A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters

Finnerty, Patrick

Kamada, Tomio

Ohta, Chikara

(Citation)
Concurrency and Computation: Practice and Experience,34(2):e6224

(Issue Date)
2021-02-18

(Resource Type)
journal article

(Version)
Accepted Manuscript

(URL)
https://hdl.handle.net/20.500.14094/90009368

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE

A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters

Patrick Finnerty[1] | Tomio Kamada[1] | Chikara Ohta[2]

[1]Graduate School of System Informatics, Kobe University, Hyogo, Japan
[2]Graduate School of Science, Technology and Innovation, Kobe University, Hyogo, Japan

Correspondence
Patrick Finnerty, Email:
finnerty.patrick@fine.cs.kobe-u.ac.jp

Present Address
657-0013
CS29, Graduate School of System Informatics
Rokkodai-cho 1-1,
Kobe-shi Hyogo-ken Japan

Abstract

Global Load Balancer libraries should be easy to use and allow users to easily obtain good performance for their applications on a variety of distributed systems. In this paper, we introduce a new tuning mechanism to our Java implementation of the Lifeline-based Global Load Balancer which automatically adjusts the task granularity to reach good performance based on some selected runtime metrics. We evaluate our system against four backtrack-search problems on both a many-core supercomputer environment and on a Beowulf server, achieving ideal performance with our tuning mechanism on the supercomputer. We also identify the limits of our mechanism in handling situations with reduced imbalance.

KEYWORDS:
Java, Distributed Computation, Load Balance, Cluster of SMP

1 | INTRODUCTION

Distributed computation involves splitting the overall computation in several fragments and assigning these fragments to computing resources for them to be processed in parallel. Achieving good performance is a challenge, especially with unpredictable or irregular computation. Without intervention during the computation, some compute nodes may finish their fragments earlier than others and remain idle while busy nodes are still processing their larger fragments. This problem is known as load imbalance. A kind of problem likely to suffer from this issue is backtrack-search algorithms. They usually involve a depth-first traversal of an exploration tree whose branches can be explored in parallel. However, branches are unlikely to bear similar-size sub-trees, resulting in load imbalance.

To address this issue, dynamic load balancers relocate fragments of the computation from busy nodes to idle nodes during the computation, trying to keep as many compute nodes busy for as long as possible. With the now widespread use of multi- and even many-core computers, load balancers are expected to balance the load both between and within compute nodes. Implementing such load balancing schemes is difficult and requires careful synchronization between compute nodes. Programmers therefore rely on existing libraries or frameworks to load-balance their computation. We believe they should be easy to use for programmers and not require any complicated adjustments/configurations.

Over-decomposition with profile-based approaches[1] can be successful in adjusting parameters of a load balancer in iterative applications. However, we cannot apply these techniques to our backtrack search applications as the computation needs only be performed once. We focus on the multithreaded lifeline load balancer first created in X10[2]. This scheme provides simple abstractions to programmers. However, there remain some settings that can greatly influence performance. In particular, the task granularity, which determines how often load balance operations are performed, can either produce excessive overhead if set too low or starvation if set too high. We introduce a novel tuning mechanism to this library which dynamically adjusts this parameter based on some selected runtime metrics. Users of the library no longer need to go through a tedious trial and error process to determine a good value for this setting as the tuner will automatically adjust it to a suitable level, achieving reasonable performance.

This is an extended version of our previous publication in which we demonstrated the capabilities of our first tuning mechanism design on the Oakforest-PACS supercomputer[3]. In this article, we improve on our first design by using a new metric, making it more robust against variations in problem implementation. We also expand our evaluation to a Beowulf server. The main contributions of this paper are:

- The porting of the X10 multithreaded lifeline-based global load balancer to Java
- The integration of a tuning mechanism to the load balancer which dynamically adjusts the granularity to increase performance
- The evaluation of our tuning mechanism on a many-core supercomputer environment as well as a multi-core Beowulf server

The remainder of this article is organized as follows. We first recall some useful context in Section 2 before introducing the design of our tuning mechanism in Section 3. We evaluate the capability of our tuning mechanism to automatically achieve performance obtained with manually adjusted settings in Section 4. We discuss some related works in Section 5 before concluding in Section 6.

2 | BACKGROUND

2.1 | Distributed Computation with APGAS

The Partitioned Global Address Space (PGAS)[4] is a programming model that allows programmers to manage a distributed environment. The address-space is split into several partitions (called “Places” in X10 terminology[5]) on which a process operates. The mapping of places to physical machines can be adjusted freely. A typical use in our situation generally involves a single process (or “place”) per physical machine but it is entirely possible to map several places to a single computer. The address space is “global” in the sense that any data present in one partition can be accessed remotely from another.

```java
import static apgas.Constructs.*;

import apgas.Place;

class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Running main at " + here() + " of " + places().size() + " places");

    finish(() -> {
      for (Place p : places()) {
        asyncAt(p, () -> System.out.println("Hello from " + here()));
      }
    });

    System.out.println("Bye");
  }
}
```

Listing 1: Distributed Hello World in Java

```
Running main at place(0) of 4 places
Hello from place(0)
Hello from place(3)
Hello from place(1)
Hello from place(2)
Bye
```

Listing 2: Sample output of the Hello World program of Listing 1 running with 4 places

The Asynchronous Partitioned Global Address Space (APGAS) programming model extends the PGAS model by integrating features in the language to manage asynchronous tasks operating on the various partitions. In X10[5], which implements the APGAS programming model, asynchronous activities are spawned and controlled by using specific keywords such as `async`, `at`, and `finish`.

More recently, the constructs used in X10 were ported to Java in a pure Java implementation[6]. The keywords of X10 were translated into static methods taking lambda expressions as parameters. Java programmers can now write X10-style programs by simply importing the APGAS for Java library into their program. The `asyncAt` method is used to spawn an asynchronous activity on the place specified as parameter. The `finish` method is used to wait until all the asynchronous activities (recursively) spawned within its closure complete.

A short distributed "Hello World" program with a sample output are presented in Listings 1 and 2. In this example, the main thread running at Place 0 will not progress further than the `finish` method until all the places (including itself) have written their message to the standard output before writing "Bye."

2.2 | The multithreaded lifeline-based load balancing scheme

Using the constructs of X10, the IBM team was able to develop a new load balancing scheme, the Lifeline-based Global Load Balancer[2]. Its main feature consists in establishing pre-determined channels for work stealing between places, the so-called "lifelines." When a place runs out of work and is unable to steal some from a randomly selected victim, it signals its "lifelines" that it needs some work and passively waits until either work is sent to it, or the computation completes. This mechanism allowed the load balancing scheme to maintain high efficiency even for large cluster sizes up to several thousand processors[2].

