**Noname manuscript No.**
(will be inserted by the editor)

# Unbalanced Tree Search on a Manycore System using the GPI programming model

**Rui Machado · Carsten Lojewski · Salvador Abreu · Franz-Josef Pfreundt**

Received: date / Accepted: date

**Abstract** The recent developments in computer architectures progress towards systems with large core count (Manycore) which expose more parallelism to applications. Some applications named irregular and unbalanced applications demand a dynamic and asynchronous load balance implementation to utilize the full performance a Manycore system. For example, the recently established Graph500 benchmark aims at such applications. The UTS benchmark characterizes the performance of such irregular and unbalanced computations with a tree-structured search space that requires continuous dynamic load balancing. GPI is a PGAS API that delivers the full performance of RDMA-enabled networks directly to the application. Its programming model focuses the use of one-sided asynchronous communication, overlapping computation and communication. In this paper we address the dynamic load balancing requirements of unbalanced applications using the GPI programming model. Using the UTS benchmark, we detail the implementation of a work stealing algorithm using GPI and present the performance results. Our performance evaluation shows significant improvements when compared with the optimized MPI version with a maximum performance of 9.5 billion nodes per second on 3072 cores.

**Keywords** GPI · Work Stealing · Load Balancing · UTS · Manycore

Rui Machado
E-mail: rui.machado@itwm.fhg.de

Carsten Lojewski
E-mail: lojewski@itwm.fhg.de

Franz-Josef Pfreundt
E-mail: pfreundt@itwm.fhg.de
Fraunhofer Institut Techno-und Wirtschaftsmathematik
Competence Center for High Performance Computing
Kaiserslautern, Germany

Salvador Abreu
E-mail: spa@di.uevora.pt
University of Evora
Evora, Portugal

## 1 Introduction

The development of parallel applications requires different optimization and programming techniques to match the different types of applications. Within the whole range of applications, some applications possess characteristics which allow to classify them as irregular and unbalanced. Examples of such applications include optimization problems or heuristic search problems and are important in different domains such as SAT solving and machine learning.

The characteristics of such applications include unpredictable communication, unpredictable synchronization and/or a dynamic work granularity. One common requirement is a asynchronous and dynamic load balancing scheme because the inherent unpredictability of the computation does not allow a static partitioning of work across the computing resources.

The UTS benchmark [1] aims at the characterization of such unbalanced computations and at measuring their efficiency in terms of load balancing. It accomplishes this using a search space problem where a large tree of parametrized characteristics is traversed. The tree traversal generates imbalance during run-time according to the tree characteristics, therefore requiring a implementation that minimizes this imbalance efficiently. This includes low communication overheads and low idle times.

GPI stands for Global address space Programming Interface and is a PGAS (Partitioned Global Address

Space) API targeted at RDMA-enabled interconnects such as Infiniband. It provides a different programming model than the message passing paradigm of which MPI is the most widely used standard. GPI focuses on one-sided communication and the development of more asynchronous algorithms, leveraging the capabilities of modern interconnects to overlap communication with computation. As GPI focuses on the communication, we developed the ManyCore Threading Package (MCTP) to harness the computation power of recent systems composed of several cores. The MCTP is a threading package based on thread pools that abstracts the native threads of the platform and provides advanced features for multithreaded programming. The GPI programming model presents an alternative for the development of parallel applications running on modern systems. It already showed an advantage in some types of parallel applications with excellent results [4].

In this paper we are interested and focus on the dynamic and asynchronous load balancing problem required by irregular applications. More concretely, we present an implementation of work stealing using GPI and evaluate our implementation using the UTS benchmark which targets exactly that. The evaluation used two different and large (up to 300 billion nodes) tree configurations and was performed on a recent many-core system and we demonstrate the scalability of our implementation on up to 3072 cores. In both cases the GPI version outperforms the MPI version by a maximum factor of 2.5 in terms of raw performance (number of nodes processed per second), reaching a maximum performance of 9.5 billion nodes per second.

