Using Load Information in Work-Stealing on Distributed Systems with Non-uniform Communication Latencies

Vladimir Janjic and Kevin Hammond

School of Computer Science, University of St Andrews, United Kingdom
{jv,kh}@cs.st-andrews.ac.uk

Abstract. We evaluate four state-of-the-art work-stealing algorithms for distributed systems with non-uniform communication latenices (Random Stealing, Hierarchical Stealing, Cluster-aware Random Stealing and Adaptive Cluster-aware Random Stealing) on a set of irregular Divide-and-Conquer (D&C) parallel applications. We also investigate the extent to which these algorithms could be improved if dynamic load information is available, and how accurate this information needs to be. We show that, for highly-irregular D&C applications, the use of load information can significantly improve application speedups, whereas there is little improvement for less irregular ones. Furthermore, we show that when load information is used, Cluster-aware Random Stealing gives the best speedups for both regular and irregular D&C applications.

1 Introduction

Work stealing [5], 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 proposed and implemented for systems with non-uniform communication latencies, that is for cloud- or grid-like systems [2,4,16,17], and for high-performance clusters of multicore machines [13]. However, most of these algorithms are tailored to highly *regular* applications, such as those using simple Divide-and-Conquer (D&C) parallelism. This paper considers how work-stealing can be generalised to *irregular* parallel D&C applications, so covering a wide class of real parallel applications. In particular, we compare the effectiveness of different work-stealing approaches for such applications, and describe improvements to these approaches that provide performance benefits for "more irregular" parallel applications. This paper makes the following main research contributions:

– We compare the performance of state-of-the-art work-stealing algorithms for highly-irregular D&C applications, providing insight into whether the "best" methods for regular D&C applications also perform well for irregular ones.
– We evaluate how well these algorithms could perform if they had access to *perfect* load information, i.e. how much speedup could be improved if this

information was available. This gives insight into whether load information can improve work-stealing and also tests the limits to these improvements.
– We investigate *how accurate* this load information needs to be to provide some benefit. Since it is impossible to obtain fully-acurate instantaneous load information for real distributed systems, this gives insight into whether work-stealing can benefit from less accurate load information.

We address these three issues using high-quality simulations in the SCALES simulator [8]. The decision to use simulations is driven by our goal of considering the theoretical limits to improvements that can be obtained by using fully-accurate load information and also to quantify the extent to which those limits can be approached using more realistic partial load information. This would not be possible using a real implementation, since we cannot instantaneously communicate load information between distributed machines. The general operation of the simulation has been verified against a real distributed system [8], so we have a high degree of confidence in its ability to predict scheduling performance. We stress that, in this paper, unlike other work [16,8,2], we are not concerned with the question of how to *obtain* accurate load information. Rather, we are interested purely in the *impact* of this information on load-balancing. A comparison of different heuristics for obtaining load information can be found in [8].

2 Work-Stealing on Systems with Non-uniform Latencies

We consider distributed clusters, where each cluster contains one or more (parallel) machines (that is a cloud- or grid-like server farm). Each machine in a cluster forms a processing element (PE) that can manage its own set of independent parallel tasks (which will normally be evaluated using lightweight multi-threading) Each PE has its own *task pool*, which records the tasks that are owned by that PE. When a PE starts executing a task, it converts it into a *thread* (which we will assume is fixed to one PE). Tasks can, however, be migrated between PEs. In a work-stealing setting, whenever a PE has no tasks in its task pool, that PE becomes a *thief*. The thief sends a *steal attempt* message to its chosen *target* PE. If the target PE 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, a negative response can be sent to the thief, who then deals with it in an appropriate way (either by initiating another steal attempt or by delaying further stealing). The main differences between the various work-stealing algorithms that we consider lie in the way in which thieves select targets, and in the way in which targets that do not have enough tasks respond to steal attempts. This paper considers the following four state-of-the-art work-stealing algorithms:

– *Random Stealing* [4] – Targets are chosen randomly, and targets also forward steal attempts to random PEs. This is used by e.g. Cilk [3] or GUM [15].
– *Hierarchical Stealing* – The PEs are organised into a tree (based on communication latencies). A thief first attempts to steal from all of its children

