How to be a Successful Thief

Feudal Work Stealing for Irregular Divide-and-Conquer Applications on Heterogeneous Distributed Systems

Vladimir Janjic and Kevin Hammond

School of Computer Science, University of St Andrews, Scotland, UK
vj32@st-andrews.ac.uk, kh@cs.st-andrews.ac.uk

Abstract. Work Stealing has proved to be an effective method for load balancing regular divide-and-conquer (D&C) applications on heterogeneous distributed systems, but there have been relatively few attempts to adapt it to address *irregular* D&C applications. For such applications, it is essential to have a mechanism that can estimate dynamic system load during the execution of the applications. In this paper, we evaluate a number of work-stealing algorithms on a set of generic Unbalanced Tree Search (UTS) benchmarks. We present a novel *Feudal Stealing* work-stealing algorithm and show, using simulations, that it delivers consistently better speedups than other work-stealing algorithms for irregular D&C applications on high-latency heterogeneous distributed systems. Compared to the best known work-stealing algorithm for high-latency distributed systems, we achieve improvements of between 9% and 48% for irregular D&C applications.

Keywords: Irregular Parallelism, Work Stealing, Divide-and-Conquer, Heterogeneous Systems.

1 Introduction

Work stealing [4], where idle "thieves" steal work from busy "victims", is one of the most appealing load-balancing methods for distributed systems, due to its inherently distributed and scalable nature. Several good work-stealing algorithms have been devised for distributed systems with non-uniform communication latencies [2,3,14,15,13]. Most of these algorithms are, however, tailored to *regular* Divide-and-Conquer (D&C) applications. In scientific computations, for example, it is very common to encounter *irregular* D&C applications, where the structure of parallelism for different application tasks varies quite significantly.

We previously showed [10] that in order to obtain good speedups for irregular D&C applications, it is essential to use *dynamic system load information* as part of a work-stealing algorithm. In this paper, we consider several algorithms that can be used to obtain suitable load information. In particular, we compare *centralised* (where this information is kept on a fixed set of nodes), *distributed* (where this information is exchanged in a peer-to-peer way between all nodes)

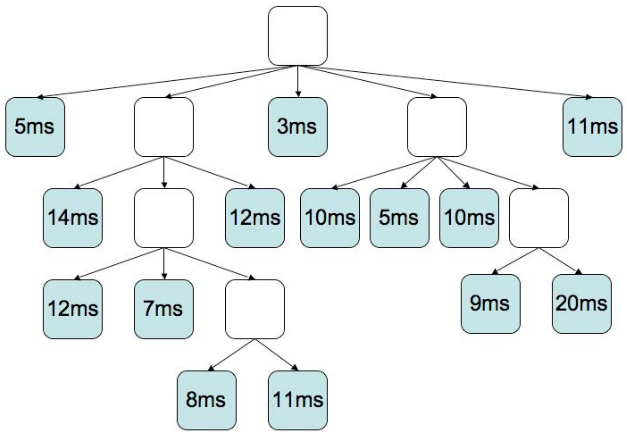


Fig. 1. Example task graph for an irregular D&C application

and *hybrid* (a combination of the two) methods for obtaining load information. We also describe a novel *Feudal Stealing* algorithm, a new algorithm that exploits dynamic load information, but which significantly outperforms other similar work-stealing algorithms in terms of the speedups that are obtained.
This paper makes the following specific research contributions:

– We present a novel *Feudal Stealing* algorithm, which uses hybrid dynamic load information approach;
– We evaluate representative state-of-the-art work-stealing algorithms that use centralised, distributed and hybrid dynamic load information on the Unbalanced Tree Search (UTS) benchmark of irregular D&C applications;
– We show, using simulations, that our new Feudal Stealing algorithm delivers notably better speedups on irregular D&C applications than the other state-of-the-art algorithms we consider.

