Scalable Work-Stealing Load-Balancer for HPC Distributed Memory Systems

Clement Fontenaille[1,2](✉), Eric Petit[3], Pablo de Oliveira Castro[1], Seijilo Uemura[1], Devan Sohier[1], Piotr Lesnicki[2], Ghislain Lartigue[4], and Vincent Moureau[4]

[1] Li-PaRAD, University of Versailles, Versailles, France
clement.fontenaille@uvsq.fr
[2] Atos-Bull, Paris, France
[3] Intel Corporation, Santa Clara, USA
[4] CORIA-CNRS, University of Normandie, Saint-Étienne-du-Rouvray, France

Abstract. Work-stealing schedulers are common in shared memory environments. However, large scale distributed memory usage has been limited to specific *ad-hoc* implementations preventing a broader adoption. In this paper we introduce a new scalable work-stealing algorithm for distributed memory systems as well as our implementation as the TITUS_DLB library. It is based on Kleinberg's small-world graph. It allows to control the communication patterns and associated runtime overheads while providing efficient heuristics for victim selection and results routing. To validate our approach, we present the DLB_Bench benchmark which emulates arbitrary workload distribution and imbalance characteristics. Finally, we compare TITUS_DLB to the *ad-hoc* solution developed for the YALES2 computational fluid dynamics and combustion solver. We achieve up to 54% performance gain over thousands of cores.

1 Introduction

In high-end HPC machines, the current architecture trend is to dramatically increase the number of cores. Managing large scale concurrency is a challenge for many HPC applications and runtimes, which eventually hit a *scalability wall*.

Load balancing systems optimize workload distribution and resource usage to improve the scalability of unbalanced applications.

Work-stealing, as presented in [7,8], is an asynchronous distributed decentralized dynamic load balancing algorithm.

In order to implement scalable work-stealing for large scale distributed memory systems, we take into account an overlooked limitation of the classical victim selection strategy: random victim selection may trigger the connexion of all possible pairs of processes, and the expected memory and time overhead limits scalability.

© Springer Nature Switzerland AG 2019
G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 146–158, 2019.
https://doi.org/10.1007/978-3-030-10549-5_12

In this paper, we introduce the TITUS_DLB library: a new scalable approach to the work-stealing algorithm for dynamic load-balancing targeting large scale computations on distributed memory systems.

We control runtime overheads by constraining the communication pattern of the work-stealing algorithm to a scalable, low diameter, family of overlay networks: the Kleinberg's small-world sparse neighboring graph class. This overlay network provides short paths, allowing for efficient scheduling. Moreover, to return the data resulting from the execution of relocated computation, we propose a scalable results routing strategy.

Using a synthetic benchmark which emulates arbitrary workload characteristics and distribution, we study the efficiency of our scheduler in various configurations up to 3584 cores. We also evaluate the performance of our implementation compared to a hierarchical work-sharing approach in use in the ill-balanced computation of detailed chemistry from the YALES2 computational fluid dynamic and combustion application and achieve up to 54% speedup at 3584 cores.

2 Context and Objectives

We address load balancing using a relocatable tasks representation of a given computation.

Tasks are indivisible self-contained units of sequential work. They consume exclusive input data and produce results data. Relocating a task requires relocating its input data. Each task is spawned by its owner process initially holding the required input data. We do not address tasks dependencies. A task is completed when the produced results have been stored in its owner's memory. The execution time of each task is presumed to be irregular and unpredictable.

Parallel work resolution is completed as soon as the global set of tasks has been executed and the termination detection algorithm has converged.

We are interested in minimizing the parallel resolution time. We measure the time spent between the beginning of parallel work and the termination detection on each process. The maximum of these measured times is the parallel resolution time. Assuming homogeneous processor capabilities, we deduce the parallel efficiency against an hypothetical resolution time with perfect scheduling and zero overhead, i.e. the average work per process.

Blumofe *et al.* [5] introduce the *work first principle*: they observe that the available parallelism in parallel programs is vastly superior to the parallelism exploited for its execution, *i.e.* scheduling efficiency is driven by the work scheduling overhead, rather than that of the critical path scheduling overhead. As a first step towards an efficient and scalable distributed task scheduling algorithm for such applications, we address the problem of scheduling a set of tasks available for computation. All tasks are spawned before parallel resolution begins, and the critical path of the scheduled computation is the resolution time of the longest task to solve, which is assumed to be a small fraction of the average work per process.