(which may recursively attempt to steal from their children); only if no work is found will it ask its parent for work. This algorithm is used by the Atlas [2] and Javelin [11] runtime systems for distributed Java.
- *Cluster-Aware Random Stealing (CRS)* – Local stealing (within a cluster) and remote stealing (outside of a cluster) are done in parallel. That is, a thief will attempt to steal from a random PE within its own cluster, and, in parallel, will attempt to steal from a remote PE. Targets always forward the steal attempt to a PE in its own cluster if they have no work. A very similar algorithm is used in the Satin [18] extension to Java, which provides primitivies for divide-and-conquer and master-worker parallelism[1].
- *Adaptive Cluster-Aware Random Stealing (ACRS)* – An improvement to the CRS algorithm, where thieves prefer to steal remote tasks from clusters that are nearer to them [17].

Note that the focus of these algorithms is on the selection of potential victims. This is the main issue for distributed systems with potentially high communication latencies, so that thieves obtain work quickly. Following the usual practice in work-stealing for divide-and-conquer applications, we assume that the *oldest* task from the task pool is sent to the thief in response to the steal attempt, and that the *youngest* task from the PE's task pool is chosen for local execution. This allows locality to be preserved, while large tasks are transferred over the network. Many other factors may also influence speedups, such as task queue locking [10] and identifying termination conditions [13,14]. A comparison of several other policies for task pool management can be found in [9].

3 Irregular Divide-and-Conquer Applications

This paper focuses on the irregularity in parallelism that arises from an unbalanced task tree, i.e. where some of the tasks created by a parallel task are sequential, but where others are themselves parallel. This kind of irregularity arises in many benchmarks for load imbalance as well as in many realistic applications that deal with irregular or unbalanced data-structures. For example, the *Unbalanced Tree Search* benchmark dynamically creates highly-unbalanced trees and then processes their nodes [12]; in the *Bouncing Producer and Consumer* benchmark [6], a producer creates a set of consumer subtasks, and then nominates one of these consumers as a new producer. Unbalanced task trees arise, for example, in real applications that trace particles through a 3D scene (e.g. the *Monte Carlo Photon Transport* algorithm [7], where the input data determines how unbalanced the task tree is). Implementations of the *Min-Max* algorithm, which prune a tree of game positions, also exhibit irregularity of this kind.

We have previously introduced a formal statistical measure for the degree of irregularity for such an application [8]. Intuitively, the more unbalanced the

[1] In Satin, a target that has no tasks to send to the thief returns a negative response, rather than forwarding the steal attempt, as here. We have found that our version performs much better in our context.

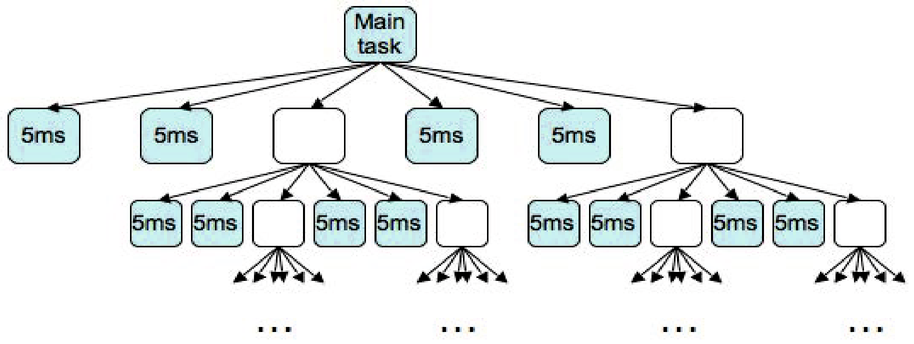


Fig. 1. Task graph for an example *DCFixedPar(6,3,5ms,T)*

task tree of such an application is, the more irregular is the application. Due to space constraints, we will use this intuitive "measure" of the irregularity of an application, without defining it rigorously here. Our main focus is on *divide-and-conquer applications with fixed parallelism*, denoted by *DCFixedPar(n,k,S,t)*. In such an application, every nested-parallel task creates n subtasks, where every k-th subtask is itself nested-parallel (and the others are sequential). Below some threshold t, all tasks are sequential with size S. Such applications are examples of irregular D&C applications, where k determines the degree of irregularity, and n determines the number of tasks. The larger k is, the more unbalanced is the task tree, and, therefore, the more irregular is the application. Figure 1 shows the task tree of an example *DCFixedPar(6,3,5ms,t)* application.

