Victim Selection and Distributed Work Stealing Performance: A Case Study

Swann Perarnau
RIKEN AICS
Kobe, Japan
perarnau@riken.jp

Mitsuhisa Sato
University of Tsukuba/RIKEN AICS
Tsukuba, Japan
msato@cs.tsukuba.ac.jp

***Abstract*—Work stealing is a popular solution to perform dynamic load balancing of irregular computations, both for shared memory and distributed memory systems. While shared memory performance of work stealing is well understood, distributing this algorithm to several thousands of nodes can introduce new performance issues. In particular, most studies of work stealing assume that all participating processes are equidistant from each other, in terms of communication latency.**

This paper presents a new performance evaluation of the popular UTS benchmark, in its work stealing implementation, on the scale of ten thousands of compute nodes. Taking advantage of the physical scale of the K Computer, we investigate in details the performance impact of communication latencies on work stealing. In particular, we introduce a new performance metric to assess the time needed by the work stealing scheduler to distribute work among all processes. Using this metric, we identify a previously overlooked issue: the victim selection function used by the work stealing application can severely impact its performance at large scale. To solve this issue, we introduce a new strategy taking into account the physical distance between nodes and achieve significant performance improvements.

***Keywords*-distributed load balancing; work stealing; latency**

I. Introduction

The current petascale supercomputers are in majority composed of multicore, shared memory nodes, linked together by a high speed network. As architectures evolves, both the number of cores inside a compute node and the number of these nodes in such systems grow. As an example, the K Computer, ranked third in the TOP500 of November 2012, is composed of more than 80,000 compute nodes, each with 8 cores. Distributing computations, in particular irregular ones, at such scale requires increasingly complex and dynamic load balancing systems.

Work stealing [1] is a provably efficient scheduling algorithm for such distributed, dynamic load balancing requirements. It is becoming increasingly popular, both for shared memory systems (intra-node load balancing) and in distributed settings (inter-node load balancing). The performance of work stealing in shared memory in particular is the subject of a large body of work, both theoretical and experimental [1]–[6].

Unfortunately, while the shared memory issues of work stealing like locality or lock contention are well understood, scaling such scheduling algorithm to large distributed systems introduces new ones. For example, standard message passing programming models might require that a worker stop advancing the computation to answer steal requests from others, thus slowing down the application. Distributed termination detection is another well studied issue. In this paper, we will focus on a previously overlooked issue in High Performance Computing settings: the impact of large scale latencies on load balancing performance.

Using the publicly available, MPI-based implementation of the Unbalanced Tree Search benchmark, we evaluate the performance of work stealing at the scale of several thousands compute nodes. Interestingly, at these scales, allocating several MPI processes by compute node results in a worse performance than using a single process per node for this implementation. Studying the problem further, this paper makes the following contributions.

First, we introduce a metric to study the time needed by the work stealing algorithm to distribute work among processes. Based on a lightweight trace of the scheduler, this *scheduling latency* helps us compare work distribution strategies. Second, we demonstrate that by changing the way a victim is chosen during work stealing, taking into account latencies between nodes, we can mitigate work distribution issues and improve significantly the performance of the benchmark. Finally, to the best of our knowledge, we provide the first performance comparison between different process allocations strategies at the scale of several thousands compute nodes for work stealing, and the first use of latency-conscious victim selection at this scale. We will also demonstrate that even a simple, portable implementation of work stealing can perform efficiently at this very large scale.

The next section presents the UTS benchmark we used in this paper, details its work stealing implementation and motivates further our study. Section III introduces our new measure of work stealing performance and its use on the reference version of UTS. In Section IV, we study two alternative victim selection schemes for this benchmark and demonstrate significant performance improvements. These improvements are discussed further in Section V, using additional traces. We also briefly discuss in this section the impact of work granularity on these improvements. We conclude this paper by a review of related works in Section VI and suggesting future experiments in Section VII.

II. The UTS Benchmark

For our study, we chose to analyze the UTS (Unbalanced Tree Search) Benchmark. This benchmark became popular in recent years to evaluate dynamic load balancing implementations, both in shared memory and distributed memory frameworks. First introduced by Prins *et al.* [7] and then formalized by Olivier *et al.* [8] UTS is, as its name suggests, an irregular application. The benchmark is designed to perform the parallel traversal of a randomly generated unbalanced tree, counting the total number of nodes.

The input tree is implicit: each node in the tree contains all the information required to generate its children. Thus, each process involved in the computation is responsible for the generation of a part of the tree, managing nodes in a local stack. The shape, size and depth of the tree are controlled by a set of input parameters to the benchmark, and for a set of parameters, the same tree will always be generated no matter the underlying hardware or language. This is achieved by using SHA hashes for random child generation. Moreover, dynamic load balancing is required during execution, as only one process start with the root of the tree, and the tree is heavily unbalanced. This imbalance is created by the relative short depth of generated trees compared to their size (billions of nodes with a depth in the order of ten for example) and the exact child generation process. For example, UTS is able to generate binomial trees. In such trees, with parameters m and q, a node generate m children with probability q and 0 with probability $1 - q$. Thus, while all nodes have the same expected number of children, subtrees will vary greatly in size, requiring frequent load balancing between processes. This benchmark was used in several studies to evaluate load balancing frameworks, and work stealing in particular [7]–[10].

A. Work Stealing Implementation

The portable nature of the tree generation algorithm of UTS allowed several implementations, in different languages to exist. In the remainder of this paper, we will use the public implementation [11] using MPI with work stealing[1]. While other studies demonstrated good performance and scalability of UPC and X10 implementations of this benchmark, this MPI version is the most portable one. Indeed, it only requires version 2 of the MPI standard. It is also a very simple implementation, making the modifications appearing latter in this paper easier.