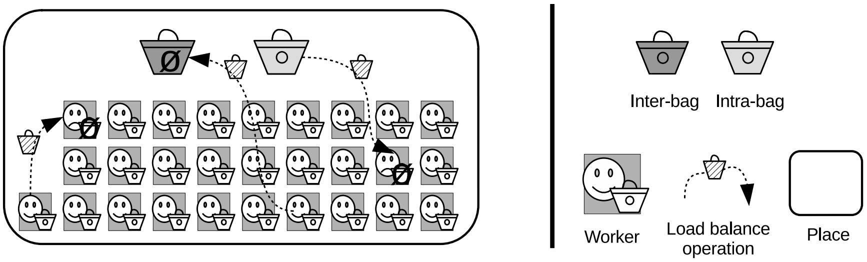

FIGURE 1 Overview of the multithreaded global load balancer operations within a place

A later evolution of this scheme, the multithreaded lifeline-based global load balancer[7] keeps the same computation abstraction and lifeline mechanism between places but makes each place run multiple worker threads in parallel instead of a single one as per the original scheme. With this scheme, it is no longer necessary to use multiple places (or processes) per host. Instead, a single place containing as many workers as there are cores on the underlying processor can be used. This also brings the opportunity for workers on the same host to easily share information as they operate in shared-memory. A graphical representation of the design including the main load balance operations that occur within a host is depicted in Figure 1.

Instead of making remote steals when a worker runs out of work, the remote steals are made when all the parallel workers on the host run out of work. Within a place, each worker holds its own dedicated bag instance throughout the computation. Load balance operations are achieved through the use of two additional bag instances that are not processed by any worker. One bag - the intra-bag - is primarily used to handle load unbalances within a host, while the second bag - the inter-bag- is used to handle steals attempts coming from remote places. The workers on the place collaboratively maintain some work available in both of these bags for a potential thief, be it a local worker or a remote place. In its original X10 implementation, this load balancer suffered from a few problems concerning the scheduling of messages. These were resolved in our Java implementation.

The abstractions provided to the programmer under both of these schemes are summarized in the `Bag` abstraction. In our Java implementation of the load balancer, it comes as an interface that programmers need to implement in their class that contain the data-structures that represent the computation at hand. The methods that programmers need to implement are the following:

- `void process(int, R)`: processes a certain amount of computation, that amount being specified by the integer parameter of the method. An instance of the result type `R` is also provided to the worker to read and/or write information shared with the other workers on the same host during the computation.
- `B split(boolean)`: returns a new bag instance containing a fragment of the computation held by the current bag. The boolean parameter is here to indicate that in the event the instance cannot be split, all of its contents should be given away.
- `void merge(B)`: merges the contents of the bag instance given as parameter into this instance.
- `boolean isEmpty()`: indicates if this bag is empty, i.e. if it does not contain any work.
- `boolean isSplittable()`: indicates if this bag can be further split, i.e. if work can be taken from it without emptying it altogether.

- `void submit(R)`: is called when the computation has ended and the result gathering phase begins. This method gives the opportunity to the bag to put its contribution to the final result into the instance provided as parameter.

To balance the load, a fragment of the computation can be obtained from a bag by calling the `split` method on it before being transferred and `merge`d into another bag instance. The `split` and `merge` methods' implementation is entirely left to the programmer. This grants complete control over the internal data structure used to represent the computation. The library remains oblivious to the data structure used by the bags, and while programmers are advised to program the `split` method such that half of the contents of the bag are given away, there is no actual mechanism to enforce it. The library guarantees that calls to the `split` and `merge` methods on any bag are made sequentially. Programmers do not have to concern themselves about potential concurrent accesses made to their Bag implementation as they do not occur in the multithreaded lifeline-based global load balancer.

3 | TUNING MECHANISM

There are several parameters that the user of the load balancing library can set to their preference. Parameters related to the distributed nature of the computation such as the "lifelines" seem to have satisfactory heuristics[2]. As for the number of concurrent workers per host, we set it to the number of cores available on the processor to use as much computing power as possible.

Our main contribution focuses on the task granularity. This is a more sensitive setting since this integer parameter does not have any meaning outside the context of a specific application. Setting an arbitrary value on the library side is not satisfactory. Also, we cannot expect users of the library to be able to predict what a good value would be for their application. The ideal grain size will vary depending on the problem at hand, as well as the size of the cluster used. Small changes to the implementation of a problem may also dramatically change the performance characteristics of a problem.

We aim at eliminating the need for users to guesstimate this value by integrating a tuning mechanism into the load balancer library that will automatically adjust the grain size to achieve good performance. In Section 3.1 we detail how the grain size influences the behavior of the load balancer. We then discuss the assumptions and heuristics on which our tuning mechanism relies in Section 3.2. Implementation details are briefly discussed in Section 3.3.

3.1 | Influence of the granularity on the Worker activity

The multithreaded global load-balancing scheme relies on several kinds of asynchronous activities to handle the distributed computation[7]. The details of the various interactions between these activities are beyond the scope of this paper. However, it is worth discussing the main routine of the "worker activity" along with some of the load-balancing mechanisms within a host as they are directly relevant to how our tuning mechanism operates. The main routine of the worker activity is presented in Listing 3. Note that some elements pertaining to synchronization were removed for clarity.

```
 1 void workerActivity (Bag workerBag) {
 2   do {
 3     do {
 4       // Step 1 - if able, spawn a worker activity
 5       if (Nb of running workers < workerLimit && workerBag.isSplittable()) {
 6         Bag fragment = workerBag.split(false);
 7         idleWorkerBag.merge(fragment);
 8         asyncAt(here(), ()-> workerTask(idleWorkerBag));
 9       }
10 
11       // Step 2 - if the intra-bag is empty and the worker's bag can be split, feed the intra-bag.
12       if (intraBagEmpty) { // volatile boolean flag
13         if (workerBag.isSplittable()) {
14           // Workers may accumulate here
15           synchronized (intraBag) {
16             // intraBagEmpty = false; // Previous design with "early" flag update
17             Bag b = workerBag.split(false);
18             intraBag.merge(b);
19 
20             intraBagEmpty = false; // Setting the flag later
21           }
22         }
23       }
24 
25       // Step 3 - if feeding the inter-bag is needed and the workerBag can be split, feed the inter-bag
26       // Step 4 - Check if there are remote thieves that can be answered
27       // Step 5 - Yield to load-balancing activities if needed
28 
29       // Step 6 - Do some work
30       workerBag.process(n, sharedResult);
31 
32       // Repeat from step 1 until the workerBag is empty
33     } while (!workerBag.isEmpty());
34 
35     // Step 7 attempt to steal from the intra-bag
36     if (the intra-bag is not empty) {
37       workerBag.merge(intraBag.split(true));
38       intraBagEmpty = intraBag.isEmpty(); // Update the volatile boolean flag
39     }
40       // Step 7-bis if unable to steal from the intra-bag attempt to steal from the inter-bag
41       else if (the inter-bag is not empty) {
42       workerBag.merge(interQueue.split(true));
43     }
44     // If work could be obtained, repeat from step 1.
45   } while (!bag.isEmpty());
46   // The worker could not get work from either bag, it stops.
47   // It may be spawned again by another worker performing step 1.
48 }
```