4 Using Load Information

For the work-stealing algorithms presented in Section 2, we can observe that the methods they use for selecting targets are partially (or, in the case of Hierarchical Stealing, fully) based on knowledge of the underlying network topology. They do not, however, depend on information about PE loads. Choosing targets in this way is acceptable for applications where the majority of tasks create additional parallelism (for example, *DCFixedPar*(n, k, S, t), where k is small). During the execution of such applications a thief that steals a task will itself become a potential victim for some other thief. This means that there are a large number of potential victims in most execution phases. This makes locating victims easy, even when it is done randomly. Indeed, Nieuwpoort et al. [16] show that very good performance can be obtained using these work-stealing algorithms, with CRS giving the best speedups for simple D&C applications (those corresponding to *DCFixedPar(*2,1*,S,t)* applications). This is because thieves can usually obtain work locally, and remote prefetching of work (via wide-area stealing) essentially comes for free, because the latency is hidden by executing locally obtained work.

The performance of these algorithms for irregular D&C applications is, however, not well understood. In these applications, most of the tasks may be sequential. This means that it is no longer true that almost every successful thief becomes a potential victim. Furthermore, in some execution phases the number of potential victims may be low, and high load imbalances may then exist. Locating the victims in a potentially large system can be hard if it is done randomly, and a thief may send many fruitless steal attempts over high-latency networks before it manages to locate a victim. Therefore, for irregular D&C applications, the CRS algorithm may not perform the best, and might indeed be outperformed by methods that do some kind of systematic search for targets, from closer to further ones (as with Hierarchical Stealing). In order to obtain good speedups, it may be essential to have some information about target loads, to minimise the time that thieves spend obtaining work. In order to investigate these issues, we first evaluate how basic work-stealing algorithms perform on highly irregular D&C applications, determining which of them gives the best speedups. We then investigate the extent to which these speedups could be increased if *fully accurate* load information is present and used in these algorithms; that is, if each thief, at the moment where it needs to select the stealing target, knows precisely how many tasks are in each PE's task pool. We, therefore, consider the following "perfect" work-stealing algorihms:

- *Perfect Random Stealing.* A target with non-zero load is chosen randomly.
- *Perfect Hierarchical Stealing.* A set of all PEs is organised into a tree. A thief checks the load of all of its children (where the load of a PE is the aggregate load of all of the PEs in its subtree). If a child with non-zero load exists, the steal attempt is sent to it. Otherwise, a thief tries to steal from its own parent. Whenever a target receives a steal attempt, if it has no work to send, it forwards the steal attempt using the same procedure.
- *Perfect Cluster-aware Random Stealing* (Perfect CRS). A thief attempts to steal in parallel from random local and remote PEs with non-zero load.
- *Perfect Adaptive Cluster-aware Random Stealing* (Perfect ACRS). This algorithm is similar to Perfect CRS, except that during the remote stealing, thieves prefer to steal from closer targets with non-zero load.
- *Closest-Victim Stealing* (CV). The closest target with work is chosen.
- *Highest-Loaded-Victim Stealing* (HLV). A thief steals from a target with the largest number of tasks.

Note that the last two algorithms do not have "basic" equivalents, since they depend on the presence of load information. We include them here because they represent fairly intuitive methods for selecting stealing targets in the presence of load information. We assume that a thief steals only one task at a time from a victim. While stealing more than one task may be beneficial where many tasks are sequential, for more regular D&C applications this can result in unnecessarily large amounts of work being transferred from the victim to the thief. Finally, we evaluate how the accuracy of load information relates to the performance of algorithms. In other words, we evaluate what happens if the load information is not completely accurate. This enables us to observe whether the load information needs to be fully accurate (which is impossible to obtain in the real word), or whether some approximation (which can be obtained by a heuristic) is enough.