This implementation of UTS follows the general structure of any work stealing application. When work is available, a process retrieve a node from its stack. This node's data is then used to compute its children, that are pushed into the stack. If no work is available, a victim process is selected, and work is fetched from its stack. This process continues until all work is exhausted. Such condition is detected by a tokenring distributed termination algorithm. Algorithm 1 resumes this behavior. Like most of the UTS implementations, our reference program also exhibits several distinguishing features. First, this work stealing implementation does not use Cilk-like [4] continuations. Instead of managing *tasks* (function and data), the process only manipulate work items (tree nodes). This reduces the overhead associated with task creation and scheduling.

[1]mpi_workstealing.c in the source distribution.

Second, work items are managed in chunks. In this application, all work items are of the same type (node in the tree) and the same size. Thus, the underlying implementation allocates memory for groups of nodes instead of per node to limit memory management overhead. These chunks also represent the steal granularity: a thief will steal a single chunk of nodes instead of a single node. Chunks also behave like a private work queue: if there is only one incomplete chunk in the stack of a process, no work can be stolen, as the first chunk is always considered private. While a large chunk size might thus limit work availability, previous studies [7], [8] have demonstrated significant performance improvements due to these chunks in UTS. While the size of a chunk can be configured, we will use the default one of 20 nodes per chunk for the remainder of this paper. The authors of UTS have previously stated that this size provides good performance on a wide range of systems.

Third, this implementation does not respect the *work-first principle*. Indeed, a process stealing work will in fact post a request to its victim by a message, and the victim will stop working on its queue to package work and send it to the stealer. The performance overhead of this implementation choice is mitigated by the use of asynchronous communication primitives. A multithreaded solution, using one thread for communication and one for work, with lockless task management could furthermore alleviate this issue, but this implementation avoids these optimizations in favor of simplicity.

```
if $rank = 0$ then
    PUSH(stack,root)
end if
while not finished do
    while $node \leftarrow$ GET($stack$) do
        while $child \leftarrow$ NEXTCHILD($node$) do
            PUSH(stack,child)
        end while
    end while
    while stack is empty do
        $v \leftarrow$ SELECTVICTIM
        STEAL($victim$)
    end while
end while
```

Fig. 1. Schematic view of the UTS work stealing implementation.

Finally, victim selection is deterministic. This last point is perhaps the most surprising one. Indeed, traditional work stealing assumes that victims are selected following an uniform random distribution. Most theoretical proofs of the efficiency of work stealing also rely on this assumption [1], [3]. However, all the implementations provided in the public source code release of UTS use a deterministic scheme. A process with rank i will choose as its first victim its neighbor (rank $i+1 \bmod number\_of\_processes$). Subsequent steals will choose the next neighbor in a round-robin fashion. Notice that a successful steal does not impact this choice: the next search for work will start at the neighbor of the last victim. Still, we will see in the next subsection that this deterministic choice is reasonable at small scale. We should also mention that this observation motivated in part the experiments of this paper.

B. Reference Performance Evaluation

We evaluate in this subsection the performance of this reference implementation, both at small and large scales.

All experiments in the remainder of this paper were performed on the K Computer [12]. It contains over 80 000 compute nodes, each composed of a single SPARC processor chip and 16 GiB of memory. The processor, a SPARC64 VIIIfx, was specifically designed for this system. This chip is produced by Fujitsu using a 45-nm process and is composed of 8 cores operating at 2 GHz for a peak performance of 128 GFLOPS [13]. It is an extended version of the SPARC-V9 architecture targeted at high performance computing. Nodes are connected by a 6 dimensional mesh torus called Tofu [14]. The compute nodes run a custom version of Linux. Both the compiler and the MPI communication library are provided by Fujitsu and contain platform-specific optimizations. Unless specified otherwise, we used the timing measurements (and additional statistics) already reported by the UTS benchmark. As the K Computer contains an impressive amount of compute nodes, we can evaluate UTS both when using all cores of a compute node and when using only one MPI process per node. The job scheduler on the K Computer was responsible for physical node allocation, and tends to distribute nodes in a 3D rectangle minimizing the average number of hops between processes.

Figure 2 presents the efficiency of our reference benchmark between 8 and 128 MPI processes. The figure lists this performance for 3 distinct process allocations. The setup named *1/N* allocates 1 MPI process per compute node. The setup named *8RR* allocates 8 MPI processes per node, using round robin numbering (processes $i, i+8, i+16, \ldots$ are on the same node). The last allocation, *8G*, groups the first 8 processes on the first compute node, the next 8 on the next node and so on. For this experiment, a binomial tree named T3XXL was used as input. This tree is recommended by the UTS creators for this range of MPI processes and was used in other studies [8]. The exact parameters of this tree are displayed in Table I.

Figure 3 presents the speedup of this same benchmark between 1024 and 8192 MPI Processes. To ensure enough work was available, we choose another binomial tree (T3WL). Its characteristics are also reported in Table I. Unfortunately, due to runtime limits of the job scheduler, the runtime of UTS with this tree and a single MPI process could not be measured directly (it exceeds a day). Thus it was extrapolated from the speed, in node searched per second, of the previous input tree search. We rely on the assumption that all single MPI process executions, for the same type of generated trees,

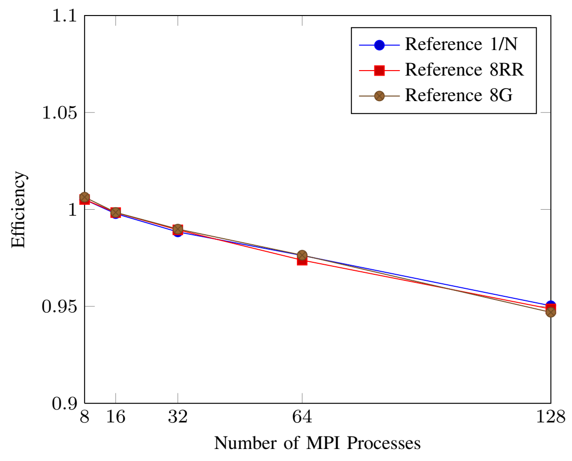

Fig. 2. Efficiency of the reference MPI Workstealing implementation between 8 and 128 MPI processes with various process allocations.