2 Work Stealing for Divide-and-Conquer Applications

Divide-and-conquer (D&C) applications can be represented by task trees, where nodes in the tree represent tasks, and edges represent parent-child relationships. One especially interesting class is that of *irregular* D&C applications, whose task trees are highly unbalanced (see Fig. 1), and which frequently occur in scientific computations (e.g. Monte-Carlo Photon Transport simulations [7], implementations of the Min-Max algorithm and Complex Polynomial Root Search [6]). They are also frequently used as benchmarks for evaluating load-balancing algorithms (e.g. the Unbalanced Tree Search [12] and Bouncing Producer-Consumer [5] benchmarks). We focus on load balancing irregular D&C applications on widely distributed systems, which comprise a set of *clusters* communicating over high-latency networks. Each cluster comprises a set of *processing nodes*, where nodes from the same cluster are connected by fast, local-area networks. Such systems correspond to grid-like or cloud-like server farms. They are typically highly heterogeneous both in terms of the characteristics of the individual nodes (processing power, memory characteristics etc.) and the networks that connect clusters (communication latency, bandwidth etc.). This paper focuses on how to deal with heterogeneous communication latencies in such systems.

Each processing node stores the tasks that it creates in its own *task pool*. Load balancing between nodes uses *work stealing*. In such a setting, when a node's task pool becomes empty, that node becomes a *thief*. The thief sends a *steal attempt* message to its chosen *target* node. If the target has more than one task in its task pool, it becomes a *victim* and returns one (or more) tasks to the thief. Otherwise, the target can either forward the steal attempt to some other target or, alternatively, it can send a negative response to the thief, which then deals with it in an appropriate way (either by initiating another steal attempt or by delaying further stealing). To make our discussion more focused, we assume that only one task is sent from the victim to the thief, and that any targets that do not have any work will always forward the steal attempt to some other target. In addition, following the usual practice in work stealing for divide-and-conquer applications, we assume that the victim always sends the *oldest* task from its task pool to the thief. In divide-and-conquer applications, this usually corresponds to the largest task, that also generates the most additional parallelism. Finally, we assume that execution starts with one node executing the main task (the root of the task tree), and all other nodes are thieves that start stealing immediately.

In order to hide the potentially high communication latencies that can occur in distributed systems, it is essential that work-stealing algorithms employ good methods for selecting targets. This serves to minimise the number of wasted messages. Work-stealing algorithms can be divided into two broad classes, according to the type of information that they use for the target selection: i) algorithms that only use static information about network topology; and ii) algorithms that also use dynamic load information.

2.1 Algorithms That Use Only Network Topology Information

The most important algorithms that use only network topology information are:

– *Random Stealing* [3], where a thief always selects a random target.
– *Hierarchical Stealing* [2], where nodes are organized into a tree, according to communication latencies. A thief first tries stealing from closer nodes (i.e. its descendants in the tree). If it fails to obtain any work there, it attempts to steal from its parent, which then repeats the same procedure.
– *(Adaptive) Cluster-Aware Random Stealing* (CRS) [14], where each node divides the set of all other nodes into two sets: i) nodes from the same cluster (*local nodes*), and ii) those outside of it (*remote nodes*). A thief attempts to steal *in parallel* from a random local node and a random remote node.In this way, thieves hope to obtain work locally, but remote stealing will prefetch work. In the adaptive variant, thieves prefer closer clusters.

We have previously shown [10] that these algorithms perform well for regular D&C applications, but that they run into problems for highly-irregular applications. In such applications, all the tasks may be concentrated on relatively few nodes, and the semi-randomised way of selecting targets that is employed by these algorithm can consequently perform badly. We have also shown that it would be beneficial to extend them with mechanisms for obtaining and using dynamic load information. Our main objective in the rest of the paper is to describe and compare different methods for obtaining and using load information in work-stealing, in order to discover the methods that give the best estimation of system load and the best overall speedups. We discuss existing methods in the next section, our novel Feudal Stealing method is described in Section 3.

2.2 Algorithms That Use Dynamic Load Information

Some work-stealing algorithms use dynamic load information to estimate the size of node task pools, and so inform the choice of target. In algorithms with *centralised* load information, a fixed set of nodes is responsible for managing load information. These nodes act as routers for steal attempts, forwarding them to targets. There are two basic ways to do this:

1. *Fully Centralised* methods, where all nodes periodically send their load information to a single *central* node. A thief sends a steal attempt to the central node, and this is forwarded to the victim that is *nearest* to the thief[1].
2. *Hierarchical Load-Aware Stealing (HLAS)* [11], analogous to Hierarchical Stealing. Each node periodically sends its load information to its parent. Based on its load information, a thief then attempts to steal from a child with non-zero load. If all children have zero load, the steal attempt is sent to the thief's parent, which then repeats the same procedure for finding work.

