Reliable Scalable Symbolic Computation: The Design of SymGridPar2

Patrick Maier    Rob Stewart    Phil Trinder

School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Scottish Programming Languages Seminar
St Andrews, 22 April 2013
Introduction — Why Do We Need SymGridPar?

Context: HPC-GAP aims to parallelise GAP for HPC platforms with $10^5$ cores.
- GAP is a computer algebra system for computational group theory.
  - 25+ years old
  - 10m+ lines of code (including libraries and tables)

Computer Algebra differs from traditional HPC problem domains.
- Lots of parallelism
  - BUT very irregular task sizes, often dynamic task creation.
- Symbolic computation keeps integer ALU busy
  - BUT no floating point operations, may require lots of memory.

“Solution”: SymGridPar middleware (~ 2007)
- Orchestrates parallel computation on black-box GAP servers across network.
- Distributes irregular tasks by on-demand random work stealing.
- Exposes general-purpose and domain-specific skeletons to GAP programmer.
SymGridPar was developed for this:

### Challenges for SymGridPar2 (SGP2)

- **Rate of node/network failures increases with cluster size.**
  - Requirement: **fault tolerance**
  - The failure rate of SGP2 computations on a large cluster should be comparable to failure rate on a single server.

- **Non-uniform node-to-node latency.**
  - Requirement: **locality control**
  - Work stealing should be aware of machine and network topology.
  - Maybe on-demand random stealing should be complemented with eager explicit pushing.
**SGP2 architecture:**
- Distributed middleware orchestrating (stateless) GAP servers across cluster.
- Communication via RPC-like protocol.

**SGP2 programming model:**
- Algorithmic skeletons (general-purpose or domain-specific).

**SGP2 implementation:**
- On top of HdpH, a Haskell EDSL for distributed-memory parallelism.
HdpH = Haskell distributed parallel Haskell is
- a parallel Haskell (ie. EDSL for task parallelism)
- for distributed memory
- implemented entirely in Haskell (+ GHC extensions).

Types:
- Par a: parallel computation (returning type a)
- Closure a: serialisable closure (of type a)
- Task = Closure (Par ()): serialisable parallel computation
- IVar a: write-once buffer (for type a)
- PE: location

Work distribution primitives:
- spark :: Task -> Par (): implicit on-demand placement
- pushTo :: PE -> Task -> Par (): explicit eager placement

Sample skeleton:
- parMap :: Closure (a -> b) -> [Closure a] -> Par [Closure b]: parallel map
Abstract view of hierarchical network topology via distance metric.

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>v</td>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>x</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
<td>1/4</td>
<td>1/4</td>
</tr>
<tr>
<td>y</td>
<td>1/2</td>
<td>1/2</td>
<td>1/4</td>
<td>0</td>
<td>1/4</td>
</tr>
<tr>
<td>z</td>
<td>1/2</td>
<td>1/4</td>
<td>1/4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Abstract view of hierarchical network topology via distance metric.

<table>
<thead>
<tr>
<th></th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>v</td>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>x</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
<td>1/8</td>
<td>1/4</td>
</tr>
<tr>
<td>y</td>
<td>1/2</td>
<td>1/2</td>
<td>1/8</td>
<td>0</td>
<td>1/4</td>
</tr>
<tr>
<td>z</td>
<td>1/2</td>
<td>1/2</td>
<td>1/4</td>
<td>1/4</td>
<td>0</td>
</tr>
</tbody>
</table>

Ultrametric distance function \( d \) on PEs defined by

\[
d(p, q) = \begin{cases} 
0 & \text{if } p = q \\
2^{-n} & \text{if } p \neq q, \, n = \text{length of common path from root to } p \text{ and } q.
\end{cases}
\]
Abstract view of hierarchical network topology via distance metric.

Ultrametric distance function $d$ on PEs defined by

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

Ball $D(p; r)$ with center $p$ and radius $r$ defined by $D(p; r) = \{q \mid d(p, q) \leq r\}$.

- Every PE $q$ inside ball $D(p; r)$ is its center.
- Every ball $D(p; r)$ is uniquely partitioned by a set of balls of radius $r/2$.

Equidistant basis for $D(p; r)$
SMP

Abstract view of hierarchical network topology via distance metric.