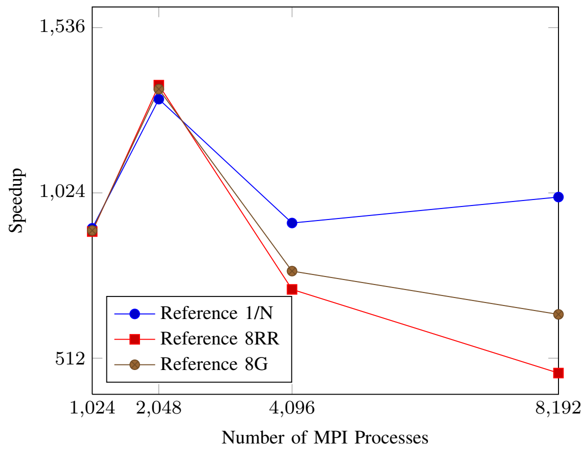

Fig. 3. Speedup of the reference MPI workstealing implementation between 1024 and 8192 MPI processes.

should have the same speed. Together, these two figures demonstrate that this UTS implementation performs very well for small numbers of MPI processes, but does not scale past 2048 nodes. Additionally, it appears that the benchmark performance is severely impacted by the way processes are distributed among compute nodes. In particular, allocating successive ranks to different compute nodes results in the worse performance observed. Indeed, the deterministic round robin victim selection is in direct conflict with the MPI process allocation in this case. These results are in line with previous performance studies of UTS, that demonstrated good scalability up to 1024 compute nodes and limited performance for more processes than that [10].

III. Measuring Load Balancing Efficiency

To understand the previous performances, we detail in this section a metric to measure load balancing efficiency.

The role of a dynamic load balancing scheme is to maximize the amount of processes having work at any given time. Thus, if a problem is comprised of enough work items, the state of

Name	Tree Type	t	r	b	m	q	Tree Size
T3XXL	Binomial	0	316	2000	2	0.499995	2793220501
T3WL	Binomial	0	559	2000	2	0.4999995	157063495159

TABLE I
UTS INPUT TREES PARAMETERS

an application should roughly be separated in three phases: the starting phase where work is distributed to all processes, the finishing phase during which work becomes scarce and the number of active processes decreases and the middle phase for which most processes are processing work. Of course, if work generation is irregular, as with UTS, balancing is also needed during the middle phase, but an efficient framework should be able to maintain a reasonable amount of processes busy. This intuition drives the definition of our performance metric. If one was to trace the active and idle phases of each process participating in the computation, it should be possible post-mortem to determine the number of active processes at any time during execution of the application.

We define here **active** phases as periods of time during which a process's stack contains work. Thus, in the chosen UTS implementation, all the time where a process is generating new nodes or handling MPI operations in between, (responding to steal requests for example) count as active. Similarly, a process is **inactive** if it does not have work locally. It should be noted that with such definition, most types of load balancing operations can be counted in either type of phase. Assuming there exists a trace of all processes indicating the time of each transition from one type of phase to the other, with the starting time of the application as $t = 0$, we now define the following metrics.

First, let $workers(t)$ be the number of processes in an active phase at time t. From this number we derive the maximum number of workers at any given time during execution W_{max} and an occupancy ratio $O(t) = \frac{workers(t)}{N}$, N being the number of processes executing the application. Second, let the *starting latency* be $SL(x) = \frac{min(t,O(t)=x)}{T}$, with T the total execution time of the application. This measure computes, for a given occupancy ratio, the first time it was exceeded during execution and represent it as a ratio of the total execution. Similarly, we define the *ending latency* as $EL(x) = \frac{T-max(t,O(t)=x)}{T}$. Intuitively, the starting latency gives us the speed, relative to the total execution time of the application, at which a number of processes becomes active. The ending latency reflects a similar idea: how far away in the execution the framework is able to maintain a number of processes active. As an example, an execution where the first time 10% of the processes have work happens 5% of the execution time after beginning has $SL(10\%) = 5\%$.

We modified the reference implementation to trace the necessary information and computed these latencies for the two 1/N executions in the previous section. As the trace only contains a time and the new state at each phase transition, it is lightweight. No significant change in execution time was detected. Starting times for each processes were recorded and

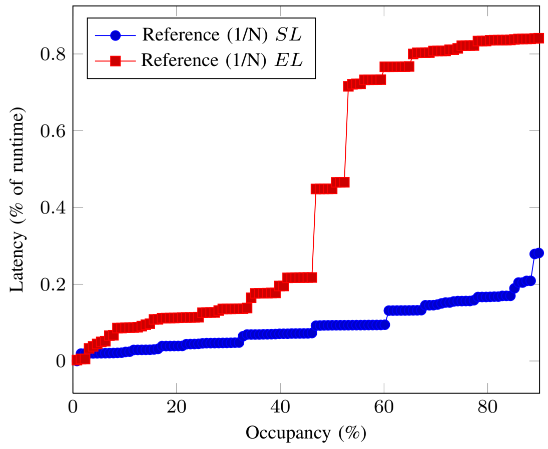

Fig. 4. Starting and Ending latencies for an execution of the reference implementation with 128 MPI processes, 1 processes per node.