Listing 3: Worker activity main procedure

When an idle host receives work, the computation received is merged into one of the workers' bag and a first worker activity is spawned with that bag given as parameter. While the worker has some work in its bag, they will loop through steps 1 to 6 of their main procedure (lines 3 to 33 in Listing 3), with step 6 consisting in performing the computation. In its first step, the worker checks if it is possible to spawn an additional worker activity in the first step of its main routine. Provided this first worker's bag can be split, another worker activity will be spawned, which will in turn (recursively) spawn other workers until the maximum number of concurrent workers on the host is reached.

In steps 2 and 3, the workers try to maintain work in both shared bags on the place. If a worker performing step 2 finds that the intra-bag is empty and that it is capable of splitting its bag, the worker splits a fragment from its bag and merges it in the intra-bag. The inter-bag involved in step 3 follows a similar scheme. We do not detail steps 4 and 5 which are involved in guaranteeing the scheduling of work stealing activities coming from remote hosts.

When a worker runs out of work after performing step 6 and exits the inner do-while loop of its procedure, it will attempt to take some computation from the intra-bag in step 7, or as a last result from the inter-bag in step 7-bis, to immediately resume its computation (lines 35 to 44 in Listing 3). If the intra-bag gets emptied as a result, another worker performing step 2 will place some computation back into it. The next worker to run out of work will therefore be able to take some computation from the intra-bag again.

If a worker runs out of work when neither the intra-bag nor the inter-bag contain any work, it will escape the outer `do while` loop (line 45 in Listing 3) and terminate. A new asynchronous worker activity may be spawned back again by a worker performing step 1 of its main routine.

The attentive reader will have noticed the parameter "`n`" of the `process` method in step 6 of the worker's main procedure (line 30 in Listing 3). This integer determines the grain size. In general, it should be seen by programmers as the number of indivisible computation units to be performed in a call to method `process` before returning. As a consequence, the grain size is correlated to how much time workers spend in step 6, influencing how often they go through the inner do-while loop. If this parameter is low, the worker activities will go through their loop more frequently. Conversely if the chosen grain is large, workers will spend more time in step 6 and go through the loop less frequently.

3.2 | Heuristics

Diagnosis of a grain too large

The purpose of the two shared bags on the host is for workers that run out of work to steal from them and continue to participate in the computation. An issue that arises when the grain is too large is that when these queues become empty, there is a delay until a worker checks the queues status in steps 2 and 3 and puts some computation back into them. As a result, workers that run out of work are more likely not to be able to steal any work in step 7 and terminate, reducing throughput. These workers will eventually be spawned back by other workers performing step 1. However, this will also happen after a delay for the same reason. A situation where the grain size is too large on a host will therefore be characterized by intervals of time where fewer than the maximum number of concurrent workers are running.

As an indicator that the task granularity is too large, we use the proportion of the time spent with the maximum number of workers. Based on the characteristics of static grain executions of the Unbalanced Tree Search benchmark, we deem the grain size to be too large if less than 90 % of the elapsed time is spent with the maximum number of workers.

Diagnosis of a grain too small

As workers go through the inner loop of their procedure, various checks are made. These consists of reading some volatile boolean flags and calling methods of atomic data structures. These are quite lightweight, but they will still generate some overhead if they are made too often. Moreover, with many workers running in parallel, there is also an increased risk of contention on the bags used for load balancing when load-balancing operations are actually needed. Since these operations are made sequentially, we risk creating a bottleneck by using a grain size too low.

We developed two heuristics to detect cases where the grain is too low. Both rely on a subtle effect low grain executions have on the handling of the intra-bag. When a worker empties the intra-bag, it sets the boolean `intraBagEmpty` flag to `true` in line 38 of Listing 3. The next worker to perform the second step of its main routine will read the value of the flag as `true` in line 12 and (if able), place some work back into the intra-bag before setting the `intraBagEmpty` flag back to `false` in line 20. Any subsequent worker to perform the check in line 12 will read the updated value of the boolean flag and move on to the next step of its routine without placing work into the intra-bag.

However, it is possible for multiple workers to read the value of the `intraBagEmpty` flag as `true` in line 12 before the first worker sets it back to `false`. In this case, the workers that read the value of the flag before it was updated by the first worker will also place some work back into the intra-bag. This situation is more likely to occur if the workers very frequently check on the bag's status, as is the case when the grain size is low. As a consequence, the intra-bag is likely to be fed several times after getting emptied only once in situations where the grain is low. Our tuning mechanism attempts to leverage that fact to detect this situation.

We could eliminate the redundant feeding of the intra-bag, either by checking the value of volatile boolean flag again inside the synchronized block (lines 15 to 21 in Listing 3) or by changing our volatile boolean flag for an atomic integer. However, the redundant feeding of the intra-bag isn't a performance problem problem in itself. Rather, the fact that it occurs beyond a reasonable level is the sign that workers go through their loop too often, creating overhead. As our tuning mechanism detects this situation and increases the grain size, the problem disappears.

Early "split/merge" design

In the first design we introduced[3], we used the ratio between the number of times work is `split` from the intra-bag and the number of times work is `merge`d back into the bag to determine if multiple workers were able to redundantly feed the intra-bag. We call the tuner that relies on this criterion "split/merge tuner." This indicator, however, relies on the assumption that the programmer implements the `split` method of its bag such that successive calls to this method recursively give away half of the contents of the bag. Under this assumption, if the intra-bag is fed by multiple workers as a result of being emptied once, it will take comparatively fewer `split` calls to empty it again than it would have if a single worker placed work into the intra-bag each time it got emptied. In our previous work, we showed that we can use this metric in our tuning mechanism by setting an adequate "trigger" level below which the grain size is deemed too small.