This paper is organized as follows: the following section presents some related work on the topic. In section 3 we briefly describe the UTS problem as a representative of the problems we are interested in. In section 4 we introduce GPI, the framework used to address the problem. We then describe our implementation in section 5 and present the results of our performance evaluation in section 6. Finally, we conclude our work in section 7 and discuss some future work.

## 2 Related work

Load balancing is a central aspect of parallel computing that as been studied and analyzed many times in the literature. In [13] several schemes for scalable load balancing are presented and analyzed for a variety of architectures. More recently, in [15] the authors discuss the new challenges in dynamic load balancing and how they address them with Zoltan [16]. A common problem concerns task scheduling of tasks organized as a task graph and dynamic and irregular task tree [14].

Work stealing, as a method for efficient load balancing, has been explored and used in different contexts. The seminal work [5] considers a shared memory setting where tasks are stolen when the dequeue of a processor becomes empty. This work is used in Cilk [12].

Recently and aimed at distributed memory machines, work on the X10 [10] programming language presented XWS [11], the X10 Work Stealing framework. The XWS extends the Cilk work-stealing framework which include several features to implement graph algorithms, global termination detection, phased computation and more.

Using the UTS benchmark as a representative of unbalanced computations that require dynamic load balancing has been explored for different programming models, exploring their main features and devising suitable techniques that match the programming model. In [6] and [7], dynamic load balancing using message passing (MPI) is examined using two approaches (work stealing and work sharing). An UPC [8] implementation of the UTS benchmark is presented and evaluated in [3]. Also following a PGAS approach, ARMCI [9] in this case, the work in [2] aims at the implications and performance of a design targeted at scale.

## 3 UTS - Unbalanced Tree Search

The Unbalanced Tree Search (UTS) benchmark was designed to represent applications requiring substantial dynamic load balance. The problem is rather simple: the parallel exploration of an unbalanced tree, by counting the number of nodes in an implicitly constructed tree that is parametrized in shape, depth, size and imbalance. Applications that fit this pattern include many search and optimization problems that must search through a large state space of unknown or unpredictable structure.

The tree is implicitly generated where each node in the tree can be generated by the description of its parent. Each node in the tree is represented by a node descriptor which is the result of applying the SHA-1 secure hash algorithm to the descriptor of the parent of the node together with the child index of the node. With this generation method, UTS defines different tree types that represent different search types or problems and different load imbalance scenarios.

One interesting point about UTS is the different implementations available. There are MPI implementations (different approaches), UPC, shmem, OpenMP and more. And all the implementations are optimized to take advantage of the features of each programming model, creating an interesting comparison point for new implementations.

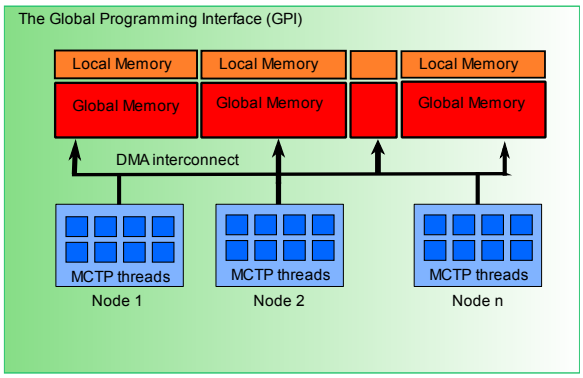



Fig. 1: GPI architecture

## 4 GPI

GPI (Global address space Programming Interface) is a PGAS API for parallel applications running on clusters. The thin communication layer, delivers the full performance of RDMA-enabled networks directly to the application without interrupting the CPU.

The figure 1 depicts the architecture of GPI.

The local memory is the internal memory available only to the node and allocated through typical allocators (e.g. malloc). This memory cannot be accessed by other nodes. The global memory is the partitioned global shared memory available to other nodes and where data shared by all nodes should be placed. The DMA interconnect connects all nodes and is through this interconnect that GPI operations are issued. At the node level, the MCTP is used to take advantage of all cores present on the system and make use of the GPI functionality and global memory. We developed the Manycore Threading Package (MCTP) in order to help programmers to take better advantage of new architectures and facilitate the development of multithreaded applications. The MCTP is a threading package based on thread pools that abstracts the native threads of the platform.