The presented implementation of our algorithm does not yet support the benchmarks generally adopted for dynamic task scheduling (see Sect. 6.1), and

focuses on the elaboration of a scalable communication pattern for this specific problem.

2.1 Context

Work-stealing approaches define two process states: processes who own work are workers, while the others are thieves. Thieves attempt to acquire work from workers using a work stealing protocol.

Following the *work first principle* [8], we attempt to minimize the amount of time spent by workers on non-working activities, and move the scheduling overhead to thieves. Work stealing protocols have been proposed which rely on RDMA to access task data and operate load balancing without affecting the execution of tasks on the worker's end [7,15].

In [22], Woodall *et al.* outline an important limitation of RDMA capable hardware: the first communication between two processes (the connection) incurs a much longer response time than the subsequent communications as well as some memory overhead. Amortizing this pair connexion overhead is a necessity for the elaboration of a scalable distributed algorithm.

We use an overlay network to constrain the communication pattern of our scheduler in order to control and amortize the overhead of RDMA connexion. As in [14,17,20], work-stealing is local to a thief's neighborhood as work spreads among thieves through the edges of the overlay network.

2.2 Work-Stealing Algorithm Description

A worker is a process that locally holds work. A worker manages two sets of local tasks: tasks are executed from the private set and thieves acquire tasks from the shared set. When one of these sets is empty, the worker re-balances them. Processes do not hold any information about tasks spawned by other processes nor about the global set of tasks.

A thief is a process that holds no work. Termination detection is performed before each theft attempt. The thief then selects a victim, as discussed in Sect. 4 and performs the work stealing protocol. When an attempt succeeds, the theft policy selects a number of tasks from the remote set of tasks to relocate.

In the studied context, the termination detection protocol is a non-blocking barrier: when a process detects that all its owned tasks have completed, local completion is reached and the barrier is entered. If local completion has been reached, the process checks the advancement of the termination detection barrier before each theft.

3 Related Work

Static load balancing approaches compute a balancing strategy before computation. These approaches do not apply to unpredictable workloads, and are subject to system noise at scale.

Dynamic load balancing redistribute workload during computation [8,20]. It is a well studied topic and an important part of task-based runtime systems, which can use both workload-specific and platform-specific information to provide portable scheduling performance.

Maintaining a centralized knowledge of the load has a limited scalability and entails an uneven usage of network leading to contention [2]. Hierarchical approaches [2,14] alleviate this issue by distributing the load balancing responsibility across a hierarchy of master processes. By contrast, work-stealing is a decentralized scheduling algorithm which principles are theoretically sound and have been well studied for shared memory execution models [8,21]. Cilk [8], X-Kaapi [9], and Intel TBB [13] are a few examples which all feature work-stealing.

Many approaches for distributed memory systems using work-stealing have been proposed [7,15,17,20]. They perform very well at spreading work across a large distributed memory system, but do not address pair initialization overheads. They rely on hybrid programming using threads, alleviating the issue, or have been tested with limited scalability or on very specific use cases such as the GRAPH500 BFS [3] or UTS [18] in which tasks do not produce individual results.

Charm++ [1] supports a variety of dynamic scheduling policies, and allows for the composition of tuned *ad-hoc* solutions, making each solution designed with specific heuristic for a given application.

ADLB [14] or YALES2 introduce a hierarchy of basic working actors (threads and/or processes) and scheduling decision makers. Such hierarchical work-sharing is the most efficient approach in use in large scale HPC applications today. However, these approaches approaches may yield uneven resource usage, over-synchronization (jitter on the higher level of the hierarchy impacts overall performance), and require careful tuning and adaptation at large scale [14].

In this paper, we present a general-purpose non-hierarchical scalable approach to load-balancing that spreads the scheduling overhead and related network usage among the distributed system. Our algorithm uses a scalable overlay network to constrain the victim selection with interesting properties discussed in Sect. 4.

4 Work-Stealing on Smallworld Graph

In this section we discuss how Kleinberg's small-world graphs [11] are a good candidate for our constrained work-stealing algorithm.

They are built on top of a regular spanning lattice, *e.g.* a two-dimensional grid, that defines $D(u, v)$ as the lattice distance between any two nodes (u, v) in the system. Random edges are added to the spanning lattice and called shortcuts, resulting in interestingly low diameter graphs. Figure 1a shows an example of such graph.