Ultrametric distance function $d$ on PEs defined by

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

Ball $D(p; r)$ with center $p$ and radius $r$ defined by $D(p; r) = \{ q \mid d(p, q) \leq r \}$.

- Every PE $q$ inside ball $D(p; r)$ is its center.
- Every ball $D(p; r)$ is uniquely partitioned by a set of balls of radius $r/2$.

Equidistant basis for $D(p; r)$
Abstract view of hierarchical network topology via distance metric.

Ultrametric distance function $d$ on PEs defined by

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

Ball $D(p; r)$ with center $p$ and radius $r$ defined by $D(p; r) = \{ q \mid d(p, q) \leq r \}$.

- Every PE $q$ inside ball $D(p; r)$ is its center.
- Every ball $D(p; r)$ is uniquely partitioned by a set of balls of radius $r/2$.
  $\Rightarrow$ Equidistant basis for $D(p; r)$
Distance Metric

\texttt{dist :: PE -> PE -> Dist}

Equidistant Basis

\texttt{equiDist :: Dist -> Par [(PE, Int)]}
\texttt{-- equiDist r returns a sized equidistant basis of the ball D(p;r)}
\texttt{-- where p is the current PE.}

Bounded Spark

\texttt{spark :: Dist -> Task -> Par ()}
\texttt{-- spark r task ensures that task never leaves the ball D(p;r)}
\texttt{-- where p is the current PE.}
**Bounded parMap** (implicit work distribution)

parMapBounded

\[
\begin{align*}
\text{parMapBounded} &:: \text{Dist} \\
&\rightarrow \text{Closure (a -> b)} \\
&\rightarrow [\text{Closure a}] \\
&\rightarrow \text{Par [Closure b]} \\
\end{align*}
\]

-- bounding radius around current PE
-- function closure
-- input list
-- output list

- Tasks sparked with radius \( r \) **cannot** be stolen by nodes beyond \( r \).
- Tasks are preferentially stolen from nearby PEs (but accidents can happen).
Bounded \texttt{parMap} (implicit work distribution)

\begin{verbatim}
parMapBounded :: Dist \rightarrow Closure (a -> b) \rightarrow [Closure a] \rightarrow Par [Closure b]
\end{verbatim}

- Tasks sparked with radius $r$ cannot be stolen by nodes beyond $r$.
- Tasks are preferentially stolen from nearby PEs (but accidents can happen).
**Bounded parMap** (implicit work distribution)

\[
\text{parMapBounded} \\
:: \text{Dist} \quad -- \text{bounding radius around current PE} \\
\rightarrow \text{Closure} \ (a \rightarrow b) \quad -- \text{function closure} \\
\rightarrow \text{[Closure } a \text{]} \quad -- \text{input list} \\
\rightarrow \text{Par} \ [\text{Closure } b] \quad -- \text{output list}
\]

- Tasks sparked with radius \( r \) **cannot** be stolen by nodes **beyond** \( r \).
- Tasks are preferentially stolen from nearby PEs (but accidents can happen).
Two-level parMap (explicit and implicit work distribution)

parMap2Level

\[
\begin{align*}
\text{parMap2Level} & \quad :: \text{Dist} & -- \text{distance to push big tasks} \\
& \rightarrow \text{Closure (a -> b)} & -- \text{function closure} \\
& \rightarrow [\text{Closure a}] & -- \text{input list} \\
& \rightarrow \text{Par} [\text{Closure b}] & -- \text{output list}
\end{align*}
\]

- Pushes big tasks to equidistant basis at distance $r$ from current PE.
- Each big task sparks small tasks, bounded by radius $r/2$ around its basis PE.
Two-level parMap (explicit and implicit work distribution)

parMap2Level

:: Dist  -- distance to push big tasks
-> Closure (a -> b)  -- function closure
-> [Closure a]  -- input list
-> Par [Closure b]  -- output list

- Pushes big tasks to **equidistant basis at distance** \( r \) **from current PE.**
- Each big task sparks small tasks, **bounded by radius** \( r/2 \) **around its basis PE.**
Two-level `parMap` (explicit and implicit work distribution)

```
parMap2Level :: Dist -- distance to push big tasks
               -> Closure (a -> b) -- function closure
               -> [Closure a] -- input list
               -> Par [Closure b] -- output list
```

