The Design and Implementation of Scalable Parallel Haskell

Malak Aljabri, Phil Trinder, and Hans-Wolfgang Loidl

MMnet’13: Language and Runtime Support for Concurrent Systems
Heriot Watt University

May 8, 2013
Parallel Architectures

- Parallel architectures are increasingly multi-level e.g. clusters of multicores.
- A hybrid parallel programming model is often used to exploit parallelism across the cluster of multicores e.g. using MPI + OpenMP.
- Managing two abstractions is a burden for the programmer and increases the cost of porting to a new platform.
Glasgow Parallel Haskell (GpH) Implementations

- Identify parallelism, do not control it.
- Parallelism is supported internally by the language implementation.

Two main GpH implementations:
1. GHC-SMP: shared memory.
2. GHC-GUM: distributed memory.

- Both implementations use different but related runtime environment (RTE) mechanisms.
- Good performance results can be achieved on shared memory architectures and on networks individually, but a combination of both, for clusters of multi-cores, is lacking.
**Existing Load Balancing:**

1. Searching for Local Work.
**Existing Load Balancing:**

- Spark pools are implemented as bounded work-stealing queues.
- A work-stealing queue is a lock-free data structure.
- The owner can push and pop from one end of the queue without synchronization.
- Other threads can steal from the other end of the queue.
GUMSMP

- A multilevel parallel Haskell implementation for clusters of multicores.
- Integrates the advantages of the two GpH implementations.
- Provides improvements for **automatic load balancing**.
- **The main potential benefits of GUMSMP are:**
  - Providing a scalable model.
  - Efficient exploitation of the specifics of distributed and shared memory on different levels of the hierarchy.
  - Providing a single high-level programming model.
Memory Management: the same virtual shared heap as GHC-GUM.

Communication: the same mechanism implemented in GHC-GUM.

Load Balancing: the combination of GHC-SMP and GHC-GUM mechanisms.
GUMSMP Work Distribution Mechanism

New Load Balancing:
- Work distribution of GUMSMP is hierarchy aware.
- It uses a work-stealing algorithm, through sending FISH message, on networks (inherited from GHC-GUM).
- Within a multicore it will search for a spark by directly accessing spark pools (inherited from GHC-SMP).
GUMSMP Design Objectives

- **Even but asymmetric load balancing**: Important to maintain even load distribution, but *accept imbalances* as the communication cost increases.

- **Mostly passive load distribution**: Essential to maintain *passive load distribution*, but switch to active in some cases e.g high-watermark.

- **Effective latency hiding**: The system must be designed so that communication cost is not in the critical path of cooperating computations.
**Current Implementation**

- We achieved the basic functionality on a limited number of PEs.
- With 3 PEs, 5 cores each we have:

<table>
<thead>
<tr>
<th>Sparks</th>
<th>PEs</th>
<th>PE1</th>
<th>PE2</th>
<th>PE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global sparks</td>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Local Sparks</td>
<td></td>
<td>13</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>
The design of the new multi-level parallel Haskell implementation GUMSMP is presented.

Designed for high-performance computation on networks of multi-cores.

The design focuses on flexible work distribution policies.
- Even but asymmetric load balancing.
- Mostly passive load distribution.
- Effective latency hiding.

The main benefits:
- Scalable model.
- Efficient exploitation of distributed and shared memory on different levels of the hierarchy.
- Single programming model.
Thanks for Listening ..