Graph generation time and memory overhead is $O(d * |V|)$ as long as $d << |V|$. The graph representation can be scattered across processes, resulting in memory overhead of $O(d)$ per process.

Using such a graph $G = \{V, E\}$ to constrain victim selection in a work stealing protocol allows to connect a large number of nodes using a low and constant number of connections for each process: the degree d of the graph.

J. Kleinberg shows that G has a small diameter, $\delta = O(\log_d(|V|))$, with high probability. In other terms, it exists with high probability a path of length at most $\log_d(|V|)$ between any pair of nodes. Intuitively, this property allows work to spread efficiently in the graph.

4.1 Work Spreading

Work Reachability Criterion. Consider a worker u and a thief v at a distance Δ. The minimum number of thefts required for the worker's tasks to reach the thief is Δ. Assuming that a proportion p of the remote tasks are stolen at each successful theft, the minimum number of available tasks on u for at least one task to reach v is $O(p^{\Delta})$.

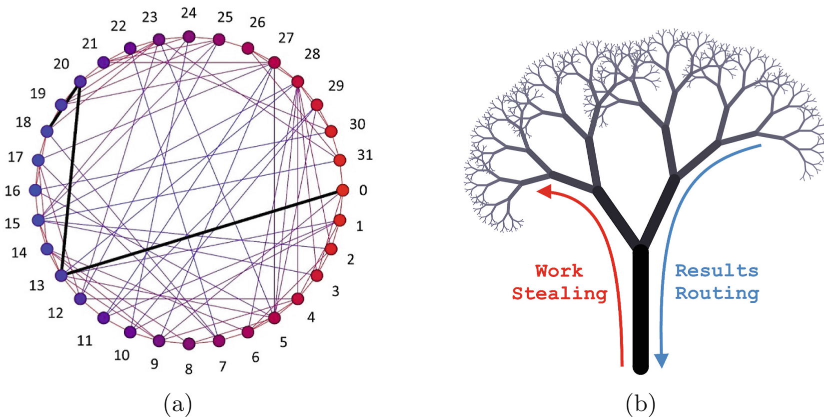

Fig. 1. (a) Example of small world graph and the generated route from node 0 to node 18 (b) Illustration of work fragmentation and results coarsening

In particular, if one worker owns all the tasks W, ensuring that all processes may participate in the computation requires that the amount of parallel tasks is at least $Wp^{\Delta} > 1 \Leftrightarrow \Delta < \log_{p^{-1}} W$.

To cope with workloads polynomial in the size of the system, $W = O(|V|^{\alpha})$ with $\alpha > 1$, a suitable overlay network must have low diameter $\Delta = O(\log |V|)$ which is satisfied.

Resilience to Jitter. The work-stealing algorithm allows spreading work through disjoint path in parallel without synchronization. J. Kleinberg shows that multiple short paths exist in such a small-world graph [12]. As such, while an homogeneous random source of jitter impacts the efficiency of a number of thefts, it is unlikely to globally slow down the work spreading process.

Data Locality. Kleinberg's graphs are randomly generated graphs built on top of a notion of distance which can be mapped to the actual hardware network topology. They are more heavily clustered than simple random graphs because of a bias toward short-length connections in the system, providing optimal route length for a simple routing strategy presented in section.

Perarnau and Sato [19] observe that given latency estimates, using a similar bias when selecting a victim for a theft promotes low-latency thefts and obtain significant performance gains.

In the experiments presented in Sect. 5, the small-world graph is built on top of the 1-dimension lattice formed by the MPI rank numbering. Assuming that the rank numbering, often based on *hwloc* [6], takes into account the physical distance in the network, a higher distance in terms of this lattice distance is likely to correspond to an equivalent or higher communication latency.

4.2 Results Routing and Coarsening

At each routing step the message is forwarded to the closest neighbor - in terms of lattice distance - to its destination [11]. J. Kleinberg shows that routes are found shortest when the probability for two nodes u, v to be connected by a shortcut is proportional to $D(u, v)^{-p}$, with p the dimension of the lattice. Figure 1a shows an example of routing. We measured that our implementation of this strategy finds routes through a 10000 nodes 64-regular random small-world graph with an average of 2.87 jumps, with maximum length of 6, over a million of random source-destination tuples.