GPI is constituted by a pair of components: the GPI daemon and the GPI library. The GPI daemon runs on all nodes of the cluster, waiting for requests to start applications and the library holds the functionality available for a program to use: read/write global data, passive communication, global atomic counters, collective operations. The two components are described in more detail in our previous contribution [4]. *(Note: GPI was previously known as Fraunhofer Virtual Machine (FVM)).*

In the context of this work, the important functionality is the read/write of global data and global atomic counters.

Two operations exist to read and write from global memory independent of whether it is a local or remote location. The important point is that those operations are one-sided and non-blocking, allowing the program to continue its execution and hence take better advantage of CPU cycles. If the application needs to make sure the data was transferred (read or write), it needs to call a wait operation that blocks until the transfer is finished and asserting that the data is usable.

Global atomic counters allow the nodes of a cluster to atomically access several counters and all nodes will see the right snapshot of the value. There are two operations supported on counters: fetch and add and fetch, compare and swap. The counters can be used as global shared variables used to synchronize nodes or events. As an example, the atomic counters can be used to distribute workload among nodes and threads during run-time. They can also be used to implement other synchronization primitives.

## 5 Dynamic load balancing with GPI

The UTS benchmark requires a asynchronous and dynamic load balancing solution. As referred, we choose a work stealing strategy to address this.

Work stealing is a relatively simple algorithm. It is triggered every time a thread runs out of local work. An application taking advantage of a work stealing algorithm usually enters the following states:

**Working**
: While a thread has work, it keeps itself busy. In the case of UTS that translates to visiting nodes, generating child nodes and add them to the work list.

**Work stealing**
: When the work is all processed and the thread will fall into an idle state, it looks for a victim to steal work from and if it finds a potential one, it performs a steal. How the search for a victim and the actual steal operation are performed is implementation dependent.

**Termination detection**
: If work stealing fails that is, no victim thus no work is found, the thread enters termination detection. Termination detection is a topic in distributed computing *per se* and several algorithms exist.

The GPI implementation of the UTS benchmarks focuses therefore on the work stealing and termination detection stages. We leverage the previous work on UTS with MPI and other implementations, taking them a starting point for our own implementation.

A common aspect to the implementations is the use of a data structure that is partitioned into two regions:

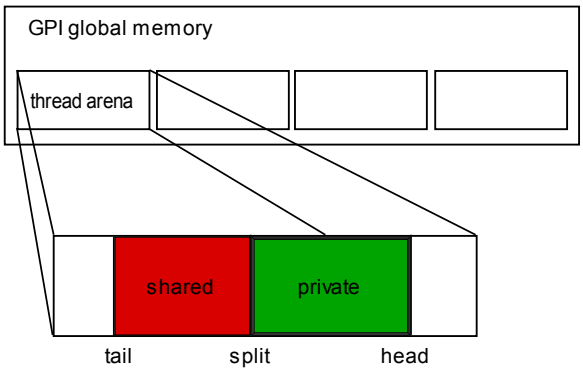



Fig. 2: Data structure and GPI memory placement

a private and shared. The private region holds the work of a worker thread and this is not available to others threads while the shared region holds the work available to be stolen. The GPI implementation uses a similar implementation of this data layout. Because we want to take advantage of GPI, we completely implemented the data structure on the global memory of GPI. This allows global availability of the data and meta-data. Moreover and because the global memory of GPI is already available from the start of the application, the data structure operations (e.g. add, remove) become cheaper since no calls to the allocator (malloc) are made and it is just a matter of working with offsets of the global memory. The figure 2 depicts the organization of the data structure and its placement the global memory of GPI.

A thread adds and removes work packages (nodes) to its private region. This translates to a simple movement of the head since there is no synchronization on this private region. When the private area hits a parametrized threshold, it releases a chunk of work to its shared region. This translates to a simple movement of the split pointer towards the head. A thread can also re-acquire chunks of work from its shared region when it exhausts the work on its private region, by moving the split pointer towards the tail. Finally, when a thread performs a steal, it does it at the victim's tail and moves it towards the head. Since it is a shared region, some mutual exclusion mechanism is required.