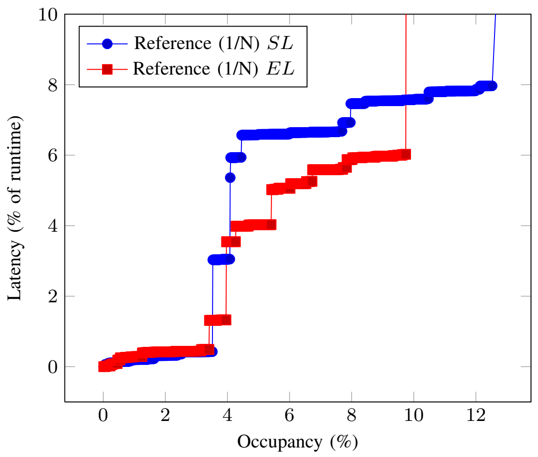

Fig. 5. Starting and Ending latencies for an execution of the reference implementation with 8192 MPI processes, 1 process per node. Data is limited to latencies lower than 10%.

the trace modified to account for clock skew. The figures 4 and 5 present these latencies. The former presents starting and ending latencies up to an occupancy of 90% for the execution of UTS over 128 MPI processes. The latter is a partial rendering of these latencies for an execution using 8192 MPI processes: we limit the graphic to latencies under 10% of the execution time. Note that based on our analysis, this particular execution never exceeded 3538 processes (43% of occupancy) with a starting latency of 52.5% and an ending one of 47.5%. As we can see, for the execution over 128 processes, the work stealing process is able to provide most workers with nodes shortly after the start of the execution, and almost to the end of it: both latencies for an occupancy of 90% are under 1% of the execution time. On the contrary, the large execution struggle to provide work to most workers: only 12.5% of the processes are active after 10% of the execution.

IV. Alternative Victim Selections

A possible explanation for this poor performance of the reference UTS implementation at large scale is the deterministic victim selection. As only process 0 has work when the application starts, looping over the ranks in round robin in search for work appears to be highly inefficient. Moreover, most proofs of work stealing demonstrated its optimality with a random victim selection process. In this section, we investigate the performance of this random victim selection strategy.

A. Traditional Random Selection

We define the random victim selection as the process of choosing with a uniform random distribution over the ranks of all other MPI processes one victim to steal. The process is repeated as long as needed, without modification, until work is found.

We modified the reference implementation to use this victim selection process. The modification is simple, only 5 lines of code were changed. This version of the benchmark is named Rand in the remainder of this paper. Figure 6 presents performance results for this implementation, with the same experimental setup than previously.

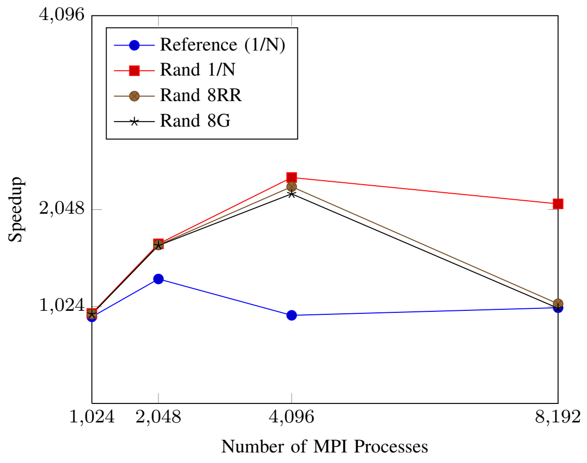

Fig. 6. Speedup of the MPI workstealing implementation with random selection, between 1024 and 8192 MPI processes. The reference performance (in 1/N) is also reported.

As we can see, using random selection results in better performance when allocating only one process per node, but does not improve the performance of executions using all the cores for 8192 MPI processes. Figure 7 traces the number of failed steals for these runs. This figure confirms that the number of failed steals decreases significantly by using a random victim selection strategy, resulting in better performance overall.

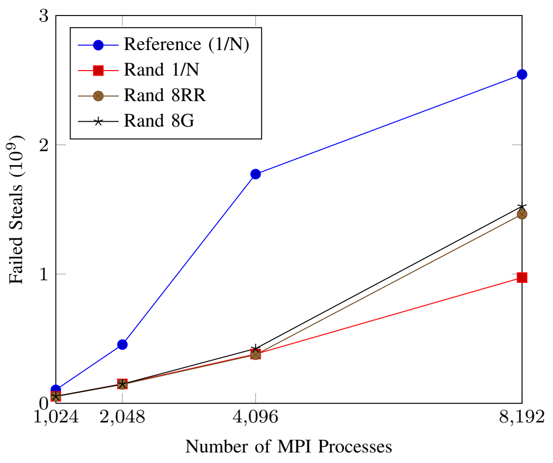

Fig. 7. Number of failed steals for the MPI workstealing implementation with random selection, between 1024 and 8192 MPI processes.

B. Skewing the Distribution

While we just showed that traditional random selection strategy performs better than the deterministic one, it might still be possible to improve this performance. Indeed, the major advantage of a random selection is to, on average, lower the number of steals required to find work. However, in modern supercomputers, not every steal attempt takes the same time. For example, communication between two MPI processes on the same CPU, or on the same *blade* will potentially be faster than across racks (more network hops are necessary). Variations in response time for a steal request could explain why using all the cores of a node decreases the performance of the application.

Our experimental platform is the K Computer. The physical organization of the computer is as follows. First, compute nodes are in groups of four on a blade. Communication across the blade use a dedicated transport. Then, 3 blades are joined together, forming a 2x3x2 cube. This cube represent 3 of the 6 dimensions of the Tofu network. Finally, these cube are joined in a 3D mesh torus, with one dimension for the rack (8 cubes are in the same rack), and two across racks. As each of these levels use different network links, latencies between nodes in the same blade are lower than inside the cube or across racks. Furthermore, the number of compute nodes inside a rack (only 96) means that an allocation of 8192 nodes can easily span across more than 80 racks, and in practice we observed that a communication between two processes can go through more than 10 hops.

