History-Based Adaptive Work Distribution*

Evgenij Belikov

Heriot-Watt University, School of Mathematical and Computer Sciences
Riccarton, EH14 4AS, Edinburgh, Scotland
eb120@hw.ac.uk

Abstract

Exploiting parallelism of increasingly heterogeneous parallel architectures is challenging due to the complexity of parallelism management. To achieve high performance portability whilst preserving high productivity, high-level approaches to parallel programming delegate parallelism management, such as partitioning and work distribution, to the compiler and the run-time system. Random work stealing proved efficient for well-structured workloads, but neglects potentially useful context information that can be obtained through static analysis or monitoring at run time and used to improve load balancing, especially for irregular applications with highly varying thread granularity and thread creation patterns. We investigate the effectiveness of an adaptive work distribution scheme to improve load balancing for an extension of Haskell which provides a deterministic parallel programming model and supports both shared-memory and distributed-memory architectures. This scheme uses a less random work stealing that takes into account information on past stealing successes and failures. We quantify run time performance, communication overhead, and stealing success of four divide-and-conquer and data parallel applications for three different update intervals on a commodity 64-core Beowulf cluster of multi-cores.

1998 ACM Subject Classification C.1.4 Parallel Architectures, D.1.1 Applicative (Functional) Programming, D.1.3 Parallel Programming, D.3.4 Run-Time Environments, D.4.1 Scheduling

Keywords and phrases Adaptive Load Balancing, High-Level Parallel Programming, History, Work Stealing, Context-Awareness

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.3

1 Introduction

In the many-core era parallelism is one of the key sources of application performance [19]. Unfortunately, exploiting parallelism is challenging due to the added complexity of managing parallelism [2], in particular of partitioning and work distribution across the available processing elements (PEs). To preserve high programmer productivity, high-level approaches to structured parallel programming delegate parallelism management to the compiler and the run-time system (RTS). Moreover, manual adaptation to rapidly evolving and increasingly heterogeneous and hierarchical parallel architectures is deemed infeasible mandating adaptive solutions to achieve high performace portability [17, 7]. Furthermore, distribution is required for scalability beyond one physical machine as often required in important domains such as Large-Scale Data Analysis, Scientific Computing and Cloud Computing.

A state-of-the-art work distribution scheme is *random work stealing* where idle workers attempt to steal from victims chosen uniformly at random. This policy is scalable due to its decentralised and probabilistic nature and proved efficient for well-structured workloads [6].

* This work is supported by the Scottish Informatics and Computer Science Alliance (SICSA).

Imperial College Computing Student Workshop (ICCSW'14).
Editors: Rumyana Neykova and Nicholas Ng; pp. 3–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, due to randomness it neglects information that can be obtained through static analysis or monitoring at run time and potentially used to improve load balancing and locality, especially for applications that can be characterised as irregular due to highly varying thread granularity and thread creation patterns. In this paper, the term *thread* refers to a light-weight thread managed in user space, not to a fully-fledged OS-thread.

We investigate the effectiveness of an adaptive work distribution scheme that aims to improve load balancing in the context of a high-level non-strict functional language – an extension of Haskell [13] – which provides a deterministic parallel programming model and supports both shared-memory and distributed-memory architectures whilst dynamically managing work distribution. This scheme mostly uses a less randomised variant of work stealing that takes into account information on past stealing successes and failures to improve workload distribution. We quantify run time performance, communication overhead, and stealing success of four divide-and-conquer and data parallel functional applications on a modern 64-core Beowulf-class cluster consisting of 8-core nodes.

2 Background and Related Work

We describe Glasgow parallel Haskell that provides a unified high-level semi-explicit deterministic parallel programming model and its RTS that was extended to take additional contextual information into account when making policy decisions[1]. Additionally, most important related work is discussed along with the relevant concepts, policies, and mechanisms.

2.1 Parallel Functional Programming

Glasgow parallel Haskell (GpH) [20] provides the `par` combinator to express advisory parallelism, which takes two arguments and potentially executes the first argument in parallel whilst returning the second that is evaluated by the parent thread. Additionally, the `pseq` combinator is defined that fixes the evaluation order by evaluating the first argument and then the second. Lazy polymorphic higher-order functions are used to define *Evaluation Strategies* [22, 15] which further raise the level of abstraction by *separating coordination from computation*, similar to algorithmic skeletons [11].

