. Kalé, B. Kramer, and M. Snir. Toward exascale resilience. *High Performance Computing Applications*, 23(4):374–388, 2009.

[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] 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.

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

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

[12] Haskell distributed parallel Haskell. Repository `https://github.com/PatrickMaier/HdpH`.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] R. Stewart, P. Trinder, and P. Maier. Supervised workpools for reliable parallel computing. In *TFP 2012, St Andrews, UK*. Springer, 2013. To appear.

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