From a single thread point of view, the whole program control structure follows **Algorithm 1**.

A GPI implementation usually has two levels. The MCTP level is the single node case and in **Algorithm 1** translates to steps 0, 1 and 2. The GPI level takes care of the remote case which translates to step 3 and most of step 4.

Each MCTP thread has its own arena organized as described above. For steps 0 and 1, each thread acts on the data structure placed on its own arena, adding,

**Algorithm 1** Program control structure

```
   while ! done do
     while there is work do
       consume work
       generate (if that is the case) new work and save it
5:     share some work if there is a surplus on the private
       area
     end while
     {Step 1: re-acquire}
     if there is work on the shared region then
       re-acquire it and go back to work
     end if
     {Step 2: local steal}
10:  if local steal is successful then
       go back to work
     end if
     {Step 3: remote steal}
     if remote steal is successful then
       go back to work
15:  end if
     {Step 4: termination detection}
     enter barrier and termination detection
   end while
```

removing work and re-acquiring it when needed. When they run out of work, threads must look and steal work (Step 2) from other threads. The thief thread can and does peek the status of the other thread's arena and if the shared region has more than a chunk of work, it is a potential victim. Because a mutual exclusion mechanism is required, the thief thread locks the data structure (each data structure has one lock to access its shared region and pointers), makes sure the work is still available and modifies the tail of the victim. Finally, the lock is released and the thief can move the stolen work to its own private region. The choice of the victim and whether a local steal happens follows a rather simple heuristic: the thief circulates over all other threads and if a surplus of work is found it immediately takes that thread as its potential victim.

When a MCTP thread does not find any local work to steal it tries to steal from a remote thread (Step 3) and has to resort to GPI.

### 5.1 Remote work stealing

In the remote case, the work stealing operation relaxes the meaning of a steal (a thief usually steals without the victim to know it) and requires the participation of the victim. The participation of the victim come as a requirement due to the need of mutual exclusion on the access to the shared region of the victim. The remote thief is potentially trying to steal work concurrently with other nodes and the other threads on the same node of the victim.

Our implementation applies a request/polling strategy: the thief requests and the victim polls and responds to requests. This ensures that the access to the victim's shared region is atomic since the victim itself will perform it.

Each worker thread has the added responsibility of handling steal requests from other nodes which are directly targeted at it. Added to the normal program control structure, each thread polls for pending remote steal requests. If it finds a request, a reply to the thief is sent. The reply takes the form:

- a work chunk
- no surplus of work but the node still has private work
- no work at all

If there is work available, it comes from the shared region and the victim performs a local steal to itself. The work chunk is reserved for the steal request and the victim issues a remote write directly to the thief's private region. This communication is a non-blocking one-sided step that is queued and offloaded to the HCA allowing the victim to immediately return to its normal work loop.

Responding that there is no work to steal but that there is private work allows the thief to better evaluate the status of the victim. As it will be mentioned below, this is useful for detecting termination.

From the thief perspective, the remote work steal takes two simple steps: find the victim and send the request in case work is found. To find the victim, the thief thread takes advantage of the one-sided read primitive of GPI (RDMA) to read the status of the remote node. This is accomplished by reading some meta-data of all threads on the remote node and finding the one with surplus of work. Here the heuristic is simple: if one thread has surplus of work it becomes the potential victim and the request is sent to it. The victim only receives a request on the very probable case of having surplus of work, diminishing the possibility of a negative answer from the victim. This reduces the communication overhead and the waiting time of the thief for a negative answer. If the whole victim node is out of work, the thief tries another potential node until it tried all nodes. The thief tries all nodes in a ring pattern, starting at the node where it performed the last steal.

### 5.2 Termination detection

The current implementation uses a simple termination detection algorithm. When a thread finds no work to steal there is potential for the termination state. As with work stealing, the termination detection works at the two levels, local and remote.

The local termination detection level works as follows: when worker threads of each node are not able to find work to steal they enter a local cancel-able barrier which allows them to return to the stealing state in case new work is made available. All but the last thread of the node stay in this unsellable barrier. The last thread on each node, by acknowledging that the node is completely idle and no work was found, enters a second level of the termination detection.