To reflect these variations in response times for a steal, we designed a random selection strategy using a biased distribution. The idea is the following: while preserving the ability to steal any process, weight the probability of one process stealing another by the distance between those two. The farther a process is, the lower the probability of being chosen. As the Fujitsu MPI implementation provide extensions to query the 6D coordinates in the Tofu network of any MPI rank, we used the Euclidean distance between nodes to weight the probability. Thus, let $p(i,j)$ the probability of rank i stealing rank j ; $x_i, y_i, z_i, a_i, b_i, c_i$ be the coordinates in the Tofu network of rank i and $e(i,j)$ the euclidean distance between ranks i and j:

$$w(i,j) = \begin{cases} \frac{1}{e(i,j)} & \text{if } e(i,j) \neq 0 \\ 1 & \text{if } e(i,j) = 0 \end{cases}$$

$$p(i,j) = \frac{w(i,j)}{\sum_{j}^{j \neq i} w(i,j)}$$

Figure 8 gives the probability distribution function for an actual deployment of a 1024 nodes job (one process per node) on the K Computer. We modified the reference implementation to use these probabilities for victim selection. The modification uses the GNU Scientific Library [15] for random number generation and general discrete distribution sampling. This version of the benchmark is named Tofu for the remainder of this paper. We report the resulting performance in figure 9. As this figure shows, the performance of our benchmark is improved by this new victim selection strategy. Unfortunately, while all allocations strategies perform better than with the classical random selection for the same allocation, only the allocation of 1 process per node performs better than the previous best.

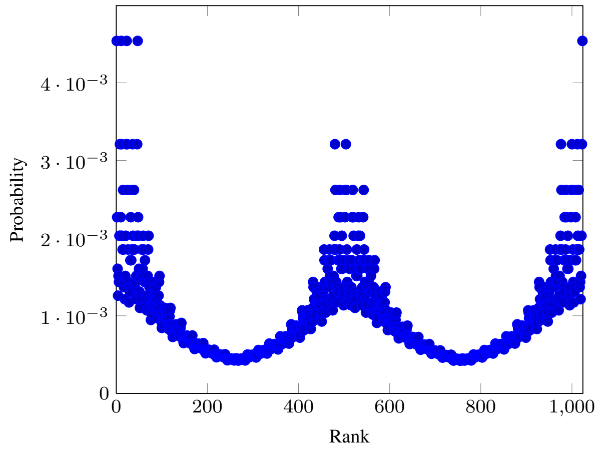

Fig. 8. Probability distribution function of p(0,x) for a example deployment on the K Computer over 1024 MPI processes, 1 per node.

To demonstrate further the efficiency of this strategy, figure 10 reports the average duration of a work discovery session on the same configuration. A work discovery session starts when a process exhaust its work and ends with either work in the queue or application termination. As we can see, the topology-specific victim selection strategy results in much faster work discovery.

C. Stealing Half the Work

We would like to point out again that the reference implementation only steals one chunk at a time. However, several studies have demonstrated that stealing half the work of the

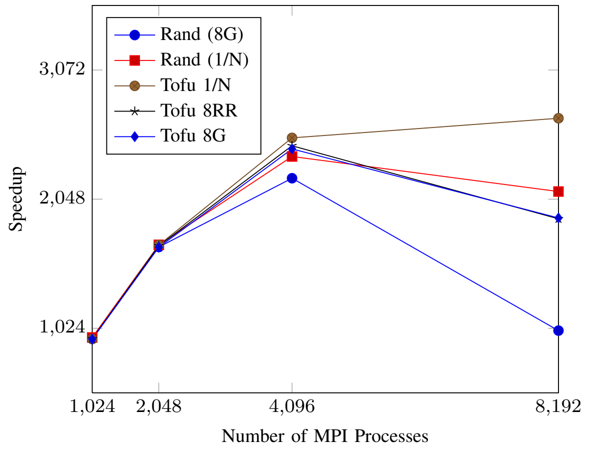

Fig. 9. Speedup of the MPI workstealing implementation with skewed distribution, between 1024 and 8192 MPI processes. The performance of random selection (in 1/N and 8G) is also reported for reference.

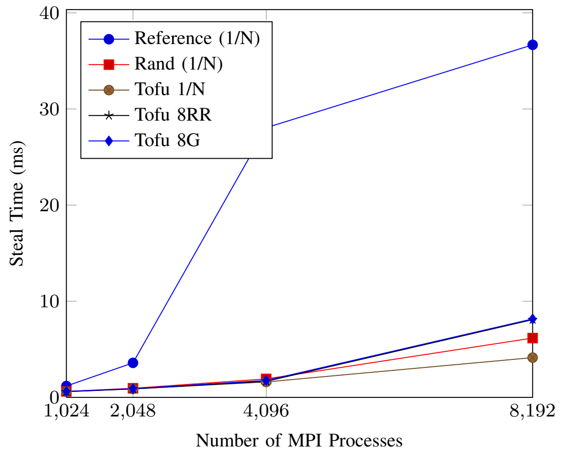

Fig. 10. Average duration a of work discovery session of the MPI workstealing implementation with skewed distribution, between 1024 and 8192 MPI processes. Statistics for random selection and the reference implementation is also reported for reference.

victim is an optimal strategy [2], or at least provide good performance. The reason is very intuitive: stealing half the work make it possible for a thief to be stolen himself as soon as it retrieves work, making the overall work availability better. To reflect this, we modified further our various implementations so that steals transfer half the chunks instead of just one. Figure 11 presents the performance of these versions on 8192 nodes, allocating one process per node.

This figure confirms our assumptions. While none of the previous modifications impact meaningfully the performance of UTS at small scale, the combined use of our skewed victim selection and half-stealing performs 3 times better than the original. More importantly, this last version is able to speedup up to 8192 MPI processes, with all process allocation strategies, while it was not the case for the original version after 2048 processes. To further understand the performance differences between those two versions we compare in Figure 12 their starting latencies.