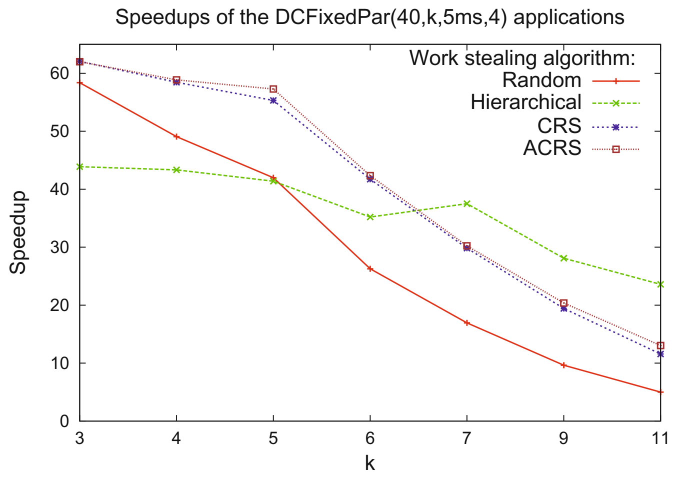
5 Experiments

All experiments were conducted using the SCALES simulator [8], which was developed for the sole purpose of testing the performance of work-stealing algorithms on parallel systems with non-uniform communication latencies. SCALES supports several popular parallelism models, such as divide-and-conquer, data parallel and master-worker. It independently simulates the load-balancing events for each PE, such as sending/forwarding steal attempts and the transfer of tasks between PEs. It also simulates the overheads for individual load-balancing events (such as sending steal attempts, packing and unpacking of tasks, executing tasks). SCALES has been shown to accurately estimate speedups under various work-stealing algorithms for realistic runtime systems [8].

In order to keep the number of experiments manageable, we use the same simulated system in all of our experiments. A number of experiments on other simulated systems can be found in [8], which confirm the conclusions found here. Our system consists of 8 clusters of PEs, with 8 PEs in each cluster. Clusters are split into two *continents* of 4 clusters each, with an inter-continental latency of 80ms. Each continent is split into two *countries*, with an inter-country latency of 30ms. Finally, each country is split into two sites, with an inter-site latency of 10ms. In the remainder of the paper, the PE that executes the main application task is the *main* PE; the cluster containing the main PE is the *main cluster*; and all other clusters are *remote* clusters.

5.1 Performance of the Basic Algorithms

For our first set of experiments with irregular D&C applications, we focus on the *DCFixedPar(40,k,5ms,4)* applications. The size of sequential subtasks is set to *5ms* to produce an application with fine-grained tasks. Note that, as k increases, the applications become more irregular. Figure 2 shows the speedups that we obtained under the basic algorithms. We observe that the CRS and ACRS algorithms give the best speedups for more regular applications. However, as the applications become more irregular (for $k > 6$), we observe that Hierarchical Stealing starts to outperform both CRS and ACRS. The reason for this is that Hierarchical Stealing gives much more uniform work distribution than CRS, where most of the tasks are executed by PEs from the main cluster, and where PEs in the remote clusters are mostly idle. For highly irregular applications we observe that CRS and ACRS deliver poor speedups of 10-12, whereas Hierarchical Stealing still manages to deliver good speedups of 25-30. This experiment reveals two things. Firstly, it shows that the situation for irregular applications is less clear cut than for regular ones, where the CRS algorithm constantly delivers the best speedups. We can see that for less irregular applications, CRS is still the best choice. For highly-irregular ones, however, Hierarchical Stealing is better. Secondly, as the irregularity of the applications increases, speedups decrease sharply for most of the algorithms. The exception is Hierarchical Stealing, which still manages to deliver good speedups, even for highly-irregular applications. A similar situation exists for other *DCFixedPar* applications. If we increase the