Notably, the model is deterministic by design thus preventing the occurrence of race conditions and deadlocks that are notoriously difficult to detect and correct. For an overview of parallel programming models refer to recent surveys [4, 10]. GpH delegates most of the parallelism management to the RTS to provide architecture-independence at language level, whilst retaining optimisation flexibility at the system level. Below Listing 1 illustrates the use of the GpH combinators to parallelise the `QuickSort` algorithm. Note the conciseness and the close correspondence of the code to the common mathematical notation.

Listing 1 `QuickSort` Implementation in GpH.

```
par_qsort :: Ord a => [a] -> [a]
par_qsort [] = []
par_qsort (pivot:xs) = lower `par` higher `pseq` merge
  where lower  = par_qsort [y | y <- xs, y <  pivot]
        higher = par_qsort [y | y <- xs, y >= pivot]
        merge  = lower ++ (pivot:higher)
```

[1] the source code of this experimental RTS is available upon request

2.2 Adaptive Run-Time System Policy Control

GUM (Graph Reduction for a Unified Machine Model) [21] is an adaptive RTS for GpH that implements distributed graph reduction using a *virtual shared heap*. Figure 1 depicts the GUM control model which is based on observing system state as well as receiving some environmental information and controlling the policy decisions based on this information.

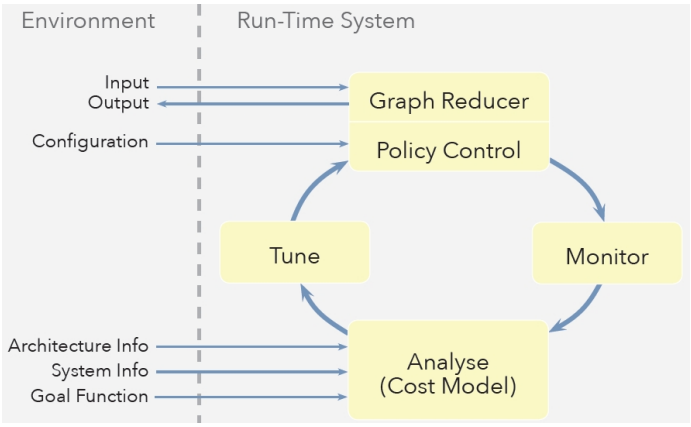

Figure 1 GUM Control Model Enables Adaptation.

The emphasis of the control model is on flexible adaptation at run time as opposed to commonly used static partitioning and work distribution schemes (e.g. in MPI or OpenCL). For every `par`, a *spark*, i.e. a pointer to an unevaluated closure in the shared graph representing potentially parallel work, is added to the spark pool. Sparks are cheap and can be turned into light-weight threads for parallel execution, which are more expensive as they store their state and a stack. A parent thread can *subsume* a child thread by evaluating the spark it has created sequentially to reduce thread creation overhead and increase granularity, a mechanism similar to lazy task creation [16].

Work Stealing Passive work distribution or work stealing is an established mechanism used in many language run-time systems to balance computational load across PEs [6]. It is scalable due to its decentralised nature and efficient for well-structured workloads since stealing efforts are amortised across idle PEs. Once no local threads are available the system sends a steal request to a PE chosen uniformly at random if the local spark pool is empty. If a PE receives a steal request (a so-called `FISH` in GUM terminology), it first looks for work in the spark pool and if there are sparks available donates the oldest and thus probably large spark in FIFO fashion [8]. Alternatively, if no work is available it forwards the request to another randomly chosen PE, unless the message exceeds its time-to-live limit. In that case the request is sent back to the original PE and is registered as a failed stealing attempt.

Recently, GUM has been ported to computational Grids and results on a heterogeneous cluster demonstrated benefits of using information such as computational power of the PEs, load, as well as latency between them [1]. In particular, simply placing main PE on the most powerful node of the most powerful cluster lead to increased performance. Similarly, a simulation study of divide-and-conquer applications on heterogeneous clusters comprising homogeneous PEs, showed that using load information is beneficial in a hybrid locally centralised and globally distributed scheme, where one PE is chosen to manage information as cluster head, whereas across clusters the heads communicate in a decentralised fashion [14]. Moreover, language extensions demonstrated the importance of improving locality [3].

3 Using Historical Information in Work Distribution Decisions

Several complementary policies were identified based on an application characterisation [5]. The key idea is to use monitored RTS-level information to *de-randomise* work stealing to increase the flexibility of adaptation to architectural and behavioural system-level changes. Here we investigate the effectiveness of using dynamic information about past stealing successes and failures. Additionally, we discuss the importance of selecting a suitable update interval. To our knowledge the use of this policy has not been previously explored in the context of a non-strict functional language with a semi-explicit parallel programming model.