- Pushes big tasks to **equidistant basis at distance** \( r \) from current PE.
- Each big task sparks small tasks, bounded by radius \( r/2 \) around its basis PE.
Two-level parMap (explicit and implicit work distribution)

parMap2Level

:: Dist
-> Closure (a -> b)
-> [Closure a]
-> Par [Closure b]

- Pushes big tasks to equidistant basis at distance $r$ from current PE.
- Each big task sparks small tasks, bounded by radius $r/2$ around its basis PE.
Two-level `parMap` (explicit and implicit work distribution)

```
parMap2Level :: Dist -- distance to push big tasks
    -> Closure (a -> b) -- function closure
    -> [Closure a] -- input list
    -> Par [Closure b] -- output list
```

- Pushes big tasks to **equidistant basis at distance** \( r \) **from current PE.**
- Each big task sparks small tasks, **bounded by radius** \( r/2 \) **around its basis PE.**
Two-level **parMap** (explicit and implicit work distribution)

\[
\text{parMap2LevelRelaxed} :: \text{Dist} \rightarrow \text{Closure} (a \rightarrow b) \rightarrow [\text{Closure} a] \rightarrow \text{Par} [\text{Closure} b]
\]

- Pushes big tasks to **equidistant basis at distance** \( r \) **from current PE.**
- Each big task sparks small tasks, **bounded by radius** \( r \) **around its basis PE.**
Limitation: Topology restricted to 2 levels (distances 0, $\frac{1}{2}$ and 1).

Big task granularity of 4 two-level skeletons on HECToR (128 cores)

- Two-level work distribution outperforms plain random stealing.
- Pushing of big tasks (as parMap2Level does) is more robust than stealing.
Limitation: Topology restricted to 2 levels (distances 0, $\frac{1}{2}$ and 1).

Weak scaling of two-level skeletons on HECToR (up to 32K cores)

- Estimated slowdown factor < 2 when scaling architecture by a factor of $2^{10}$.
- parMap2LevelRelaxed beats parMap2Level with increasing irregularity.
High-level API: SGP2 fault tolerance is transparent to users.
- APIs of FT and non-FT skeletons are identical.

Low-level: Replicating failed tasks
- Each task supervises the nodes executing its subtasks
  - tracking movement of subtasks due to work stealing.
- On detecting node failure, supervising task re-distributes all failed subtasks
  - implicitly if subtask was created by `spark`,
  - explicitly if subtask was created by `pushTo`.
High-level API: SGP2 fault tolerance is transparent to users.
- APIs of FT and non-FT skeletons are identical.

Low-level: Replicating failed tasks
- Each task supervises the nodes executing its subtasks
  - tracking movement of subtasks due to work stealing.
- On detecting node failure, supervising task re-distributes all failed subtasks
  - implicitly if subtask was created by `spark`,
  - explicitly if subtask was created by `pushTo`.
**High-level API:** SGP2 fault tolerance is transparent to users.
- APIs of FT and non-FT skeletons are identical.

**Low-level: Replicating failed tasks**
- Each task supervises the nodes executing its subtasks
  - tracking movement of subtasks due to work stealing.
- On detecting node failure, supervising task re-distributes all failed subtasks
  - implicitly if subtask was created by `spark`,
  - explicitly if subtask was created by `pushTo`.

```
    fault tolerant parMap2Level \frac{1}{2}
```

```
re-spark
```
Fault tolerant work stealing protocol

- **Assumption:** Nodes fail but message passing is reliable (no silent losses).
- **Overheads:**
  - Network: 2 messages (small, asynchronous)
  - Supervisor memory: copy of spark, map of spark ID to 1 or 2 locations
**Limitation:** No tracking of task movement due to work stealing.

**Cost of no failure:** 5% - 7% overhead.

**Cost of one failure:**

![Graph showing runtime and number of reallocations with one node failure.](image-url)
Thanks for Listening

Summary:
- SGP2 **locality control** improves random stealing of irregular tasks, on large architectures.
- SGP2 **fault tolerance** protects computations from occasional node failures, with low overhead.

References:
  www.macs.hw.ac.uk/~pm175/papers/Maier_Trinder_IFL2011_XT.pdf
  www.macs.hw.ac.uk/~pm175/papers/Maier_Stewart_Trinder_SAC2013.pdf

For Haskellers: SGP2 is not yet publicly available, but HdpH is.
- cabal install hdph