After each theft, work is fragmented in two chunks. After computing, each chunk's resulting data has to be routed back to their owner. This effect is represented graphically in Fig. 1b. In order to mitigate congestion due to the sheer amount of routed chunks, we leverage the following routing properties:

- Whenever any two sets of resulting data belonging to the same owner are routed to the same node, they will be routed through the same nodes for the remainder of their routes.
- As sets of results from the same owner get closer to their destination, the probability to be routed to the same nodes increase.

Our results routing protocol aggregates buffered results chunks by next hop in route. As a consequence, the number of communications is asymptotically smaller than the number of chunks to route (i.e. the number of successful theft performed to balance load).

4.3 Memory and Communication Management

We use GASPI [10], an explicit Partitioned Global Address Space - PGAS - approach: accesses to remote memory locations are issued through traditional communication function calls. GASPI communication phases can occur in a traditional MPI program. It offers fine grained one-sided communication and a notification mechanism, allowing a truly asynchronous implementation.

Each process allocates dedicated memory segments which can be remotely accessed through read, write, atomic, and collective operations. A worker's set of shared tasks is stored in the *TASK* segment. A *TMP* segment receives task resulting data as they are computed. The *RESULTS* segment receives results chunks as they are routed and implements a simple lock-based remote memory reservation protocol. Metadata are stored on each segment in order to implement the related functionalities.

The set of private tasks is stored in a private memory buffer. Tasks are executed last-to-first, and balancing the sets of private and shared tasks is a local memory copy. Accessing the *TASK* segment of a given process requires obtaining a lock through a remote atomic compare and swap operation. If such a lock attempt succeeds, the thief performs two remote read operations. The first read operation acquires metadata information. Then, using this information the remote set of shared tasks is copied to the local *TASK* segment. The lock ownership on the remote *TASK* segment is then transfered to the victim. Workers check the state of the *TASK* segment between the execution of tasks, if a theft occurred and the lock ownership has been transfered, the set of shared tasks is empty and all data have been copied. half the set of local private tasks is then moved to the *TASK* segment, metadata are updated, and the lock is released. In Sect. 6.1, we provide hints for refining this strategy.

5 Use Cases and Experiments

We ran experiments on the Myria cluster at CRIANN, a Tier-2 computing center. Each compute node has 28 Intel Broadwell Xeon cores. Nodes are connected with Intel Omni-Path. We run on a maximum of 3584 cores on 128 compute nodes.

Pair Initialization Connection. In the presented experiments, we generate small world graphs with an average degree of $4log_2(|V|)$. It is an arbitrary choice - among those which satisfy $d << |V|$ - which implications at scale should be investigated further. We establish all connections allowed by the small-world overlay network and measure a 18 s of pair connection time at 3584 cores. This does not represent a significant overhead for typical HPC applications. This pair connection time is excluded from the presented performance.

5.1 DLB_Bench: A Synthetic Proto-Benchmark

We introduce DLB_bench, a dynamic load balancing benchmark that allows us to assess the performance of our scheduler against arbitrarily challenging initial work distribution. In this paper, we spread 70% of the tasks among 10% of processes (Fig. 2) following a gaussian distribution.

Due to our small-world overlay network, some processes are not connected to any work owner and most resulting data have to be routed to a small and clustered set of ranks.

Table 1 presents the various workload profiles considered in this study.

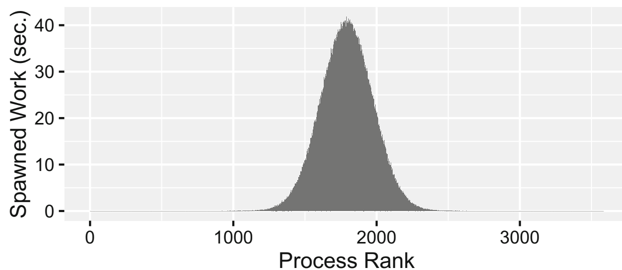

Fig. 2. DLB_Bench: workload distribution

Table 1. DLB_Bench: workload profiles

Average work per process	Average problem size per process	Tasks per process	Shared task segment size	Cycle/byte
10 s	4, 16, 32 MB	2 k, 8 k, 16 k	2, 8, 16 MB	750, 3 k, 6 k
5 s	2, 8, 16, 32 MB	1 k, 4 k, 8 k 16 k	1, 4, 8, 16 MB	375, 1.5 k, 3 k, 6 k
1 s	1, 4, 8, 16 MB	500, 2 k, 4 k, 8 k	0.5, 2, 4, 8 MB	150, 600, 1.2 k, 2.4 k

Workloads and Efficiency. Table 1 presents the workloads simulated using DLB_Bench. It is a weak-scaling experiment: overall problem size and overall work duration scales with the number of cores. Per process values are given and are referred to using average work (W/n) and problem size per process (S/n). We measure the efficiency of our load-balancer for each of the 11 selected workload profiles up to 3584 cores, and present the performance of the median execution - in terms of parallel efficiency - out of 5 repetitions. We did not observe significant performance variability for workload profiles where work per process is 5 and 10 s.