3.1 History-Based Stealing

Work stealing is mainly concerned with the following decisions: a) which PE should one steal from as a thief (i.e. PE with no work); b) to which PE should one forward a `FISH` as a victim with no work; c) which of the available sparks should one donate as a victim with work.

Our policy extension is aimed at reducing communication overhead by increasing the fishing success ratio (i.e. the percentage of sent `SCHEDULE` messages containing work in relation to the total number of sent `FISH` messages requesting some work) by monitoring and storing information on recent stealing successes and failures. We investigate whether simply trying to steal from PEs where recent stealing attempts were successful yields any substantial benefits. If no most suitable PE could be selected due to either lack of successful stealing attempts or due to stale information, the algorithm falls back to random stealing. The policy is expected to work best in cases where a set of parallelism generators is fairly stable over time. The overhead is low as it involves counters and updating cost is amortised as it happens at garbage collection times and on arrival of `FISH` or `SCHEDULE` messages.

The key change to the mechanism is in victim selection: a table is maintained that records whether last stealing attempt from a given PE was successful. Logically, this can be viewed as a function $f(i :: PEid) \rightarrow (successInfo_i, timeStamp_i)$. Table 1 shows how the stored data is interpreted to select a PE with most consecutive successes, tie-breaking on the index.

Table 1 Overview of the Stored Historical Information.

information table field	value = 0	value > 0
history information	failed stealing attempt	number of consecutive successes
time stamp	information is stale	time of last update

3.2 Balancing Accuracy and Coverage

We also record a time stamp of the last update for each PE to judge whether the stored information is reliable and purge stale data at garbage collection times. An RTS flag is set to select an interval used to invalidate any table entries for PEs for which no update happened during the last interval.

The main challenge is the choice of a suitable interval such that highest *coverage* of the PEs is achieved whilst keeping *accurate* information. In fact, using stale information can be misleading and reduce fishing success ratio, which may lead to performance degradation.

4 Empirical Evaluation

We report application performance from a median run out of three on a cluster of multi-cores with 64 PEs and focus on relative speedups as we are interested in the behaviour of the parallel applications. We use CentOS 6.5, GHC 6.12.3, gcc 4.4.7, and PVM 3.4.6. Due to space limitations the focus is on cluster results as it is a more challenging architecture because of higher inter-node latency and hence higher associated communication costs.

The Beowulf cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores, 256 KB L2 cache, 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores, 6MB shared L2 cache and 16GB RAM. The machines are connected via Gigabit Ethernet with an average latency of 150ns.

4.1 Parallel Applications

We use several applications from the *nofib* suite [18] and from a study of Evaluation Strategies [15]. The applications employ simple yet expressive *divide-and-conquer* (D&C) and *data-parallel* patterns which are considered representative of a wide range of applications [11, 9]. Using D&C, the final result is computed by merging the solutions of smaller problems obtained by recursively sub-dividing and solving the initial problem. A *threshold* can be used to increase granularity by limiting spark creation to a specified depth of the tree. Data-parallel applications exploit the parallelism by applying a function to the elements of a data structure. Granularity can be tuned by *chunking* several elements together. We measure run times, the number of messages sent, and fishing success ratio on up to 64 PEs.

- The D&C `parfib` program computes the Nth Fibonacci number using arbitrary-length integers; we use $N = 50$ and a threshold of 23; used to assess thread subsumption capabilities of the RTS, this benchmark is representative of regular and flat D&C applications with a single source of parallelism; splitting and the combining phases require two arithmetic operations on integers of arbitrary length, the sequential work is exponential.
- The D&C `worpitzky` application from the symbolic computation domain checks the Worpitzky identity for two arbitrary-length integers; we take 19 to the exponent of 27 and use a threshold of 10; at the top level this requires one exponentiation, one equality comparison, and a sum of n intermediate results, which are computed in parallel and for the other part requires two arithmetic operations and binomial computation using three factorial and three arithmetic operations; parallel computations include a single source of parallelism and three arithmetic operation for both the combine and the split phase.
- The D&C `coins` program computes ways to pay out a specified amount from a set of coins; in our case the value is 5777; the program is similar to `parfib` as the split and the combine phases require one arithmetic operation each, whilst sequential solution requires finding suitable permutations of coins.
- The data-parallel `sumeuler` program computes the sum over euler totient numbers in a given integer interval (which plays the role of explicit chunking to control granularity) by applying a function to each element of the list generated from the given intervals in parallel, and is fairly irregular; we use interval from 0 to 100000 with a chunk size of 500; all the degree of parallelism with merely 200 sparks is relatively low and all the parallelism is generated in the beginning of the execution by the main PE.