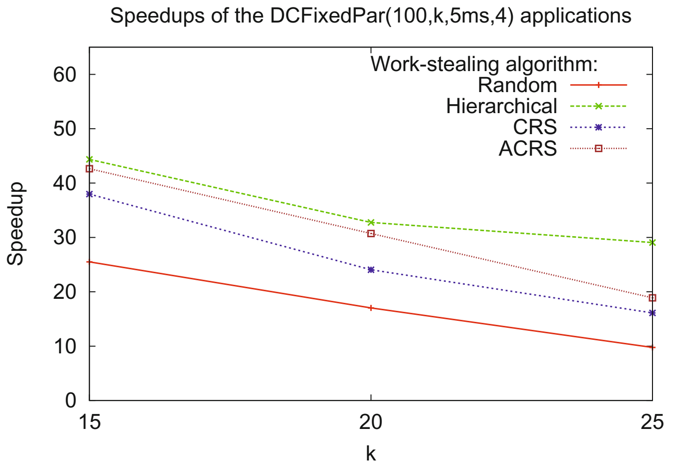
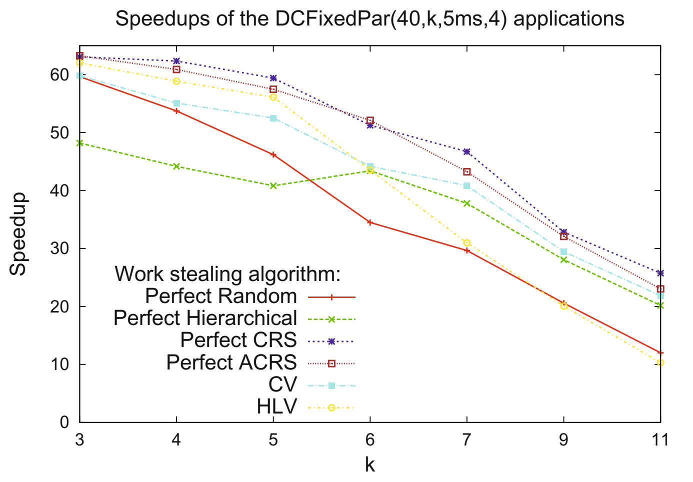


Fig. 2. Speedups under the basic algorithms for the *DCFixedPar(40,k,5ms,4)* applications (above) and *DCFixedPar(100,k,5ms,4)* applications (below)

number of subtasks from 40 to 100, as shown in the bottom of Figure 2, we obtain similar results. Since all the *DCFixedPar(100,k,5ms,4)* applications are highly irregular, Hierarchical Stealing gives the best speedups.

5.2 Performance of the Perfect Algorithms

We now consider the perfect algorithms. Figure 3 shows the corresponding speedups and relative improvements for the *DCFixedPar(40,k,5ms,4)* applications. It is clear that CRS and ACRS give the best speedups. Since all thieves know exactly where to look for work, thieves from remote clusters manage to steal a lot of work, so Perfect CRS does not suffer from the same problem as the



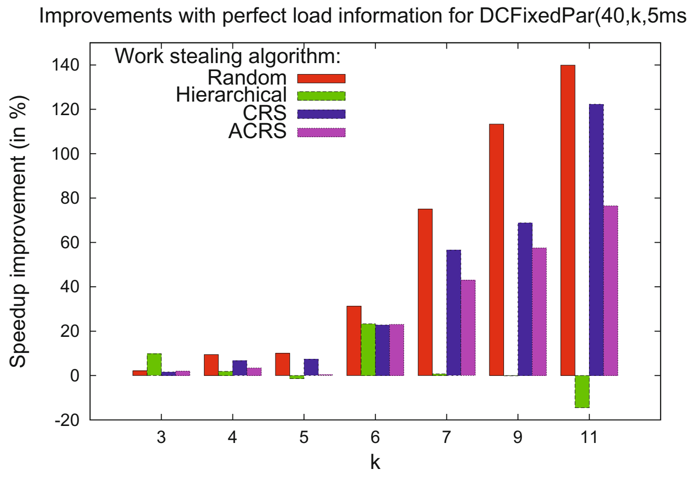


Fig. 3. Speedups (above) and relative Improvements in speedups (below) using perfect load information for the *DCFixedPar(40,k,5ms,4)* applications

basic version of CRS. CV and Hierarchical Stealing perform similarly to CRS and ACRS for more irregular applications, but are consistently worse. Random Stealing and HLV are considerably worse than these four algorithms. The bottom half of Figure 3 shows the improvement in speedup for the perfect versions. We observe small improvements for less irregular applications, but very good improvements (70%-120%) for highly-irregular applications for Random Stealing, CRS, and ACRS. For Hierarchical Stealing, we only observe very small improvements. In some cases, Hierarchical Stealing without load information is actually better than with fully accurate load information. Finally, Figure 4 shows corresponding speedups for the *DCFixedPar(100,k,5ms,4)* applications. We once again observe that CRS and ACRS give the best speedups.