The main appeal of such algorithms is that load information is updated regularly, and it will therefore be relatively accurate. However, this also means that significant strain may be placed on the central nodes, since they have to communicate frequently with the rest of the system. For high-latency networks with fine-grained tasks, this information may also be inaccurate. In contrast, for algorithms using *distributed* load information, each node holds its own approximations of the load of all other nodes. A representative example is the *Grid-GUM Stealing* algorithm [1]. In this algorithm, timestamped load information is attached to each stealing-related message (steal attempts, forwarding of steal attempts etc.). The recipient of a message compares this information against its own load information, and updates both the information that is contained in the message (if the message is to be forwarded further) and its own load information. Provided that nodes frequently exchange load information, they will then obtain good approximations to the system load. In algorithms with distributed load information, work-stealing is still fully decentralised, and no significant overheads

[1] We also considered a variant of this algorithm where the thief forwards the steal attempt to a *random* target with work, rather than to the closest one. This variant, however, delivered consistently worse speedups, so we will not consider it further.

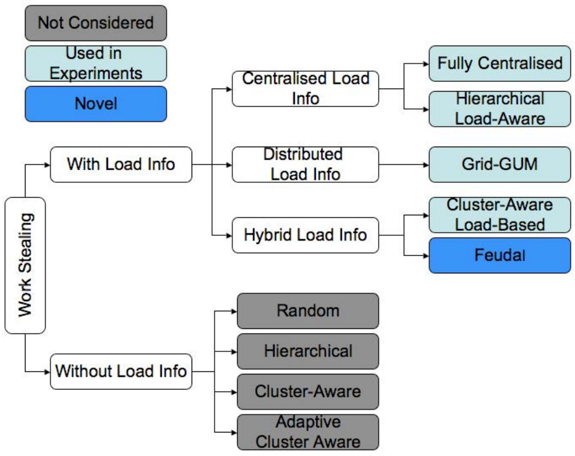


Fig. 2. The work-stealing algorithms that this paper considers

are introduced in approximating system load [1]. However, the accuracy of the load information that a node has (and, conversely, that the rest of the system has about that node) depends on how often it communicates with the rest of the system. An isolated node can easily have outdated load information, and the rest of the system may also have outdated information about its load. *Hybrid* algorithms combine both types of algorithms in order to overcome the disadvantages of each approach. We present one such method here, *Cluster-Aware Load-Based Stealing (CLS)* [14], before introducing our novel extension, *Feudal Stealing*. Like the CRS algorithm, the CLS algorithm considers only two levels of communication latencies (local and remote). In each cluster, one node is nominated as a central node, and every other node in the cluster periodically sends its load information to this node. All nodes in the cluster, apart from the central node, perform only (random) local stealing. When the central node determines that the load of the cluster has dropped below some threshold, it initiates remote stealing from a randomly selected remote node. This approach has several drawbacks. Firstly, tasks that are stolen remotely are always stored on central nodes, which means that additional messages are needed to distribute these tasks to their final destinations. Secondly, as with CRS, when all tasks are concentrated on a few nodes, random remote stealing may be unacceptable. Fig. 2 gives an overview of the work-stealing algorithms that we consider in this paper.

3 Feudal Stealing

Feudal Stealing represents an attempt to combine the best features of the CRS, CLS and Grid-GUM Stealing algorithms while avoiding their drawbacks. Its basic principle is similar to the CRS algorithm. A thief initiates both (random) local and remote stealing in parallel. Remote stealing is done via central nodes (one for each cluster, as in the CLS algorithm). The thief sends a *remote-steal*

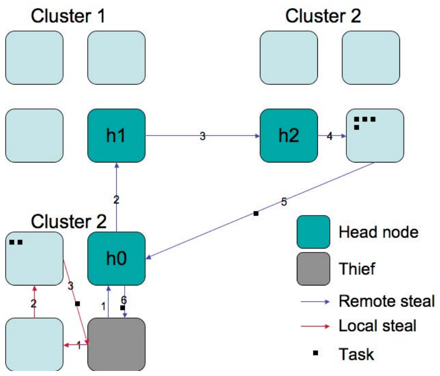


Fig. 3. Overview of Feudal Stealing

message to its central node, which then forwards it to some other, appropriately chosen, central node. When a central node receives a remote-steal message, it forwards it either to some node in its own cluster or to some other central node, depending on the cluster load. When a node is found that has work, one task from its task pool is returned to the original thief via the central nodes.