4.2 Results and Evaluation

The early results shown below demonstrate the effects of using history-based stealing but are rather indicative than conclusive. In Figure 2, the left column of graphs shows the run times (in seconds, note the different scales) of the applications on 2, 4, 16, and 64 PEs and the right column presents the corresponding fishing success ratios (in percent of total number of `FISH` messages). We observe that run times decrease by an *order of magnitude* for all applications demonstrating scalability. However, for most applications the benefit from adding more PEs reduces with the number of PEs due to lack of work and increased overheads. We aim to reduce communication overhead and improve load balancing by using the enhanced policy.

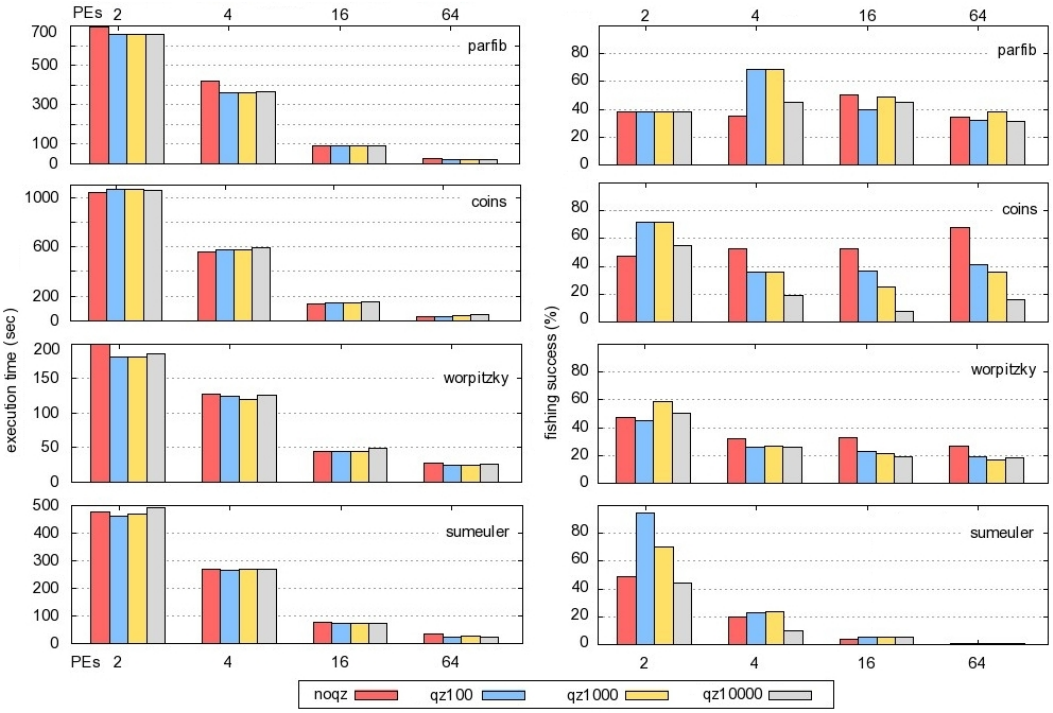

Figure 2 History-Based Stealing: Execution Times and Fishing Success Ratios.

Each bar for each number of PEs represents the baseline random stealing (leftmost of the four) or using history with a small (qz100 that stands for 100 ms), medium (qz1000) or large interval. A small interval leads to rapid invalidation of information so that we ensure high accuracy but coverage is often low, whereas opposite is the case if the interval is large. As noted above, the challenge is in finding optimal interval to balance accuracy and coverage.

On 64 PEs the run times for applications using history are consistently decreased by up to 34% (as for `sumeuler`). We can attribute the effectiveness of the policy for `sumeuler` to the fact that all parallelism is generated by the main PE, hence past behaviour appears predictive of future behaviour during the initial phase of the computation. In particular for low numbers of PEs the fishing success ratio is mostly higher if history is used. However, it is lower in most cases for D&C applications as new parallelism sources are created dynamically but are rather short-lived. There is not much difference in success ratio for `sumeuler` for higher PE numbers as there is not enough work available[2]. By contrast, more work is generated

[2] also indicated by the high percentage of `FISH`es in relation to the total number of messages (cf Table 2)

Table 2 Number of `FISH` Messages versus Total Sent Messages (on 64 PEs).

interval	noqz		qz100		qz1000		qz10000	
	FISHes	Total	FISHes	Total	FISHes	Total	FISHes	Total
sumeuler	34968	35714	18650	19451	20395	21200	15190	16004
parfib	5137	12134	3965	9027	4874	12196	4801	10793
coins	7710	28774	10541	27856	12694	30804	38715	63320
worpitzky	73278	153438	71437	125510	74370	125630	82155	139641

throughout a longer phase of the D&C computations. However, history appears misleading in this case as generators only create few sparks and further sparks are generated elsewhere.