Figure 3 presents the parallel efficiency for the simulated workload characteristics of Table 1.

Down to 5 s of average work per core, our load balancing strategy generally shows high efficiency. As expected, a performance loss appears with low arithmetic intensity problems due to data transfer time overhead. When scheduling 1 s of average parallel work, the scheduling overhead becomes significant. To achieve more than 75% parallel efficiency on the studied workloads up to 3584 processes, we observe that the presented implementation requires both more than 1 s of execution time and a minimum of about 400 cycles per byte arithmetic intensity. Further investigation, which we do not present in details due to lack of space, suggest that these figures can be improves by suggestions presented in Sect. 6.1.

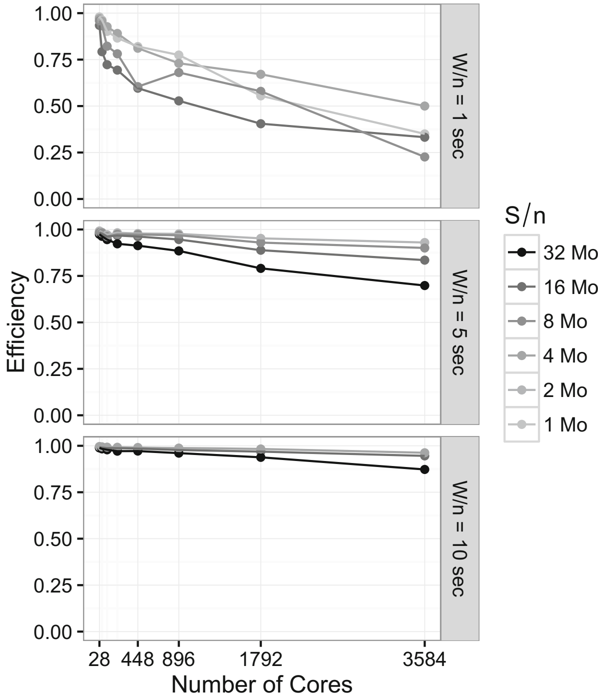

Fig. 3. DLB_Bench weak scaling efficiency

5.2 YALES2 Multi-physics Solver

YALES2 is a scalable multi-physics solver for HPC developed at CORIA. The code features many different solvers, collaborations with several academic laboratories and industrial partners, and it is used to solve large scale HPC problems in industrial companies such as SAFRAN, SOLVAY or ADWEN. It aims at modeling reactive flows in complex burners from primary atomization of the liquid fuel up to pollutant production. Detailed chemistry, in which tens of species are transported and react with each other, has gained a lot of interest due to the enabling available computational power. However, combustion is a localized and dynamic phenomenon that occurs in thin reaction zones at the sub-millimeter scale. Integrating the stiff chemical reactions requires few data and incurs high arithmetic intensity in the reaction zones, entailing an unbalanced workload.

YALES2 features a scalable *ad-hoc* task-based hierarchical work-sharing scheduler. Processes are attributed to a group using workload estimation based on previous iterations in order to provide approximated inter-group load balancing. However, the quality of this approximation drops at high number of cores and for very dynamic use-cases where reaction zones prediction is hard. At group level, processes operate a round-robin master/slave scheduling policy: masters distribute their work to their group and pass on the master token.

Table 2. Preccinsta: Workload profiles at 3584 cores

Mesh size	Average work per core range over 10 iterations	Problem size per core	Tasks per core	Shared tasks segment size	Average cycle/byte
14 M	0.15–0.23 s	0.28 MB	655	448 KB	1484
110 M	0.47–0.65 s	2.2 MB	5171	448 KB	550

Despite its high scalability, it does not satisfy the *work-first principle* [8]: workers pro-actively distribute their work among processes in their group and communication volume scales proportionally to the amount of work for any workload profile and imbalance.

We interfaced our approach in place of this built-in load-balancer, allowing us to test and evaluate our strategy in a real-life application at large scale, and demonstrate its efficiency empirically.