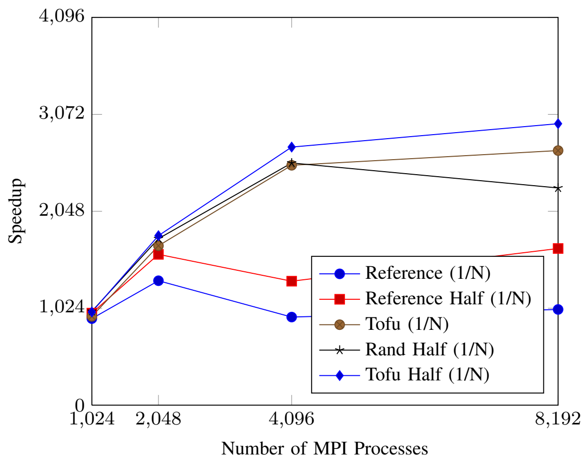

Fig. 11. Speedup of the MPI workstealing implementations stealing half the work, between 1024 and 8192 MPI processes.

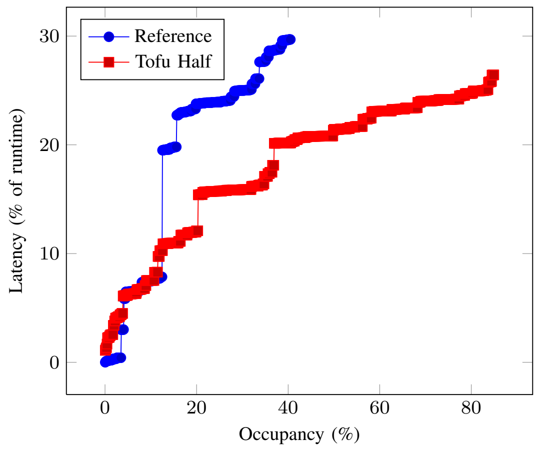

Fig. 12. Comparison of starting latencies between the reference implementation and the optimal one, with 8192 MPI processes, 1 process per node. Tracing data was limited to the first 5000 phases of each process.

As we expected, while the reference implementation is struggling to provide work to most processes during the whole execution, the optimized version achieves a higher occupancy significantly faster. Figure 13 also confirms that the optimized version maintains a high occupancy until late in the execution.

V. Further Discussion

In this section, we first detail further how the final version of the benchmark (skewed victim selection and half stealing) performs better. In particular, we look at search time and failed steal information. Secondly, we discuss the possibility that these performances are impacted by work granularity: how much compute time each node of the tree (work item) takes to process.

A. Search Time and Failed Steal

We now look closer into the performance of this skewed victim selection. Two metrics reported by UTS are of interest: the average search time and the number of failed steals. The search time is defined by UTS as the portion of the execution

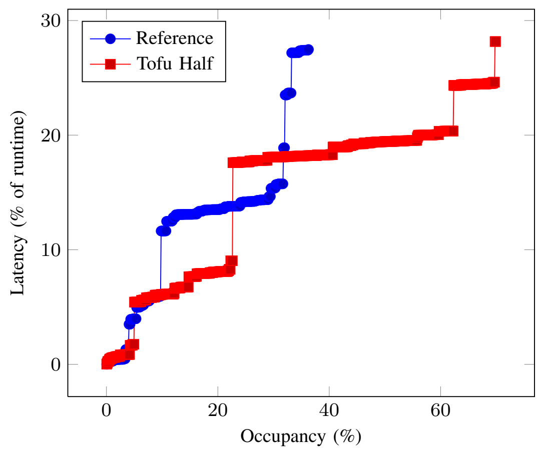

Fig. 13. Comparison of ending latencies between the reference implementation and the optimal one, with 8192 MPI processes, 1 process per node. Tracing data was limited to the last 5000 phases of each process.

time a process was waiting for a steal answer (work or no work). The number of failed steals report the number of steal requests that were answered negatively.

We report in Figure 14 the average search time of our optimized benchmarks (skewed victim selection and steal half) and the original values, while Figure 15 displays the accumulated number of failed steals. As expected, taking into account network latencies and and stealing half the chunks of the victim greatly diminishes the time spent searching for work. The number of steals failing also decreases, as a result of better work distribution.

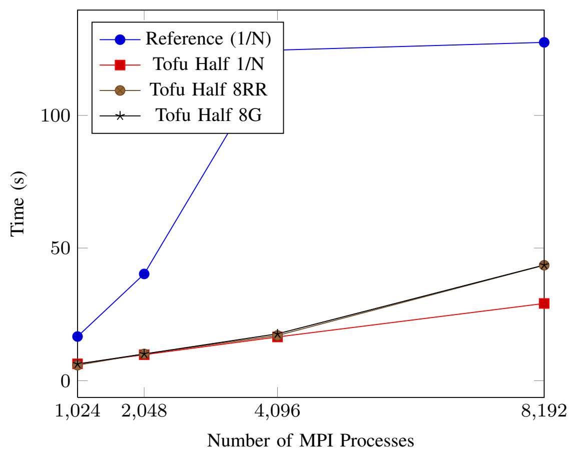

Fig. 14. Average search time of a process. Comparison between original version (1/N allocation) and skewed victim selection with half stealing for a number of processes from 1024 to 8192.

B. Impact of Work Granularity

All the experiments presented before are configured with the same work granularity: when a node is created by UTS, a single round of SHA is computed for its id. This same id is latter used to compute its number of children. On the K Computer, UTS is able to process an average of 970000 nodes

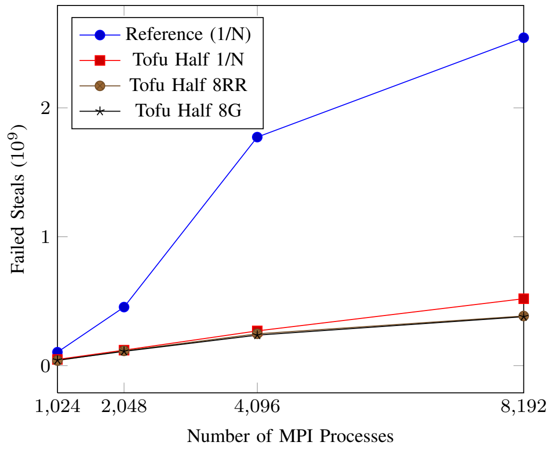

Fig. 15. Number of failed steals. Comparison between original version (1/N allocation) and skewed victim selection with half stealing for a number of processes from 1024 to 8192.