New "merge/empty" design

We have since departed from this criterion and designed a new version of our tuning mechanism. We now directly measure the number of redundant feedings of the intra-bag by comparing the number of times the intra-bag was emptied in lines 37–38, with the number of times a worker puts work back into the intra-bag in lines 17–20 of Listing 3. We call this new criterion "merge/empty" ratio because in the context of our global load balancer library, it corresponds to the number of times workers `merge` work into the intra-bag divided by the number of times the intra-bag gets emptied. From a runtime perspective, it is the ratio between the number of workers that go through the `if` block in lines 13–20, and the number of times the boolean flag which guards this `if` block is set to `true`, allowing workers to enter it.

3.3 | Integration with the GLB runtime

The tuning mechanism is implemented as an extra asynchronous activity - the tuning activity - on each place of the global load balancer's runtime. The tuner is periodically called and remains inactive the rest of the time.

When the tuner is called, it directly reads the information accumulated in the load balancer's local logger. By comparing the current values with the ones from the previous time the tuner was called, the tuner is able to determine what took place during the last interval. It is then able to draw its conclusion and modify the grain size if necessary.

Initial experiments showed that the two indicators we use to detect if the grain is too low or too high are not infallible. Throughout the execution, there are times when the indicators draw the "wrong" conclusion or contradict themselves. As a moderation mechanism, we choose to only modify the grain size if the same conclusion is drawn by the tuner twice in a row.

When changing the value, we double (or divide) the current value by a factor 2. Combined with a tuning interval of 1 millisecond, this allows us to cover the very large range of values that the grain can take over a short period of time. We purposely set the initial grain size at a very low value of 10. The tuner activities of each place are free to adjust the grain as soon as the computation starts, and do so independently from one another. As a result, the chosen grain on two different compute nodes of the distributed computation may differ.

We did not witness any overhead imputable to this extra activity. This was checked by running distributed computations with the tuner activity active but keeping the chosen grain fixed. These executions produced the same execution time as regular fixed grain executions without this additional activity. This can be explained by the fact that the decision making takes an insignificant amount of computing power.

4 | EVALUATION

4.1 | Benchmarks used

To evaluate the performance of our tuning mechanism, we use four backtrack-search applications: N-Queens, Pentomino, the Traveling Salesman Problem (TSP), and the Unbalanced Tree Search (UTS).

We implemented these problems in a similar manner, using a pair of arrays to describe ranges of branches at each level of the exploration tree. Splitting the exploration is reduced to dividing this interval into two, matching the lowerbound of the thief to the upper bound of the victim for each layer of the exploration. This operation is therefore bounded in time and space by the depth of the exploration. As the branches are implicitly described, the size of the data transfered from host to host during load balance operations is independent from the actual amount of work transferred.

In this section, we briefly introduce each application and discuss some selected details about our implementations.

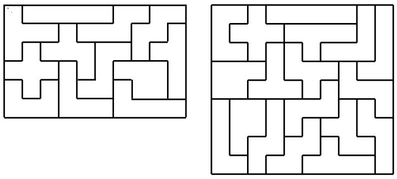

FIGURE 2 A particular solution to the Pentomino and the One-sided Pentomino problems

N-Queens

N-Queens consists in finding all possible arrangements such that a maximum number of queens are placed on a chessboard of width N without two queens able to attack one-another. We model the problem as an exact cover problem, which consists in finding all the different subsets of rows of a matrix of 0s and 1s such that for each column of the matrix, exactly one row has a 1 in that column. In the case of the N-Queens problem, the columns of the cover matrix correspond to the files, ranks, and diagonals of the chessboard. Each row in the matrix corresponds to a possible queen placement on the board and contains four 1s: one for the rank, the file, the diagonal, and the anti-diagonal that the queen occupies.

We use Knuth's "Dancing Links" data structure[8] to represent the sparse matrix of the cover problem. This data structure exploits doubly linked lists to hide and restore parts of the matrix as the backtrack exploration progresses. Exploration is made in a depth-first manner. At each step in the exploration, the first column of the cover matrix that remains to be covered is arbitrarily chosen. The various rows that can cover this column represent the options available in the exploration and can be explored in parallel.

Pentomino

The pentominoes are the 12 different shapes that can be formed by stitching 5 square tiles edge-to-edge. The Pentomino problem consists in enumerating all the possible ways to arrange these 12 shapes to cover a rectangle of width 10 and height 6. The One-sided Pentomino is an analogous problem but treats the face-down variations of the chiral pentominos as pieces of their own. As a result, the problem is significantly larger, consisting of arranging 18 pieces on a rectangular board of width 10 and height 9. A solution to the Pentomino and the One-sided Pentomino problems are presented in Figure 2.

In our implementation, we use a single array with sentinels to represent the rectangular board. We recursively attempt to place every rotation of every piece on the top-most and left-most unoccupied tile of the board. If the piece can be placed, the exploration proceeds and we attempt to place the remaining pieces of the board. If the chosen piece cannot fit on the board, the next orientation and/or piece is selected as a candidate. When all the candidates at a certain stage of the exploration have been attempted, the exploration backtracks by removing the last piece that was placed and selecting a new candidate. Some restrictions on piece placement and orientation can be made to eliminate the symmetries of the problem and only enumerate the fundamentally different solutions to the problem. This also reduces the size of the exploration tree. In terms of scaling potential, this problem shows a wide exploration tree and a low computation cost for exploring individual nodes. Of all our applications, the Pentomino has the greatest potential for strong scaling on larger clusters.

Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is an optimization problem which consists in finding the shortest round-trip through a number of cities. In our implementation, we use an exact Branch & Bound algorithm. The nearest neighbor heuristic is used to favor exploration of cities that are close to the current city first. The length of the best round-trip found so far is kept in a shared object on every place in the computation. Our library periodically checks on each place if a better bound has been found and, if so, recursively propagates this newly found bound to remote places through the lifelines.

We should note that the non-deterministic nature of the load balancer can introduce great variations in the execution times of this problem as it may influence how early better solutions are found and parts of the exploration tree trimmed. This problem is also prone to poor scalability when tested in strong scaling due to the parallel exploration of branches that would be trimmed out if a better bound was found.

UTS

The Unbalanced Tree Search[9] consists in a depth-first traversal of a randomly-generated tree. We use geometric trees in which the number of children of each tree node follows a geometric distribution of average 4. The resulting tree is unbalanced by construction as two nodes on the same level are unlikely to spawn similar size sub-trees. The size of the exploration is adjusted by setting a maximum depth to the tree. The shape and the size of the tree are entirely deterministic following an initial seed and the maximum depth. Parallel traversal is done by exploring nodes that have not been traversed yet and whose sub-trees have yet to be generated.