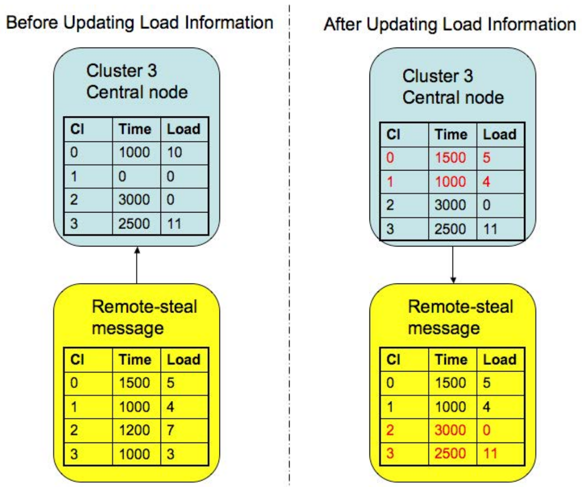
Fig. 3 gives an overview of Feudal Stealing. Here, the grey node is a thief, which initiates remote and local stealing attempts. The local stealing message, whose path is shown by red arrows, visits random local nodes, looking for work. In this case, a suitable local node is found in the second hop, and the work is immediately returned to the thief. The remote stealing message, whose path is shown by blue arrows, starts by visiting the central node of the local cluster, *h0*. It is then forwarded to the other central nodes, until one whose cluster has work is found (the third hop in the figure, *h2*). It is then forwarded to the node within the cluster that has work (hop 4), which returns the work to *h0* (hop 5). This forwards the work to the original thief (hop 6).

In Feudal Stealing, as with the CLS algorithm, central nodes keep load information for their cluster. However, they also keep load information for other clusters, and this information is used when a central node needs to decide where to forward a remote-steal message. The remote-steal message is forwarded to a random central node, with respect to estimated loads of central nodes. Load information is exchanged between central nodes in a similar way to Grid-GUM Stealing. The central node's load approximation is attached to each message that is sent (or forwarded) from that node. Each central node updates its own load information from the messages that it receives (see Fig. 4).

4 Simulation Experiments

For our simulation experiments, we use the publically-available highly-tunable SCALES[2] work-stealing simulator [8]. SCALES has been shown to accurately simulate realistic runtime systems [8,9]. It models irregular D&C applications

[2] Source code is available at `http://www.cs.st-andrews.ac.uk/jv/SCALES.tar.gz`


Before Updating Load Information
After Updating Load Information

Cluster 3 Central node (before)

Cl	Time	Load
0	1000	10
1	0	0
2	3000	0
3	2500	11

Remote-steal message (before)

Cl	Time	Load
0	1500	5
1	1000	4
2	1200	7
3	1000	3

Cluster 3 Central node (after)

Cl	Time	Load
0	1500	5
1	1000	4
2	3000	0
3	2500	11

Remote-steal message (after)

Cl	Time	Load
0	1500	5
1	1000	4
2	3000	0
3	2500	11

Fig. 4. Exchanging load information between a central node and a message

on systems consisting of inter-connected groups of clusters. We have chosen to use simulations for our experiments, rather than implementation in a real runtime-system, solely in order to abstract from the details of specific runtime systems. This enables us to ignore some specific overheads (e.g. task and thread creation, message processing) that certain runtime systems impose, and which can obstruct the results we are trying to obtain.

Our main benchmark is *Unbalanced Tree Search (UTS)* [12], which simulates state space exploration and combinatorial search problems. UTS is commonly used as a benchmark to test how a system copes with load imbalance. It models a traversal of an implicitly constructed tree, parameterised by shape, depth, size and imbalance. In *binomial* UTS trees, which we denote by UTS(m,q, $rootDeg$, S), each node has m children with probability q, and has no children with probability $1 - q$. $rootDeg$ is the number of children of the root node. When $qm < 1$, this process generates a finite tree with an expected size of $\frac{1}{1-qm}$. The product qm also determines how unbalanced the UTS tree is. As this product approaches 1, the variation in the sizes of subtrees of the nodes increases dramatically. S denotes the cost (in miliseconds) of processing a node. In order to keep the number of experiments manageable, we have decided to model one specific distributed system consisting of 8 clusters, with 8 nodes in each cluster, giving a total of 64 nodes, as shown in Fig. 5. Each cluster is split into two *continental* groups of clusters, with an inter-continental latency of 80ms. Each continental group is further split into two *country* groups, with an inter-country latency of 30ms. The latency between clusters that belong to the same country group was set to be 10ms, and the latency between nodes from the same cluster to be 0.1ms This models the latencies of a system where clusters from different continents, countries and sites within countries are connected into one large supercomputer.