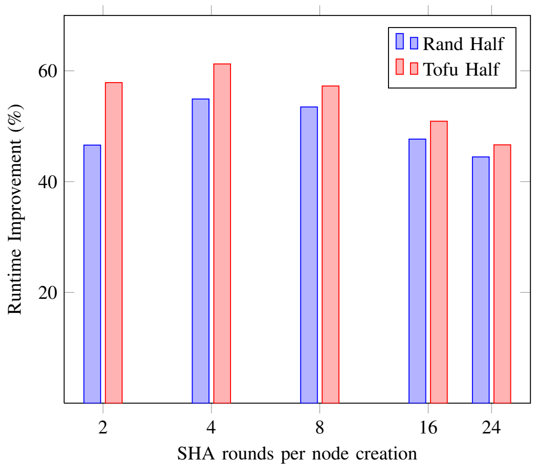

Fig. 16. Comparison between runtime improvements of topology-aware and random victim selection over "Reference Half", for varying work granularities. 1/N allocation over 8192 nodes.

per second. Thus, a single node provide little compute time, relative to the average time it takes the application to steal work. As a reference, an execution of the "Tofu Half 1/N" version on 8192 nodes averages, on each MPI rank, 6800 work discovery sessions, with 4 ms per session in the span of a 75 seconds execution time. If we assume that all active sessions are of the same length, it amounts to only 11 ms between two inactive ones: a steal only provides enough work for 3 times the time it takes to make it.

To study this question further, we compare in figure 16 the performance of "Rand Half 1/N" and "Tofu Half 1/N" with different work granularity. We used the UTS parameter dictating the number of SHA rounds to execute when creating a node. The figure reports the improvement those two strategies provide over the "Reference Half (1/N)" one. As we can see, as granularity increases, the difference in improvement between the two random strategies diminishes. Indeed, as each steal provides more work (in compute time) to the thief, the impact of varying latencies between steal requests on work balance is lowered.

These last observations confirms that *how* victims are selected in a work stealing algorithm can be an important optimization point, and that the relation between latency and work granularity should be studied in more details in the future, in particular in contexts with large number of compute nodes and their distance to each other becoming heterogeneous.

VI. Related Works

Efficient load balancing of parallel applications is a vast and widely studied topic. In particular, irregular applications are challenging problems, as their work is distributed unevenly among processes. Such applications thus require dynamic load balancing during their execution. Whether these applications run in shared memory systems and in distributed memory ones, their load balancing is still nowadays the topic of intensive research.

Work stealing, popularized by Cilk [4] is a scheduling solution to such issues. In Cilk, an extension of C (and later C++), work is divided in tasks (continuations), and each worker manages a stack of local tasks. Workers execute this stack in FIFO order, and when a worker stack becomes empty, it *steals* the bottom task of another worker. The work on Cilk also introduced the *work-first principle*: when given a choice implementing work stealing, always favor executing work over performing scheduling operations. In particular, most of the overhead of the steal operation is assumed by the stealer. Since then, numerous frameworks have used work stealing to perform load balancing in shared memory. It is the case for example of some OpenMP [16] implementations, Intel TBB [6] and XKaapi [5] for example. An extensive body of work also focus on performance issues for work stealing implementation, including cache thrashing [17]–[19], lock contention from multiple steals to the same worker [20] or the use of GPU architectures [21].

In distributed memory settings, a number of recent parallel languages for HPC have studied work stealing mechanisms, UPC [18] and X10 [9] for example. As these languages try to address both intra-node performance and inter-node performance, studies focus on both shared memory issues of work stealing and distributed ones. Distributed work stealing indeed suffers from additional performance issues. The unavailability, until recently, of one-sided communication primitives for example impeded the application of the work-first principle: to steal a worker, it was necessary to send him a message and wait for its response. Distributed termination detection is another widely studied issue [9], [10].

UTS was designed from the start as a benchmark representing irregular applications. Several works investigated the use of work stealing in UTS. Oliver *et al.* [8], in the paper formalizing UTS as a benchmark, focus on shared memory performance, implementing it with both UPC and OpenMP. The paper also contains an experiment in distributed memory using UPC, but the number of processes is limited to 24 at most. One particular work stealing parameter is studied: chunk size. An MPI implementation of UTS is also studied in [22]. The authors studied both work sharing and work stealing load balancing, focusing their study on message polling and distributed termination detection. The study was limited to small executions, up to 64 processes. More recent papers using UTS include Dinan *et al.* [10]. By building a work stealing framework on top of ARMCI [23], the authors were able to use one-sided operations to implement efficient task queues, both inside a compute node and between nodes. They also implemented an *aborting steal* mechanism allowing a steal to fail fast when no work is available. While the authors report scaling UTS to 8192 processes as well, their experiment are limited to 2048 compute nodes inside a single cluster. Our work studies the scalability of UTS at a larger level, and without using one-sided communications or shared memory optimizations that can hide heterogeneous latencies between ranks. Saraswat *et al.* [9] is another recent performance study of UTS. The authors, understanding that fast distribution of work is critical to performance, suggest a distribution scheme based on *lifelines*. These lifelines act as work distribution links that worker favor over random work stealing. After the number of steal attempts exceeds a threshold, idle worker wait for their lifelines to provide work, thus limiting the lock and network contention in the system. This paper also includes *lifestories*, a graphic representation of each process activity during an execution. These lifestories are very similar to the trace we suggest to compute starting and ending latencies, but the authors limits their usefulness to activity graphs, whereas we uses them to measure the speed of work distribution in the system. Furthermore, this paper limit experimentation to a thousand of processes. To the best of our knowledge, our work is the first to consider victim selection as an optimization point for UTS. Finally, several papers demonstrated the usefulness of stealing half the work in UTS [10], [24].