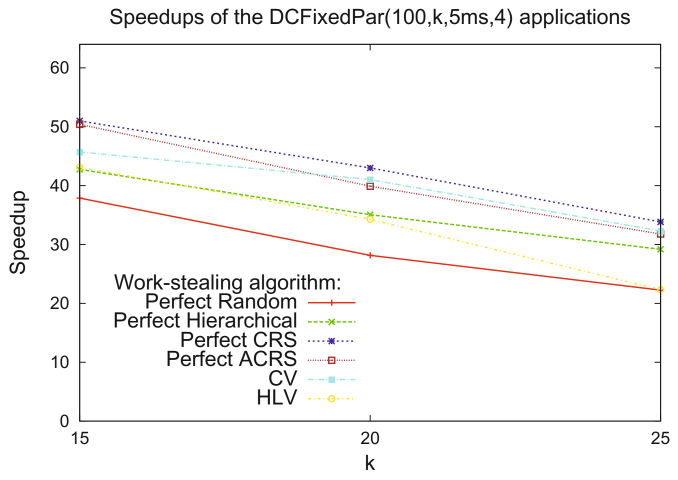


Fig. 4. Speedups under the perfect algorithms for *DCFixedPar(100,k,5ms,4)*

5.3 How Accurate Does Load Information Need to Be?

In the previous section we have seen that the use of fully-accurate load information brings significant speedup benefits when using the Random, CRS and ACRS algorithms for highly-irregular D&C applications. With fully accurate load information, CRS and ACRS give the best speedups both for less irregular and for more irregular applications. Coupled with the fact that these algorithms are also the best for regular applications, it seems that they are the algorithms of choice for work stealing on distributed systems, *provided* we can obtain a good approximation of PE loads during the execution of the application.

In this section, we will focus on the CRS algorithm, since it gives similar results to ACRS, and is easier to implement. The natural question to ask is *how accurate* load information needs to be for the load-based CRS algorithm to obtain good speedup improvements over the basic version. It is obvious that obtaining *perfect* load information (where each PE has fully accurate information about the load of all other PEs) is simply impossible in real systems. We, therefore, now investigate the extent to which an application's speedup changes when load information becomes outdated to some extent.

Let us denote the set of all PEs by $\{P_1, \ldots, P_n\}$, and the load of PE P at time t by $L(P,t)$. The load information $I(Q,t)$ that a PE Q has at time t can then be represented as a set $\{(P_i, L(P_i,t_i)|P_i \in \{P_1,\ldots,P_n\}\}$ of PE-load pairs, where the load of P_i was accurate at time t_i. For perfect information (denoted by PI), $t_i = t$ for all PEs, so $PI(Q,t) = \{(P_i, L(P_i,t)|P_i \in \{P_1,\ldots,P_n\}\}$. We introduce the idea of *outdated information with a delay of k time units* by

$$OI(Q,t,k) = \{(P_i, L(P_i, t - lat(Q,P_i) - k)|P_i \in \{P_1,\ldots,P_n\}\},$$

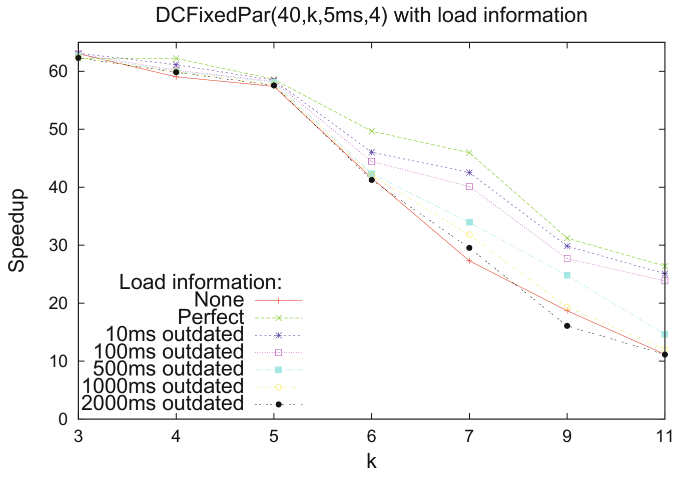


Fig. 5. Speedups with outdated load information when using CRS

where $lat(Q, P_i)$ is the communication latency between PEs Q and P_i. This represents a more realistic setup, where the age of the load information that PE Q has about PE P depends on the communication latency between P and Q, and also on the fixed delay k (in time units) of the delivery of such information. In an even more realistic setup, k could be a function, rather than a constant, so this delay could be different for different PEs.