We conduct the evaluation of this problem in weak-scaling, that is, we use increasingly larger trees for increasingly larger clusters.

4.2 | On many-core clusters

We first perform an evaluation of the global load balancer library without the tuning mechanism on the OakForest-PACS supercomputer to determine the best performance achievable on each of our problems. Due to space constraints, we can only show a selection of the relationship between the grain size and the execution times on our problems in Figure 3. On each problem, we have a range of acceptable values grain values, which can be wide (such as Pentomino) or more narrow (like TSP and UTS). Although the range of acceptable grain sizes tend to overlap between different clusters, they can shift or get narrower as we increase the cluster size. We therefore need to conduct this thorough evaluation for all cluster configurations to identify the best performance possible for each of our problem. We then compare how both our tuning mechanisms fare against this best "Fixed Grain" executions. The main hardware characteristics of the OakForest-PACS supercomputer are summarized in Table 1 while the problem parameters used are shown in Table 2. The results are summarized in Figure 4. Part of the results presented here were first published in the PMAM'20 workshop[3].

As a general remark, both our tuning mechanism operate as intended on NQueens, Pentomino, the Traveling Salesman problems, and UTS, delivering close to the best fixed grain executions recorded.

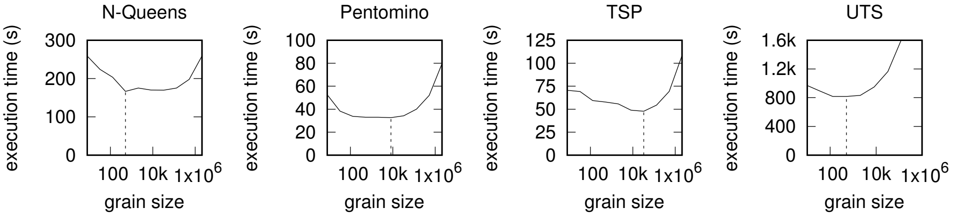

FIGURE 3 Relationship between the grain size and the execution time of our four benchmarks when running on 16 hosts of the OakForest-PACS supercomputer

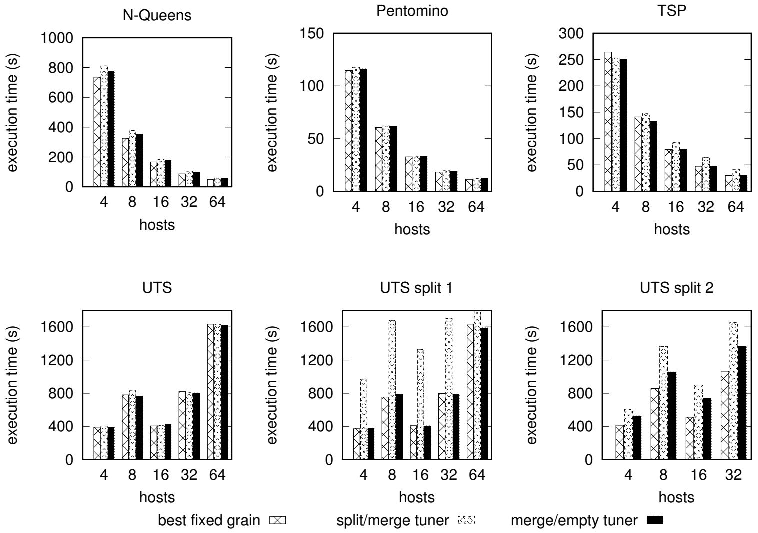

FIGURE 4 Execution time comparison between the best fixed grain and our "split/merge" and "merge/empty" tuning mechanisms on the OakForest-PACS supercomputer

We even achieve slightly better performance on the TSP when running on 4 to 16 hosts with our new "merge/empty" tuner. We also note that our new "merge/split" design shows identical performance (on Pentomino and UTS) or better performance (on N-Queens and TSP) than the "split/merge" tuner. This is a net improvement over our first design and is not the only advantage brought about by the new criterion, as we will discuss in the following section.

4.3 | Robustness of the merge/empty criterion over the split/merge criterion

To demonstrate the robustness of our new tuner design, we implemented two variants of the Unbalanced Tree Search (UTS) benchmark which differ in the implementation of their `split` method.

- UTS split 1: The intra-bag gives away its entire contents when the split method is called on it. As a result, the maximum split/merge ratio is 1, and lower if the intra-bag is redundantly fed as is the case at lower grain sizes.
- UTS split 2: The intra-bag makes a diagonal cut for each work fragment received. The number of split operation needed to empty the intra-bag is therefore proportional to the number of times it was fed. As a result, the split/merge ratio remains largely the same, regardless of if there were redundant merges made.

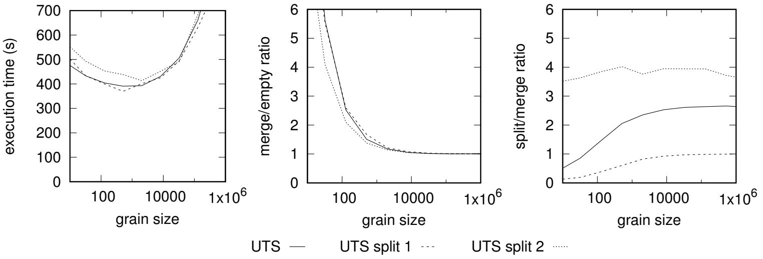

FIGURE 5 Execution time, split/merge, and merge/empty ratios of 3 implementations of the UTS benchmark depending on the fixed grain size chosen

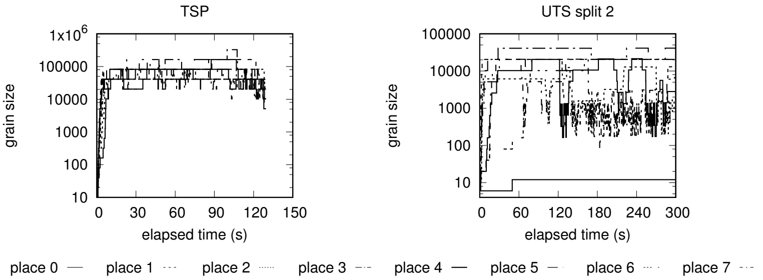

FIGURE 6 Evolution of the grain size on a 8 host execution of the Traveling Salesman Problem and the UTS split 2 benchmark on the OakForest-PACS supercomputer

The relationship between the grain size, the execution time, and the two ratios that we used in our tuning mechanisms to detect when the grain size is too low are presented in Figure 5. We can see that the execution time depends on the proper balance of the grain chosen for the execution. We can also recognize just how much the intra-bag is redundantly fed at lower grain sizes by looking at the "merge/empty ratio" plot. We note that the merge/empty curves of the regular UTS implementation and the two variants are almost identical. We therefore expect our new tuning mechanism which relies on this criterion to be able to accommodate for these vastly different splitting implementations.