The second level of termination detection, global termination, is handled by one single thread. The reason for this is - using the current simple implementation - to avoid that all idle threads wildly keep looking for work and thus putting a high pressure on the interconnect.

The last thread keeps looking for the availability of work on remote nodes and trying to steal a chunk. And it knows if the remote nodes still have private work since the response from the remote node includes both situations. When this last thread realizes that all nodes are out of work - they all responded with "there is no work at all" - it increases an atomic counter by one and waits until this atomic counter reaches the total number of nodes. The global atomic counters of GPI are used for this termination flag. As all nodes increase the termination flag, the last thread on each node responsible for global termination detection warns the other worker threads waiting on the barrier that termination has been reached and they can exit.

On the other hand, if the last thread waiting for termination detection finds and steals some remote work, it cancels the barrier on where the other local threads are waiting making them return to the normal program loop.

### 5.3 Prefetching work

GPI provides asynchronous communication primitives geared towards higher performance, allowing the application to overlap computation and communication. Any application making use of GPI should therefore exploit this feature.

The remote steal operation is implemented with a split-phase non-blocking semantics where a request has two distinct and independent phases (steps). In the first phase, the request is submitted and the function immediately returns (non-blocking) not waiting for the request completion - the calling worker thread is free to continue its execution with some other work. In the second phase, a request is then checked for completion.

We take advantage of this and implemented a work pre-fetch step in order to overlap the communication involved in remote steal response with the normal program structure.

The prefetch step is triggered if the two following conditions are met:

- the worker thread tries to acquire work from its shared region and the work left there is smaller than a chunk of work - the thread is running out of work.
- there is a imminent remote steal - the threads on the same nodes do not have enough shared work for a local steal.

If the conditions are met, one remote steal request is submitted solely to the neighbor node (the nodes are paired in a ring). When the threads actually runs out of work, it first checks if a prefetch request was issued and in that case, checks its completion and answer. If the answer was positive, the thread avoids trying to steal work from other threads and can immediately continue.

Although this adds some small extra overhead to the worker thread (submitting the request, checking if a prefetch was issued), we observed a 95% success ratio between submitting a remote steal request and getting a positive response on the overall prefetch steps performed.

## 6 Experimental results

In this section, we present experimental results obtained on the evaluation of our implementation.

All results were obtained on a system of up to 256 nodes where each node is equipped with a Intel Xeon X5670 CPU ("Westmere") running at 2.93GHz with 6 cores and 12MB of L3 cache. Each node has two of such processors providing 12 threads (Hyper-Threading is disabled) making up to 3072 threads on the whole system. The nodes are connected via a Mellanox MT26428 QDR (40 Gb/sec) Infiniband card.

The UTS problems used for the performance evaluation are two, representing two different types and sizes: a geometric tree of about 270 billion nodes and a binomial trees with size of about 300 billion nodes.

We compare our implementation to the work stealing MPI implementation of UTS due to the general availability of MPI and since it is the standard for the development of parallel applications. The MPI implementation used was MVAPICH2 1.5.1.

The performance evaluation was run up to 3072 cores (threads) with 256 nodes. For each node setup, the execution made use of the Full node or Half node. Full node means that we use the maximum number of physical cores available (12) and Half means only half of those (6).

The reason behind this differentiated test is to observe the threads contention effects on the overall performance. Since we are using a mixed scheme for local and remote steals and the single node steals imply a locking mechanism, having a larger number of threads (cores) should present some contention effects.

The figure 3 presents the results obtained for a geometric tree of about 270 billion nodes.