Figure 5 shows the speedups of the *DCFixedPar(40,k,5ms,4)* applications under the CRS algorithm using outdated load information with various delays (in *ms*). As expected, speedups decrease as the load information become more outdated. However, we can still observe good speedups when the information is relatively recent (with a delay of up to 100ms). With a delay of 500ms, the information is still usable (i.e. speedups are notably better with than without load information). For large delays of 1000ms and 2000ms, the load information becomes practically unusable. Note that the load information delays in our experiments are rather high with respect to the sizes of sequential tasks. In applications with coarser-grained tasks, load information with the same delay will have less impact on the application's speedup, since PE loads will change more slowly. We therefore conclude that significant speedup improvements can be achieved for the CRS algorithm not only with perfect information, but also with relatively recent load information (which it is possible to obtain in real systems). Only if load information is completely outdated does it becomes unusable.

6 Conclusions and Future Work

In this paper, we have investigated the performance of four state-of-the-art distributed work-stealing algorithms (Random Stealing, Hierarchical Stealing, Cluster-aware Random Stealing and Adaptive Cluster-aware Random Stealing) for irregular Divide-and-Conquer parallel applications. We have shown that, surprisingly, for highly-irregular applications Hierarchical Stealing delivers the best speedups. This differs from regular D&C applications, where previous work [16,8] has shown that CRS delivers the best speedups.

We have also investigated the speedup improvements that can be made if accurate system load is available. Our results show that perfect load information brings significant speedup benefits for highly-irregular D&C applications. Surprisingly, for less irregular ones, the availability of perfect load information is not too significant. Our results show that if some load information is available, then the CRS and ACRS algorithms deliver the best speedups for irregular D&C applications. We have, moreover, shown that in order to obtain good speedups with these algorithms, it is not necessary for the load information to be perfect: a good approximation also suffices. This clearly indicates that the CRS and ACRS algorithms are the best ones to choose for work-stealing on distributed systems with non-uniform and potentially high communication latencies.

In future, we plan to extend the CRS algorithm with mechanisms for approximating load information. Several good mechanisms for obtaining good approximations of load information already exist, e.g. Grid-GUM [1] uses a fully-distributed mechanism of load information propagation, and Atlas uses a Hierarchical mechanism [2]. We intend to consider a combination of fully-distributed and fully-centralised approaches, where load information is centralised within the low-latency networks, and distributed over high-latency ones. We also intend to investigate heuristics for estimating the number of tasks that should be transferred in one steal operation and the impact that sending more than one steal attempt has on various algorithms.

Acknowledgments. This research is supported by European Union grants SCIEnce (RII3-CT-026133), ADVANCE (IST-248828) and ParaPhrase (IST-288570), and by EPSRC grant HPC-GAP (EP/G 055181).

References

1. Al Zain, A.D., Trinder, P.W., Michaelson, G.J., Loidl, H.-W.: 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., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. In: Proc. PPoPP 1995: ACM Symp. on Principles and Practice of Parallel Prog., pp. 207–216 (1995)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM 46(5), 720–748 (1999)
5. 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)

6. 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)
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 (2011)
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. Michael, M.M., Vechev, M.T., Saraswar, V.A.: Idempotent Work Stealing. In: Proc. PPoPP 2009: 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog., pp. 45–54 (2009)
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.F., 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. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based Global Load Balancing. In: Proc. PPoPP 2011: 16th ACM Symp. on Principles and Practice of Parallel Prog., pp. 201–212 (2011)
15. Trinder, P.W., Hammond, K., Mattson Jr., J.S., Partridge, A.S., Peyton Jones, S.L.: GUM: A Portable Parallel Implementation of Haskell. In: Proc. PLDI 1996: ACM Conf. on Prog. Lang. Design and Implementation, pp. 79–88. ACM (1996)
16. 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)
17. Van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Kielmann, T., Bal, H.E.: Adaptive Load Balancing for Divide-and-Conquer Grid Applications. Journal of Supercomputing (2004)
18. Van Nieuwpoort, R.V., Wrzesińska, G., Jacobs, C.J.H., Bal, H.E.: Satin: A High-Level and Efficient Grid Programming Model. ACM TOPLAS: Trans. on Prog. Langs. and Systems 32(3), 1–39 (2010)