On the contrary, the split/merge ratio on which our previous design relies shows great difference. We had chosen to describe programs whose split/merge ratio was below 2 as having a grain size too low. This worked for the original splitting implementation (labeled "UTS" in Figure 5) but will not for the other two implementations.

With the "split 1" and "split 2" variants, the "split/merge" ratio remains respectively below or above 2 regardless of the grain size used. As a result, our original "split/merge" tuner will fail to recognize the situation correctly for both of these problems and yield poor performance.

Looking at the results on the two UTS variants in Figure 4, the limits of our "split/merge" tuner become evident, sometimes presenting execution times more than double the best fixed grain achieved. By contrast, our new design yields execution times almost identical to the best fixed grain executions on the "split 1" variant. The largest gap occurred on the 16 hosts configuration with an execution time longer by just 30 seconds.

On the "split 2" variant, our new tuner design clearly outperforms our first "split/merge" design. However, we observe a certain performance gap to the best fixed grain executions with run times about 20 % longer (the largest gap being 40 % on the 16 nodes execution). This can be explained by the grain value chosen by our tuning mechanism. Figure 6 presents the evolution of the grain size as chosen my our "merge/empty" tuner on an eight host execution of the TSP and the UTS "split 2" variant. On the TSP execution, our tuner mechanism behaves as intended, increasing the grain size from its initial value on all the hosts. However, on the UTS split 2 variant, Place 0 keeps a very small value for the first 5 minutes presented here. It is only in the last five seconds of this execution (not shown in Figure 6) that the grain size on host 0 is finally increased. We are able to account for this phenomenon, which we discuss in Section 4.4.

4.4 | Limitations when load-balance operations are not needed

So far, we focused our analysis on clusters of many-core processors. However, nothing is preventing us from using our library on clusters of ordinary multi-core processors. In fact, this load-balancing scheme was first designed for such systems[7].

We were concerned that our tuning mechanism would not work with fewer workers per place. In particular, the criterion that determines if the grain size is too low relies on multiple worker threads entering the same branch of their main routine before one of them completes it. As we arbitrarily choose to launch our computation with a small grain value, our tuning mechanism may fail to raise the grain size from this original value to a satisfactory level by lack of contention in the workers' main procedure.

We evaluated the performance of our tuning mechanisms on our Beowulf server which is fitted with two 12-core Intel Xeon processors. The hardware details of this server are presented in Table 1. We evaluate the performance of our four benchmarks in three different configurations:

- 24(worker/process)x1(process)
- 12(worker/process)x2(process)
- 6(worker/process)x4(process)

The results are presented in Figure 7. With the exception of the UTS benchmark, executions with either of our tuning mechanisms show a performance gap compared to the best performance obtained with fixed grain executions. However, the cause of these gaps was not what we anticipated.

A detailed look at the grain chosen by our tuning mechanisms allows us to obtain more insights. Representative grain evolutions over time for each cluster configuration are presented in Figure 8. On the execution with a single process, the tuner keeps the grain size at its initial value of 10 for a long time. On the particular execution shown in Figure 8, only after 2 and a half minute have elapsed does the grain size is suddenly increased to the ideal range around 10 thousands.

We can explain this phenomenon by the nature of the criteria used to detect that the grain is too low. In its current design, the intra-bag needs to be emptied for the tuner to obtain data to be able to make its decision. In such an execution where all the work is concentrated on the single process participating in the computation, the workers all obtain a large amount of work to begin with. They will therefore be able to keep going through their loop without running out of work for a long time. Moreover, a large amount of computation is likely to have been placed in the intra-bag. This means

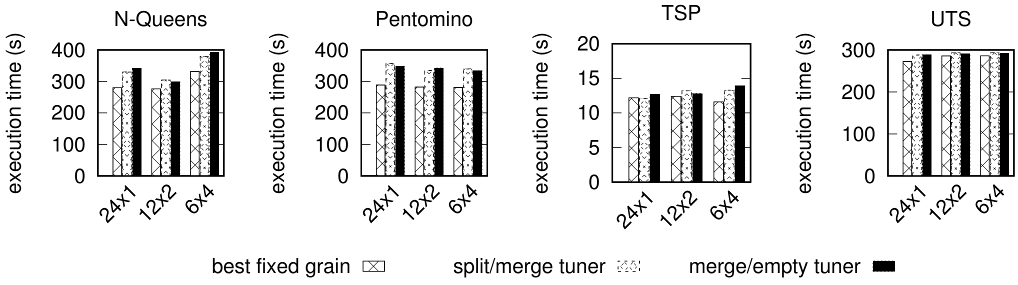

FIGURE 7 Execution times of our four benchmarks in the three configurations on our beowulf server

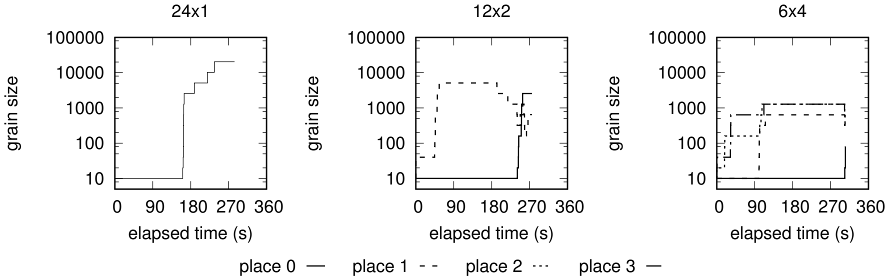

FIGURE 8 Evolution of the grain size over time of the TSP problem with our "merge/empty" tuner on the three cluster configurations of our Beowulf server

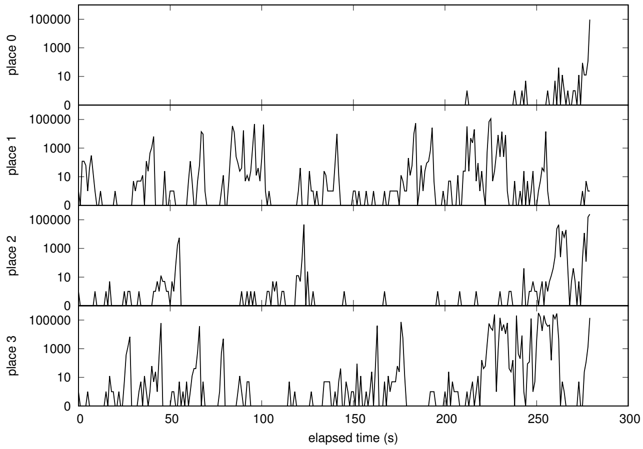

FIGURE 9 Evolution of the number of times the intra-bag is emptied on each place of a four-place execution of the Pentomino problem on our Beowulf server

that even when the workers start running out of work, it may also take a while to empty the intra-bag a first time. Only when load imbalance actually occurs on the single place of the computation is the tuning mechanism able to recognize that the grain is too low and finally increases it.