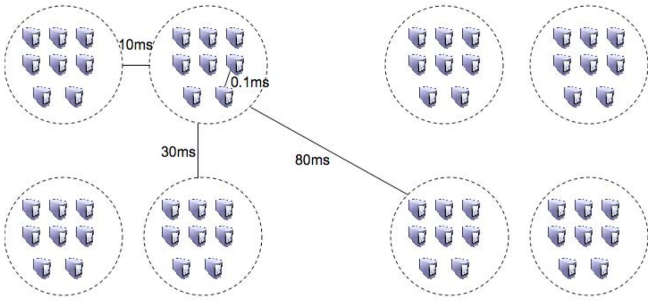


Fig. 5. The model of a system used in simulations

4.1 Results

This section evaluates the speedups that can be obtained by work-stealing algorithms that use dynamic load information. We define speedup to be the ratio of the simulated runtime of an application on a one-node system node to that on the distributed system. In our first set of experiments, we consider the $UTS(7, q, 3000, 5ms)$ applications, where q takes values $0.08, 0.09.0.1, 0.11, 0.12$, 0.13 and 0.14. This represents applications with a large number of tasks (approximately 7011, 8246, 9223, 13044, 18751, 33334 and 150000 respectively), and with increasing irregularity (where as q increases, so the tree becomes more unbalanced). Fig. 6 shows the speedups of these applications under all of the algorithms that we have considered. We observe that for less irregular applications, Feudal Stealing, Grid-GUM Stealing and Fully Centralised Stealing have approximately the same performance. However, for more irregular applications, Feudal Stealing notably outperforms all the other algorithms. We can also observe that algorithms that use distributed load information (Feudal Stealing and Grid-GUM Stealing) generally outperform those that mostly rely on centralised load information. The improvements in speedup that Feudal Stealing brings over the next best algorithm (Grid-GUM) vary from 9% for $UTS(7, 0.11, 3000, 5ms)$ ($qm = 0.77$) to 48% for $UTS(7, 0.14, 3000, 5ms)$ ($qm = 0.98$).

Fig. 7 helps explain these results. We focus here on the three algorithms that deliver the best speedups (Fully Centralised, Grid-GUM Stealing and Feudal Stealing), and on the highly-irregular applications. The left part of the figure shows the percentage of successful steal attempts (i.e. those that manage to locate work). As expected, we can see that, for more irregular applications, the fully-centralised algorithm has the highest success rate. Feudal Stealing also has a very high success rate, whereas Grid-GUM is noticeably worse than the other two. The right part of the figure shows the average time that it takes for a node to obtain work (i.e. the average time between the initiation of a successful steal attempt and the arrival of the work to the thief). We can see that the time it takes to locate work is greatest for fully centralised stealing. This is expected, since all steal attempts need to go via the central node. This makes the steal attempts quite expensive for nodes that are further away from the central node. In Feudal Stealing and Grid-GUM, nodes are able to obtain work much