Workload. Figure 4 presents a typical workload distribution of an iteration of the studied experiment. Compared to the DLB_Bench workload presented in Fig. 2, work is less imbalanced, and initial task owners are distributed more evenly across the system. While the number of tasks spawned on each process is roughly the same and all the tasks carry the same amount of input data, the source of the imbalance is the unpredictability of each task's execution time.

Table 2 shows the main characteristics of scheduled workload. There is little variation in the amount of work and relative imbalance from one iteration to another.

Experimental Results. Figure 5 presents the performance of TITUS_DLB compared to the original load balancing strategy implemented in Yales2. Starting with a 14 million elements mesh, we refine the mesh in order to obtain a 110 millions elements test case. The total amount of work scales with the number of elements. Using these two use cases, we perform a strong scaling experiment up to 3584 cores. We run the first 10 time step iterations of the simulation, and measure the time spent in the chemistry simulation phase. In order to exclude

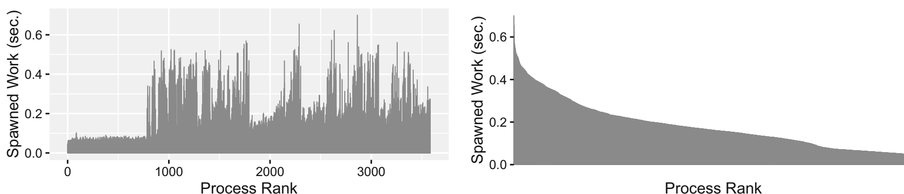

Fig. 4. Preccinsta: work distribution (Sorted)

pair initialization time for the original scheduler, we exclude the first iteration from these results. We compute parallel efficiency and speedup over the original scheduler from the sum of these times for the other 9 iterations.

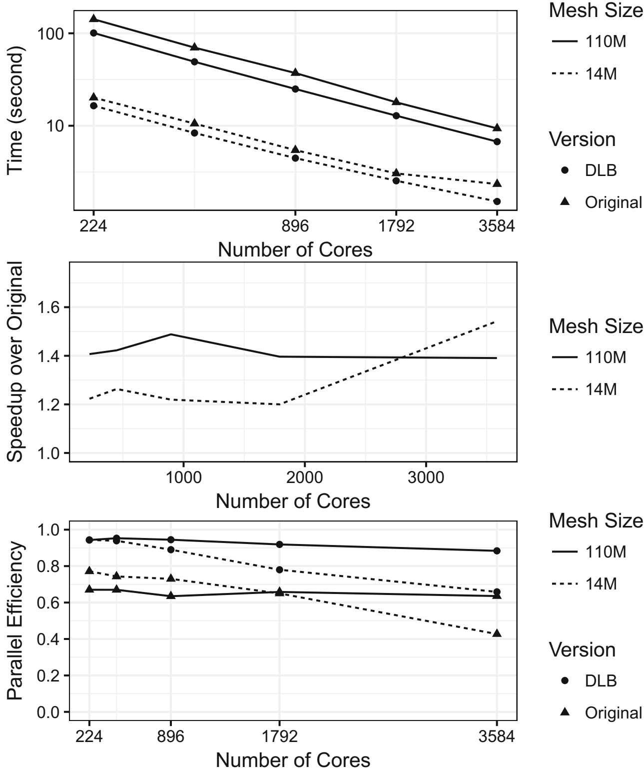

Fig. 5. Compared strong scaling performances of TITUS_DLB against Yales2's built-in dynamic scheduler for two mesh sizes

In all tested configurations, TITUS_DLB outperforms the original dynamic scheduler significantly, and up to 54% at 3584 core. The best efficiency achieved by the built-in dynamic scheduler on 3584 cores is 65% on the 110 M element case. TITUS_DLB achieves 88% efficiency, speeding up execution by 39%.

6 Conclusion

We propose a dynamic load-balancing strategy for large scale computations which we successfully demonstrate on a coupled chemistry and CFD simulation.

The small-world based communication pattern we introduce allows an efficient implementation of the work-stealing algorithm for large scale distributed computations.

The TITUS_DLB Library and DLB_Bench are released under open-source LGPL3.0 license at https://github.com/EXAPARS/TITUS.

6.1 Future Work

We are interested in developing an analytical model to provide a theoretical analysis of the presented approach.