We see a similar pattern on the two- and four-process executions, with place 0 keeping its grain value at its initial value of 10 up to the very end of the computation where it finally jumps. On the other places however, the grain size is increased by the tuning mechanism from the start. This can be explained by the fact that these places obtain a fragment of the computation held by place 0 through their lifeline steals. This fragment being smaller, they encounter load imbalance very soon in the computation.

This is confirmed by how frequently the intra-bag is emptied throughout the computation. If we focus on Figure 9, we can see that the places 1, 2, and 3 face situations where the intra-bag is emptied from the start of the computation. By contrast, the intra-bag is emptied on place 0 after 240 seconds have elapsed in the 270 second execution. This reassures us in the capability of our tuning mechanism to operate on systems with fewer concurrent workers as it appears that the reduced number of worker per place is not what causes the tuner to fail. Rather it is the lack of load imbalance that prevents it from recognizing situations where the grain is too low.

We believe this also explains the performance gap we saw with our "merge/empty" tuner design on the UTS split 2 variant on the OFP supercomputer. The particular implementation of the UTS split 2 variant (which yields little work when the `split` method is called on a bag) causes the workload to have greater difficulty trickling down from the bags used for load balancing on place 0. As a result, it will take longer for the intra-bag of place 0 to get emptied and for the tuning mechanism to detect the overhead.

We have attempted to resolve this problem by spuriously setting the `intraBagEmpty` flag to `true`, making workers feed the intra-bag as if it had being emptied (see lines 12–23 in Listing 3). However, these efforts have yet to come to fruition.

Conclusion on the new "merge/empty" tuner design

Our new tuning mechanism is more robust than our first design and is capable of correctly adjusting the grain size of the computation on both cluster of many-core processors and on more common multi-core environments. However, to be able to successfully achieve this, it requires some load balance operations to take place. In cases where the computation remains concentrated on a single host, either through an inadvisable work `split`ting implementation (such as the UTS "split 2" variant) or by running on a small (maybe single host) cluster, it is possible for our current design to miss the presence of overhead and keep grain sizes that are too low to deliver good performance.

5 | RELATED WORKS

The most popular programming model for parallel computation relies on the Fork/Join model. Typical implementations in shared memory processor rely on a pool of threads, with each thread having its own queue of tasks to process. Tasks can generate new tasks which are added to the worker's queue. When a worker runs out of work, it attempts to steal tasks from a neighboring thread to resume its computation. Several works elaborate on this scheme to reduce the overhead due to the task creation, such as Wang et al[10], or influence the tasks stolen to favor cache consistency, as in LAWS[11] and Constrained Work Stealing[12]. Min, Iancu and Yelick also present their own implementation of a distributed task library over UPC. In HotSLAW[13] they define a hierarchy that matches the characteristics of the (distributed) hardware at hand (cache, socket, and node level). Workers that run out of work try to steal on workers that are close to them first, only stealing from workers further away in the hierarchy if failing to obtain some from close workers. Moreover, they adapt the number of tasks stolen at each level, with closest level steals stealing only 1 or 2 tasks and remote steals stealing half of the tasks contained in the queue. This is a characteristic not supported under our current global load balancer design as we do not control how much work is transferred when a bag is `split`.

Our tuning mechanism bears some resemblance with the adaptive grain mechanism presented by Cong et al. in XWS[14]. They reuse the task-parallelism model of Cilk and enhance it with the capabilities of the X10 language to target graph algorithms. In their target applications, each node of the graph at hand represents one task. Coarsening is achieved by grouping the nodes in the workers' queues in batches. Working threads always process and steal batches in their entirety. The appropriate batch size for each worker is dynamically adjusted following a heuristic on the current size of its queue. The abstractions offered by the Lifeline-based GLB are different. First, while programmers may choose to implement their bag as double-ended queue of tasks, there is not obligation for them to do so. Secondly, our library guarantees that bags are only manipulated by a single thread at a time. Load balance operations using either of the shared work reserve are done in mutual exclusion whereas in XWS, a worker is allowed to steal from another while it is processing a batch. Moreover, in GLB there is no relationship between the size of the grain used and the amount of work which can be stolen from a bag.

Hiraishi, Yasugi, Umatani, and Yuasa introduced their own programming and execution framework dedicated to backtrack applications called Tascell[15]. By introducing specific constructs into a C program, they eliminate the need for the programmer to explicitly spawn tasks. Instead when a worker runs out of work, their runtime automatically makes one of the concurrent workers backtrack its search to a fork-able stage in the exploration tree, perform the necessary work transfer, and return to its prior state in the exploration tree. Their approach eliminates the costly data copies that are necessary when preemptively spawning tasks as load balance is only performed on a "per need" basis. They are also able to maintain a certain degree of data locality by maintaining a fixed "workspace" for each worker thread. In these two regards, our framework approach is also effective.

Our implementations of the exploration problems have some similarities with the techniques described by Leroy et al[16]. In their article, the authors describe a technique that allows them to number all the nodes of a permutation combinatorial problem using a single interval, regardless of where they are located in the tree. A single pair of integers for the lower bound and the upper bound is therefore sufficient to describe an entire sub-tree and splitting the exploration is reduced to merely splitting this single interval. In our implementations, we rely on two arrays to describe the intervals of nodes for each level of the exploration tree. This allows us to represent the tree in a more compact way than a pool of nodes kept in a doubly-ended queue. Splitting the tree is done by duplicating these lower and upper bound arrays and matching the lower bound of the thief to the upper bound of the victim, effectively giving away half of the unexplored nodes of each level.

Posner and Fohry also experimented with the APGAS for Java library[17] on which our implementation of the Global Load Balancer relies. They introduced a new construct to the library, `asyncAny`, which creates an asynchronous task that can be executed on any place of the cluster. They then use the same lifeline scheme to dynamically relocated these asynchronous tasks between compute nodes. Their implementation of the TSP problem uses the same heuristics as ours but requires many array copies as they spawn as many asynchronous tasks as there are nodes in the first layers of the exploration tree. As a result, their implementation is about twice as slow as ours.