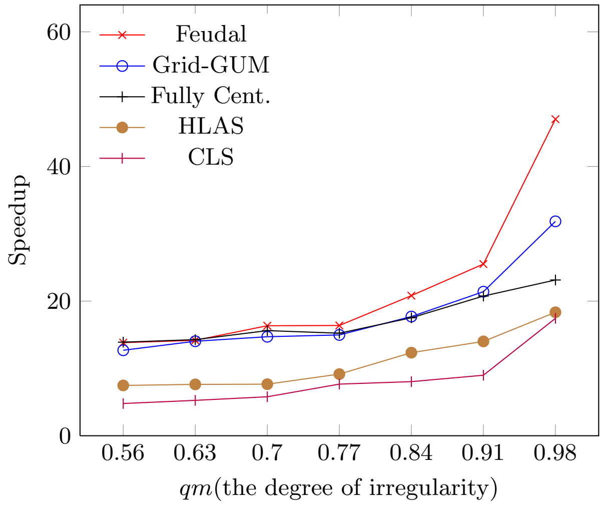


Fig. 6. Speedups for the UTS(q,7,3000,5ms) applications

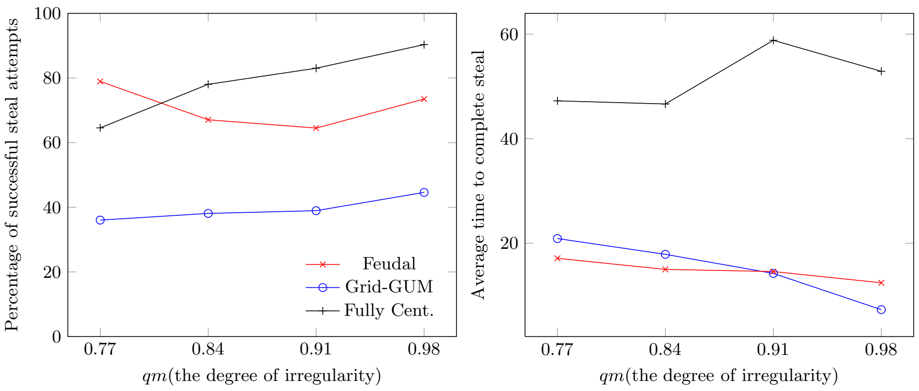


Fig. 7. Percentage of successful steal attempts (left) and average time it takes to successfully complete a steal (right) for the UTS(q,7,3000,5ms) applications

faster. Together, these two figures show why Feudal Stealing outperforms other algorithms. Central nodes are able to obtain relatively accurate load information, resulting in good selection of stealing targets, and the stealing messages are routed in such way that they quickly reach the targets with work, resulting in rapid response to the steal attempts. Grid-GUM and Fully Centralised stealing sacrifice one of these two features (accurate load information for Grid-GUM and rapid response to steal attempts for the Fully Centralised stealing) in order to make the other as good as possible.

Similar conclusions can be obtained when we look at the other examples of the UTS applications. For example, Fig. 8 shows the speedups of the UTS(15,q,3000, 10ms) applications, for $q \in \{0.062, 0.063, 0.064.0.065.0.066\}$. This represents

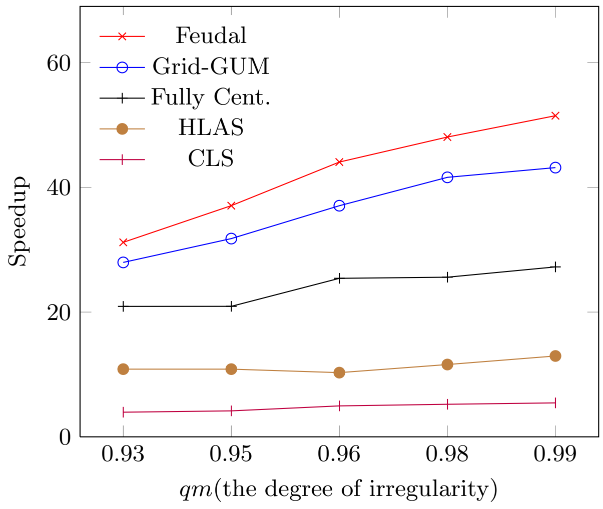


Fig. 8. The speedups of the UTS(q,15,3000,10ms) applications

applications with larger number of more coarse-grained tasks, with the high degree of irregularity ($qm \in \{0.93, 0.945, 0.96, 0.975, 0.99\}$). In these applications, the probability of a node having children is lower than for the UTS(7,q,3000,5ms) applications; however, each such node generates more children. Also, the sequential tasks are larger. We can see from the figure that Feudal Stealing and Grid-GUM give the best speedups (with Feudal Stealing performing best), and that both algorithms significantly outperform all other algorithms. The improvements in speedup of Feudal Stealing over the next best algorithm (Grid-GUM Stealing) range from 11% for $UTS(15, 0.062, 3000, 10ms)$ ($qm = 0.93$) to 20% for $UTS(15, 0.066, 3000, 10ms$ ($qm = 0.99$).