Less connected to our work, several papers have discussed *hierarchical* work stealing [18], [25]. More precisely, they suggest building a hierarchical organisation of the processes depending on the memory hierarchy of a single node or on the differences between latencies inside a cluster and across two clusters. These schemes then use fixed policies, one per level to balance the work and not necessarily based on randomness, whereas we directly use the distance between nodes as a weight for random selection. Furthermore, these studies were very limited, containing two-levels hierarchies at most and a scaling study for a small number of processes. Our skewed victim selection is not limited to any hierarchical description of the system, even if it requires the coordinate of each process inside the network. It could be applied to other node organizations than our 6 dimensional network.

VII. Conclusion

In this paper, we discussed the impact of victim selection on the performance of a work stealing application. As a case study, we chose to analyze the performance of UTS, a popular benchmark representing irregular applications needing dynamic load balancing during execution. Scaling this benchmark to 8192 compute nodes of a recent supercomputer, we identify a set of bottlenecks in the public MPI implementation of the benchmark. In particular we identify that a deterministic victim selection strategy and single-chunk steals are responsible for the poor scalability of this implementation. While these features are of little impact at the small scale UTS was most commonly studied before, they forbid this implementation to scale past 2048 nodes.

After these bottlenecks were identified, we modified the benchmark to use a skewed random distribution for victim selection. By weighting the probability of a node being stolen by its distance to the thief in the network, we more than double the performance of our reference application and allow it to scale up to 8192 processes, on the same number of nodes. Moreover, we introduce a metric to provide insight into the speed at which a load balancer can distribute work among processes. Based on an activity trace of all participating processes, this metric measure the amount of execution time needed for a minimum amount of processes to become active. Similarly, we use this information to assess the time separating a given level of activity from the end of the execution. Using various work granularities, we also identified that these latencies issues mostly impact the performance of work stealing when the amount of compute time stolen at once is small.

From these experiments we argue that, as supercomputers continue to grow in their number of computing nodes, latencies and victim selection strategies should be taken into account to achieve efficient distributed work stealing at large scales.

Studying the scalability of UTS past tens of thousands of processes is a natural extension of this study. Preliminary experiments on the K Computer indicate that additional optimizations seen in other studies like one-sided communications might be required to achieve a decent efficiency in any reference implementation to compare our work with. In the context of PGAS languages, we are also interested in implementing work stealing with dependencies. While UTS provide an excellent benchmark for pure-computation irregular applications, each work item in this benchmark is independent from the others, limiting the amount of data that need to be transfered during a steal. In the case of data dependencies, stealing a task can trigger massive communications and thus is more sensible to bandwidth inside a network. Studying the impact of the network on such problems might require new benchmarks, possibly using directed acyclic graphs generation [26] instead of random trees. Studying alternative victim selection strategies to take into account bandwidth between nodes is also one of the topics we are interested in.

Finally, a more immediate extension of our work is its application to existing, more efficient, implementations of UTS. While porting the necessary underlying communications library might be an issue, such a study would provide insight into the capability of a victim selection strategy to impact already tuned work stealing frameworks.

Acknowledgments

Part of the results were obtained by using the K computer at the RIKEN Advanced Institute for Computational Science. This work was supported by the JSPS Grant-in-Aid for JSPS Fellows Number 24.02711.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems*, 2001.

[2] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, 1999.

[3] M. Tchiboukdjian, N. Gast, D. Trystram, J.-L. Roch, and J. Bernard, "A tighter analysis of work stealing," in *Algorithms and Computation*, 2010.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM Sigplan Notices*, 1998.

[5] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *In Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[6] J. Reinders, *Intel threading building blocks*. O'Reilly, 2007.

[7] J. Prins, J. Huan, W. Pugh, C. Tseng, and P. Sadayappan, "Upc implementation of an unbalanced tree search benchmark," *Univ. North Carolina at Chapel Hill, Tech. Rep. TR03-034*, 2003.

[8] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "UTS: an unbalanced tree search benchmark," in *Proceedings of the 19th international conference on Languages and compilers for parallel computing*.

[9] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 2011.

[10] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[11] P. Sadayappan, J. Dinan, G. Sabin, C.-W. Tseng, J. Prins, and S. Olivier, "UTS official repository," http://sourceforge.net/p/uts-benchmark/wiki/Home/.

[12] H. Miyazaki, Y. Kusano, N. Shinjou, F. Shoji, M. Yokokawa, and T. Watanabe, "Overview of the K Computer system," *Fujitsu Scientific and Technical Journal*, vol. 48, 2012.

[13] T. M. et al., "SPARC64 VIIIfx: A new-generation octocore processor for petascale computing," in *IEEE Micro*, vol. 30, 2010.

[14] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6d mesh/torus interconnect for exascale computers," *Computer*, 2009.

[15] B. Gough, *GNU Scientific Library Reference Manual (3rd Ed.)*. Network Theory Ltd., 2009.

[16] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science Engineering, IEEE*, 1998.

[17] M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin, "A work stealing scheduler for parallel loops on shared cache multicores," in *Proceedings of the 4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)*, 2010.

[18] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.

[19] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory of Computing Systems*, 2002.

[20] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, 2005.

[21] T. Gautier, J. V. Ferreira Lima, N. Maillard, and B. Raffin, "Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures," in *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, 2013.

[22] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[23] J. Nieplocha and B. Carpenter, "Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Parallel and Distributed Processing*, 1999.

[24] Adnan and M. Sato, "Dynamic multiple work stealing strategy for flexible load balancing," *IEICE transactions on information and systems*, 2012.

[25] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Euro-Par 2010-Parallel Processing*, 2010.

[26] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, 2010.