All the approaches mentioned above focus on spreading the work to keep as many computing resources busy for as long as possible. In Palirria[18], Varisteas and Brorsson tackle the reverse problem consisting of matching the computing resources to the (varying) degree of parallelism of an application. On single applications, they are able to maintain ideal performance with higher efficiency by adjusting the number of allocated cores to the computation based on its potential for parallelization. Similar to our approach, they are able to make a decision on whether to change the number of cores allocated to their program based on the measurement of some selected runtime metrics over the most recent elapsed interval. Their approach makes it possible to envision several programs running concurrently and making the most of the available computing power of a shared memory processor. Transposing their approach to distributed environments seems feasible.

6 | CONCLUSION

We integrated a tuning mechanism into our Java implementation of the multithreaded lifeline-based global load balancer which dynamically adjusts the grain size of the computation at hand based on some selected runtime metrics. We evaluated the capability of our tuning mechanism to adjust the granularity on four backtrack-search applications on a many-core supercomputer and a Beowulf server. We manage to automatically obtain ideal performance on all four of our benchmarks in our supercomputer environment. We also established the robustness of our tuning mechanism against significant variations in problem implementation.

Finally, we have identified a limit to our current system which requires load imbalance to be able to correctly adjust the granularity of the computation at hand. In future work, we will attempt to integrate other criteria into the decision process of our tuning mechanism to make it capable of handling situations of reduced load imbalance.

ACKNOWLEDGMENTS

This research is partly supported by MEXT as “Exploratory Challenges on Post-K computer (Studies of multi-level spatiotemporal simulation of socioeconomic phenomena)”. This research used computational resources of the Oakforest-PACS supercomputer provided by the Joint Center for Advanced High Performance Computing (JCAHPC) through the HPCI System Research project (Project ID: hp160253).

References

1. Bak S, Menon H, White S, Diener M, Kalé LV. Multi-Level Load Balancing with an Integrated Runtime Approach. In: IEEE; 2018: 31–40

2. Zhang W, Tardieu O, Grove D, et al. GLB: Lifeline-Based Global Load Balancing Library in X10. In: PPAA ’14. Association for Computing Machinery; 2014; New York, NY, USA: 31-40

3. Finnerty P, Kamada T, Ohta C. Self-Adjusting Task Granularity for Global Load Balancer Library on Clusters of Many-Core Processors. In: PMAM ’20. Association for Computing Machinery; 2020; New York, NY, USA

4. De Wael M, Marr S, De Fraine B, Van Cutsem T, De Meuter W. Partitioned Global Address Space Languages. ACM Comput. Surv. 2015; 47(4). doi: 10.1145/2716320

5. Tardieu O, Herta B, Cunningham D, et al. X10 and APGAS at Petascale. In: PPoPP '14. ACM; 2014; New York, NY, USA: 53–66

6. Tardieu O. The APGAS Library: Resilient Parallel and Distributed Programming in Java 8. In: X10 2015. IBM. Association for Computing Machinery; 2015; New York, NY, USA: 25–26

7. Yamashita K, Kamada T. Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10. Journal of Information Processing 2016; 24(2): 416-424. doi: 10.2197/ipsjjip.24.416

8. Knuth DE. Dancing links. eprint arXiv:cs/0011047 2000.

9. Olivier S, Huan J, Liu J, et al. UTS: An Unbalanced Tree Search Benchmark. In: Almási G, Caşcaval C, Wu P. , eds. Languages and Compilers for Parallel ComputingSpringer Berlin Heidelberg; 2007; Berlin, Heidelberg: 235–250.

10. Wang L, Cui H, Duan Y, Lu F, Feng X, Yew PC. An Adaptive Task Creation Strategy for Work-stealing Scheduling. In: CGO '10. ACM; 2010; New York, NY, USA: 266–277

11. Chen Q, Guo M, Guan H. LAWS: Locality-aware Work-stealing for Multi-socket Multi-core Architectures. In: ICS '14. ACM; 2014; New York, NY, USA: 3–12

12. Lifflander J, Krishnamoorthy S, Kale LV. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In: ; 2014: 857-868

13. Seung-Jai M, Costin I, Katherine Y. Hierarchical Work Stealing on Manycore Clusters. In: PGAS '11. ; 2011.

14. Cong G, Kodali S, Krishnamoorthy S, Lea D, Saraswat V, Wen T. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In: ; 2008: 536-545

15. Hiraishi T, Yasugi M, Umatani S, Yuasa T. Backtracking-based Load Balancing. In: PPoPP '09. ACM; 2009; New York, NY, USA: 55–64

16. Leroy R, Mezmaz M, Melab N, Tuyttens D. Work Stealing Strategies For Multi-Core Parallel Branch-and-Bound Algorithm Using Factorial Number System. In: PMAM'14. Association for Computing Machinery; 2014; New York, NY, USA: 111-119

17. Posner J, Fohry C. Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library. The Journal of Supercomputing 2018; 74. doi: 10.1007/s11227-018-2234-8

18. Varisteas G, Brorsson M. Palirria: Accurate On-line Parallelism Estimation for Adaptive Work-Stealing. In: PMAM'14. ACM; 2007; New York, NY, USA: 120:120–120:131

How to cite this article: P. Finnerty, T. Kamada, and C. Ohta (2020) A self-adjusting task granularity mechanism for the Java lifeline-based global load balancer library on many-core clusters, CCPE, 2020;00:1–6

TABLE 1 Characteristics of the environments used for our evaluation

Characteristics	Oakforest-PACS	Beowulf cluster
Number of servers	8208	1
CPU	Intel Xeon Phi 7250 (Knights Landing)	Intel Xeon E5-2680 v3
CPU per node (threads)	1 (68)	2 (12 + 12)
Frequency	1.4 GHz	2.5 GHz
Memory	96 GB(DDR4) + 16 GB(MCDRAM)	135 GB (DDR4)
Interconnections	Intel Omni-Path (100 Gbps)	
Java version	OpenJdk v1.8.0_222	OpenJdk v1.8.0_172

TABLE 2 Problem settings used for the experiments involving varying number of workers on the Oakforest-PACS supercomputer and our Beowulf server

Problem	Oakforest-PACS	Beowulf server
UTS	Branching factor: 4, Depth: 18 on 4 hosts, 19 on 8 and 16 hosts, 20 on 32 hosts, 21 on 64 hosts	Branching factor: 4, Depth: 17
Pentomino	One-sided, Board width: 9, Board height: 10, with symmetry removal	One-sided, Board width: 9, Board height: 10, with symmetry removal
N-Queens	$N = 19$	$N = 17$
TSP	35 cities	24 cities