ARCHITECTURE-TRANSPARENT CONTROL OF PARALLELISM

HARDWARE PARALLELISM CAUSES SOFTWARE CRISIS

We critically rely on increasingly complex and computationally demanding software. Exploiting parallelism is crucial for performance, since modern hardware is massively parallel, heterogeneous, and hierarchical [1].

- Parallel programming is harder than sequential due to complexity of parallelism control
- Mainstream languages put the burden of parallelism control on the programmer, reducing productivity
- Non-deterministic race conditions and deadlocks are exceptionally difficult to detect and correct
- Rapid hardware evolution makes manual code adaptation infeasible
- Heterogeneity requires adaption and leads to proliferation of different programming models

We need a unified programming model that achieves high performance portability and productivity by adapting to evolving parallel architectures. High-level declarative approaches to structured parallel programming appear most promising.

PARALLEL FUNCTIONAL PROGRAMMING TO THE RESCUE

Functional programming expresses computations in terms of functions that map inputs to outputs and is founded on the lambda calculus, offering the following advantages:

- High level of abstraction increases productivity: programmers declare the algorithm and identify parallelism whilst run-time systems manage parallel execution
- Less manual architecture-specific code optimisations improve portability, whilst the run-time system adapts the execution to target architectures for performance
- No race conditions and deadlocks due to the deterministic programming model

Functional programming expresses computations in terms of functions that map inputs to outputs and is founded on the lambda calculus, offering the following advantages:

- Incremental parallelisation and sequential debugging, as parallel programs are equivalent to their sequential counterparts
- Fine-grained scalable parallelism, since side-effects are avoided allowing sub-expressions to be evaluated in parallel
- Ease of capturing generic parallel patterns using higher-order functions, e.g., Pipeline, Filter, MapReduce, Scan, TaskFarm

DESIGN OF AN ADAPTIVE RUN-TIME SYSTEM

Main goals:
- Implement distributed graph reduction using virtual shared memory view
- Flexibility to adapt to heterogeneous architectures for high performance portability
- Extensibility through modular design (similar to Map-Reduce and Multikernel approaches)
- Support for decentralised cost-model-based policy control and online profiling

- We extend the GUM parallel Haskell run-time system [2] with additional online profiling such as thread numbers differentiated by state, spark numbers, and allocation, among others
- We have characterised a set of divide and conquer and data parallel applications

HISTORY-BASED ADAPTIVE WORK STEALING

Random Work Stealing (baseline)

If random work request
If work available
Forward work request to a random PE
Else
Select task and run it

- Idea: de-randomise work stealing and improve success ratio by keeping track of the history of past stealing successes / failures and use this information to adapt donor selection

- Order of magnitude decrease in run-time for all applications (scalability)
- Lower run-time for history-based stealing (most effective for data parallel applications)
- Accuracy trade-off: coverage vs. stiffness of information

CONCLUSION

- We demonstrate how architectural trends lead to the need for performance portability
- We present and justify a declarative high-level semi-implicit approach to parallel programming that is based on adaptive architecture-transparent parallelism control and balances productivity and performance portability
- We enhanced the GUM run-time profiling module to monitor processor activity profiles including thread counts and allocation, among other statistics
- We show results of ongoing empirical evaluation of architecture-transparent parallelism control using system information (history) to adapt parallel computations (stealing)

FUTURE WORK

- Implement and evaluate Spark Co-Location policy that uses information about computational structure and architecture to decide which computations to donate
- Implement and evaluate a policy that distinguishes parallelism generators from workers and adapts load balancing based on this information
- Integration of a cost model into GUM to unify architecture-transparent control of parallelism based on architectural and system information and to facilitate optimisation of the available heuristics and aforementioned policies

References
[3] The authors are grateful to Natalia Chechina, Rob Stewart, Prabhat Tootoo and Julian Godesa for inspiring discussions.

Acknowledgments
This joint work with Dr Hans-Wolfgang Loosf and Prof Dr Greg Nicholson is supported by SICSA.

Evgueni Belikov
etb120@hw.ac.uk