5 Conclusions and Future Work

In this paper, we have proposed a novel Feudal Stealing work stealing algorithm that uses a combination of centralised and distributed methods for obtaining dynamic system load information. We compared, using simulations, the performance of Feudal Stealing against the performance of algorithms that use centralised, fully-distributed, and a combination of both methods for obtaining dynamic load information. We have shown that Feudal Stealing outperforms all of these algorithms on heterogeneous distributed systems for the Unbalanced Tree Search benchmark, which models irregular divide-and-conquer (D&C) applications. Our previous results (reported in Janjic's PhD thesis [8]) have demonstrated that Feudal Stealing also outperforms state-of-the-art work-stealing algorithms that do not use load information on heterogeneous systems, and that it delivers comparable speedups to them on homogeneous systems and for regular D&C applications. Collectively, these results show that Feudal Stealing is the method of choice for load balancing the D&C applications on various different classes of systems (high-performance clusters of multicore machines, grids, clouds, skies etc.). Furthermore, we believe that this algorithm is also applicable to other classes of parallel applications (e.g. applications with nested data-parallelism and master-worker applications).

As future work, we plan to test the implementation of Feudal Stealing in a realistic runtime system (e.g. the Grid-GUM runtime system for Parallel Haskell [1], or the Satin system for distributed Java [16]) and to evaluate its performance on larger scale parallel applications. We also plan to incorporate information about the sizes of parallel tasks in the definition of the load for each node. This information has been shown to be important for selecting the task to offload in response to a steal attempt [9],. We envisage it can further improve stealing target selection, and hence speedup. Finally, we plan to introduce a measure of heterogeneity of a computing system, and to investigate "how heterogeneous" a system needs to be for Feudal Stealing to outperform other state-of-the-art work-stealing approaches.

Acknowledgments. This research is supported by European Union grants ADVANCE (IST-248828) and ParaPhrase (IST-288570), EPSRC grant HPC-GAP (EP/G 055181 and EU COST Action IC1202: Timing Analysis On Code-Level (TACLe).

References

1. Al Zain, A.D., et al.: Managing Heterogeneity in a Grid Parallel Haskell. Scalable Computing: Practice and Experience 7(3), 9–25 (2006)
2. Baldeschwieler, J.E., Blumofe, R.D., Brewer, E.A.: ATLAS: An Infrastructure for Global Computing. In: Proc. 7th Workshop on System Support for Worldwide Applications, pp. 165–172. ACM (1996)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM 46(5), 720–748 (1999)
4. Burton, F.W., Sleep, M.R.: Executing Functional Programs on a Virtual Tree of Processors. In: Proc. FPCA 1981: 1981 Conf. on Functional Prog. Langs. and Comp. Arch., pp. 187–194. ACM (1981)
5. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable Work Stealing. In: Proc. SC 2009: Conf. on High Performance Computing Networking, Storage and Analysis, pp. 1–11. ACM (2009)
6. García Zapata, J., Díaz Martín, J.: A Geometrical Root Finding Method for Polynomials, with Complexity Analysis. Journal of Complexity 28, 320–345 (2012)
7. Hammes, J., Bohm, W.: Comparing Id and Haskell in a Monte Carlo Photon Transport Code. J. Functional Programming 5, 283–316 (1995)
8. Janjic, V.: Load Balancing of Irregular Parallel Applications on Heterogeneous Computing Environments. PhD thesis, University of St Andrews (2012)
9. Janjic, V., Hammond, K.: Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In: Proc. CCGrid 2010: IEEE/ACM Intl. Conf. on Cluster, Cloud and Grid Computation, pp. 123–134 (May 2010)
10. Janjic, V., Hammond, K.: Using Load Information in Work-Stealing on Distributed Systems with Non-uniform Communication Latencies. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 155–166. Springer, Heidelberg (2012)

11. Neary, M.O., Cappello, P.: Advanced Eager Scheduling for Java-Based Adaptively Parallel Computing. In: Proc. JGI 2002: Joint ACM-ISCOPE Conference on Java Grande, pp. 56–65 (2002)
12. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: UTS: An Unbalanced Tree Search Benchmark. In: Almási, G.S., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
13. Ravichandran, K., Lee, S., Pande, S.: Work Stealing for Multi-Core HPC Clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011)
14. Van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In: Proc. PPoPP 2001: 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog., pp. 34–43 (2001)
15. Van Nieuwpoort, R.V., et al.: Adaptive Load Balancing for Divide-and-Conquer Grid Applications. J. Supercomputing (2004)
16. Van Nieuwpoort, R.V., et al.: Satin: A High-Level and Efficient Grid Programming Model. ACM TOPLAS: Trans. on Prog. Langs. and Systems 32(3), 1–39 (2010)