We intend to extend the scope of the presented approach by allowing tasks to be spawned dynamically, providing support for a wider variety of problems, such as dynamic tasks decomposition and fork-join parallelism, allowing us to compare the performance of the proposed approach to existing dynamic task scheduling strategies through usually adopted benchmarks such as UTC and BFS.

The development of DLB_Bench gives us the opportunity to study and compare the performance of existing approaches in various configurations, which may be included in future publications.

Moreover, we plan to optimize our implementation to use shared-memory communication for intra-node communications and explore various algorithmic improvement. Some shared work-queue and associated work stealing protocol may provide lock-free algorithms as well as not require the victim to take action in between thefts. Small-world generation and victim selection strategies may take more finely data locality into account in the form of expected or average latency. Lock-free memory allocators can be found in the literature [16], which may be adapted to our results returning strategy, further alleviating contention at scale. Finally, in [4], Berenbrink *et al.* show that a number of theft policies are viable in the classical work-stealing scheme, which may be explored in this context.

Acknowledgment. This work has been funded by the European FP7 Exa2ct project, ATOS, and the ECR lab, a collaboration between CEA, UVSQ, and Intel. The authors thank the GASPI and GPI-2 development team for their very good support and advice. The authors also thank CRIANN, IT4I and BSC for the compute resources and assistance.

References

1. Acun, B., et al.: Parallel programming with migratable objects: Charm++ in practice. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 647–658. IEEE Press (2014)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
3. Bader, D.: Designing scalable synthetic compact applications for benchmarking high productivity computing systems. Cyberinfrastructure Technol. Watch. **2**, 1–10 (2006)
4. Berenbrink, P., Friedetzky, T., Goldberg, L.A.: The natural work-stealing algorithm is stable. SIAM J. Comput. **32**(5), 1260–1279 (2003)

5. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM (JACM) **46**(5), 720–748 (1999)
6. Broquedis, F., et al.: hwloc: a generic framework for managing hardware affinities in HPC applications. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 180–186. IEEE (2010)
7. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 53. ACM (2009)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: ACM Sigplan Notices, vol. 33, pp. 212–223. ACM (1998)
9. Gautier, T., Lima, J.V., Maillard, N., Raffin, B.: Xkaapi: a runtime system for data-flow task programming on heterogeneous architectures. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1299–1308. IEEE (2013)
10. Grünewald, D., Simmendinger, C.: The GASPI API specification and its implementation GPI 2.0. In: 7th International Conference on PGAS Programming Models, vol. 243 (2013)
11. Kleinberg, J.: The small-world phenomenon: an algorithmic perspective. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, pp. 163–170 (2000)
12. Kleinberg, J., Rubinfeld, R.: Short paths in expander graphs. In: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, pp. 86–95. IEEE (1996)
13. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in Intel Threading Building Blocks. Intel Technol. J. **11**(4), 309–322 (2007)
14. Lusk, E.L., Pieper, S.C., Butler, R.M., et al.: More scalability, less pain: a simple programming model and its implementation for extreme computing. SciDAC Rev. **17**(1), 30–37 (2010)
15. Machado, R., Lojewski, C., Abreu, S., Pfreundt, F.J.: Unbalanced tree search on a manycore system using the GPI programming model. Comput. Sci. Res. Dev. **26**, 229–236 (2011)
16. Michael, M.M.: Scalable lock-free dynamic memory allocation. ACM Sigplan Not. **39**(6), 35–46 (2004)
17. Min, S.J., Iancu, C., Yelick, K.: Hierarchical work stealing on manycore clusters. In: 5th Conference on Partitioned Global Address Space programming Models (2011)
18. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
19. Perarnau, S., Sato, M.: Victim selection and distributed work stealing performance: a case study. In: Parallel and Distributed Processing Symposium, vol. 28. IEEE (2014)
20. Quintin, J.-N., Wagner, F.: Hierarchical work-stealing. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6271, pp. 217–229. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15277-1_21
21. Tchiboukdjian, M., Gast, N., Trystram, D., Roch, J.L., Bernard, J.: A Tighter Analysis of Work Stealing. Angorithms and Computation, pp. 291–302 (2010)
22. Woodall, T.S., Shipman, G.M., Bosilca, G., Graham, R.L., Maccabe, A.B.: High performance RDMA protocols in HPC. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) EuroPVM/MPI 2006. LNCS, vol. 4192, pp. 76–85. Springer, Heidelberg (2006). https://doi.org/10.1007/11846802_18