As shown in Table 2, using information on past successes also significantly reduces the number of `FISH`es for the data-parallel `sumeuler` from 34968 (baseline) by 57% to 15190 (qz10000) for the rather regular D&C `parfib` from 5137 by 23% to 3965 (qz100), contributing to the reduction of communication overhead which helps reduce execution time. On the other hand, history does not reflect well the run time behaviour of `worpitzky` (modest 2.5% decrease) and `coins` (a disappointing 37% increase), where the threads are more numerous and more fine-grained than in the other applications.

5 Conclusion and Future Work

We have investigated the effectiveness of using information on past stealing successes to improve victim selection of random work stealing to increase stealing success ratio and reduce communication overhead. We quantify run time performance of four applications on a cluster of multi-cores and use profiling data to explain application behaviour. We find improved run time of up to 34% along with increased fishing success rate and reduced number of `FISH` messages for data-parallel and for regular D&C applications. However, this heuristic fails in cases where past application behaviour is not predictive of the future behaviour as it is the case for more irregular D&C applications with large number of very fine-grained threads.

Ongoing work is focused on investigating automated ways of parameter selection and tuning as well as on exploring complementary policies such as temporary switching to work-pushing and using information on the source of the sparks to avoid exporting sparks that could have been successfully subsumed by the parent and would otherwise cause additional communication overhead if the results are needed by the parent. In general, predicting the amount of work associated with a spark proved very challenging [12], hence we aim to use additional information to *co-locate* sparks from the same source of parallelism to improve locality and selection of the spark to donate according to implicit ancestry dependencies.

In the future, we plan to add larger applications to the set of our benchmarks, to enrich the used information by architectural characteristics and to use cost models as a more systematic way to adaptively control policies within a RTS to achieve high performance portability for a high-level non-strict functional parallel programming language.

Acknowledgements This work is part of a collaboration with Hans-Wolfgang Loidl and Greg Michaelson to whom the author is grateful for encouragement and helpful discussions. The author also thanks three anonymous reviewers for comments that improved this paper.

References

1 A. Al Zain, P. Trinder, G. Michaelson, and H.-W. Loidl. Evaluating a high-level parallel language (GpH) for computational Grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

2 K. Asanovic, R. Bodik, J. Demmel, et al. A view of the parallel computing landscape. *CACM*, 52:56–67, October 2009.

3 M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow parallel Haskell (GpH). *Procedia Computer Science*, 9:1807–1816, 2012.

4 E. Belikov, P. Deligiannis, P. Totoo, et al. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Heriot-Watt University, 2013.

5 E. Belikov, H.-W. Loidl, and G. Michaelson. Characterisation of Parallel Functional Applications. In *Draft Proceedings of the 2014 Symposium on Trends in Functional Programming*, Utrecht University, 2014.

6 R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

7 A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

8 F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the Conference on Functional Program Language and Computer Architecture*, pages 187–194. ACM, 1981.

9 J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

10 J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

11 H. Gonzalez-Velez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12), 2010.

12 T. Harris and S. Singh. Feedback directed implicit parallelism. *ACM SIGPLAN Notices*, 42(9):251–264, September 2007.

13 P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of the 3rd ACM SIGPLAN History of Programming Languages Conference*, pages 1–55, June 2007.

14 V. Janjic and K. Hammond. How to be a successful thief. In *Euro-Par 2013 Parallel Processing*, pages 114–125. Springer, 2013.

15 S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better Strategies for parallel Haskell. In *Proc. of the 3rd ACM Symposium on Haskell*, pages 91–102, 2010.

16 E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

17 J. Owens, D. Luebke, N. Govindaraju, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

18 W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

19 H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

20 P. Trinder, E. Barry Jr., M. Davis, et al. GpH: An architecture-independent functional language. *IEEE Transactions on Software Engineering*, 1998.

21 P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI'96 Conf.*, 1996.

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