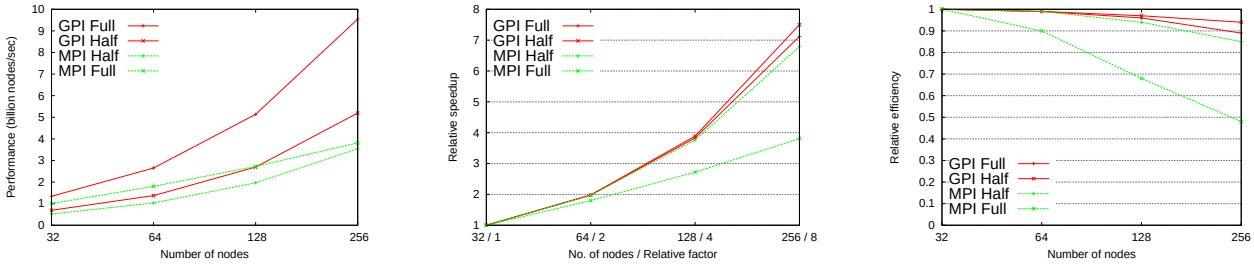
The GPI version scales well reaching a peak performance ( shown in figure 3a) of around 9.5 billion nodes per second which represents a 2.5 times speed-up factor over the MPI implementation (MPI best is 3.8 billion p/sec).

Comparing the two approaches in terms of cores used (Half or Full), we see that GPI behaves as expected that is, using more cores yields better performance. In fact and although the number of cores doubles between the Half and Full versions, using all cores attains 84% performance improvement over using half of the cores.

The MPI version suffers harder from the number of threads used, where using all cores only yields a 8% improvement over using half the cores in the largest number of nodes.

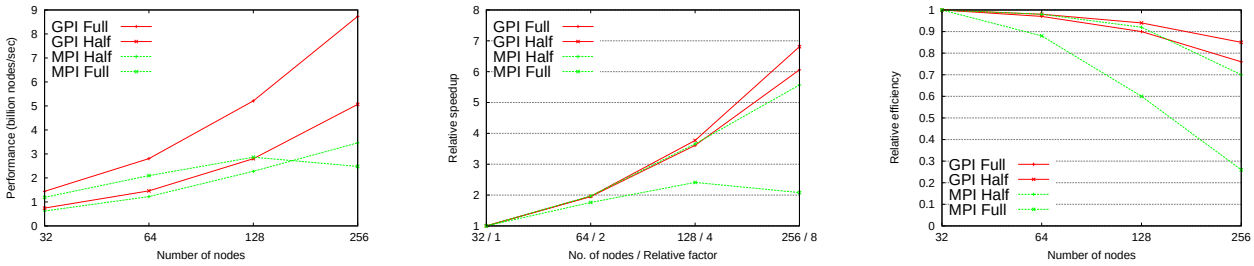
The figure 3b presents the obtained relative speedup to using 32 nodes as origin and, directly related, the figure 3c presents the relative efficiency when scaling the problem from 32 nodes (origin) to 256 nodes. Both GPI Half and GPI Full obtain high speedup (close to the maximum 8) with a relative efficiency above 89% in all cases. Worth noting is that using all cores yields slightly lower speed-up factor and efficiency than just using half of the cores. This is an expected result since we have a fixed problem size and the efficiency should decrease because of the extra overhead involved. This observation is only noteworthy since we interested in measuring the difference and quantify that extra overhead. In this case, the values are acceptable as we observe an decrement in efficiency from 94% (GPI Half) to 89% (GPI Full) with the largest node count.

The MPI implementation has two faces: using half of the cores available (half) yields good results, with a speedup close to the GPI implementations and a relative efficiency above 85% in all cases. On the other hand, using all available cores (Full) and although obtaining higher performance, demonstrates scalability problems since the maximum obtained speedup only reaches a factor of 3.81 and with a rapidly decreasing efficiency that lowers to 48% at the largest core number. In the MPI case, increasing the node count for a fixed problem size decreases by a much larger margin the efficiency.




(a) Performance (b) Relative Speedup (c) Relative Efficiency

Fig. 3: Performance on Geometric Tree ( 270 billion nodes)




(a) Performance (b) Relative Speedup (c) Relative Efficiency

Fig. 4: Performance on Binomial Tree ( 300 billion nodes)

The figure 4 depicts the largest problem - about 300 billion nodes. In this case, it is a binomial tree which presents higher load balancing requirements.

Again, GPI scales well on both versions and the performance difference between using all or half of the cores is acceptable - the worst case, using 256 nodes, using all cores achieves a 72% improvement over using half of the cores. The speedup values are not so high as with the Geometric tree problem but that is acceptable since this Binomial tree problem imposes higher load balancing requirements. Nevertheless, we see a 6.06 maximum speedup factor (at 256 nodes) which represents a 76% efficiency when taking 32 nodes as a starting point.

On the other hand, MPI has some problems at the largest core count. Using half of the available cores even yields better performance than using all cores. This is more evident on the speedup plot (figure 4b) and on the efficiency plot (figure 4c with a lowest point of 26% of relative efficiency.

## 7 Conclusion and future work

In this paper we presented our current work on applying the GPI programming model to one of the problems raised by irregular applications namely, dynamic load balancing. Moreover, we focus on a recent large system, where each node can handle up to 12 threads of execution in a total of 3072 threads and evaluate our implementation on it. Our objective was to evaluate such a programming model on the dynamic load balancing problem using recent hardware and design a solution that could improve the results obtained with current implementations on other programming models such as message passing.

We use the UTS benchmark as a representative of that class of problems and evaluated two different kinds of workloads, geometric and binomial trees. In both cases the GPI version outperforms the MPI version by a maximum factor of 2.5 in terms of raw performance (number of nodes processed per second). The performance results (9.5 and 8.7 billion nodes for the geometric and binomial trees, respectively) represent the best values obtained for the used platform that we are aware of. In terms of speedup and efficiency, we observed encouraging results.

One important aspect that we focused on and which revealed an interesting outcome, was to evaluate the performance when using all and only half of the cores

on each node of the system. Recent systems employing a cc-NUMA system architecture demand more care on the exploitation of system resources which if ignored might yield surprising results. Ideally, the performance obtained running the same problem on all cores of a node should yield doubled the performance when running it on half of the cores. In practice this is almost never the case. In our case, our GPI implementation always behaves acceptably where, in the worst case (binomial tree on 256 nodes), using all the cores yields a 72% performance improvement over using only half of the cores. On the other hand, the MPI implementation reflects better the mentioned problem and has scalability issues on the binomial tree case: using all the 3072 cores performs worse than using only half of the cores on the same node count and with a relative efficiency rapidly decreasing as the number of nodes increases.

This comes as a confirmation on the difficulties when implementing or porting algorithms to modern systems but we show that a rather simple implementation of an efficient algorithm as work stealing with very simple heuristics can benefit from the GPI programming model. The mixed nature of the programming model, handling the local and remote cases, allows to better exploit the increasing hierarchy of parallelism in more recent cluster systems. Also, by concentrating efforts in developing more asynchronous algorithms and overlapping computation and communication using the primitives of GPI results in excellent performance.

The obtained results lead us to conclude that GPI gives a very good solution to the proposed problem. While there is room for optimization, the obtained knowledge and implemented algorithms will be useful when we turn to the development of real applications that present the same parallelism requirements.

As future work we intend to do deeper experimentation with our implementation: work splitting (how much does the thief steal from the victim needs) or victim choice heuristic are details that need to be better examined and that might result in performance improvements. We intend to apply this knowledge on real workloads. More generally, we continue to improve the GPI framework and to better adapt to developments in system architectures and interconnects.

## References


1. Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark Proc. 19th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC). New Orleans, LA, November 2-4, 2006.
2. James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, P. Sadayappan Scalable Work Stealing Proc. 21st Intl. Conference on Supercomputing (SC). Portland, OR, Nov. 14-20, 2009.
3. Stephen Olivier, Jan Prins. Scalable Dynamic Load Balancing Using UPC. Proc. of 37th International Conference on Parallel Processing (ICPP-08). Portland, OR, September 2008.
4. Machado, R., Lojewski, C.: The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. Computer Science-Research and Development 23(3), 125132 (2009)
5. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Ann. Symp. Found. Comp. Sci. (1994) 356368
6. J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In Proc. of 6th Intl. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS), pages 18, 2007.
7. J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. A message passing benchmark for unbalanced applications. J. Simulation, Modelling Practice and Theory, 16(9):1177 1189, 2008.
8. UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
9. J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. Lecture Notes in Computer Science, 1586:533546, 1999.
10. P. Charles, C. Grotho, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA), pages 519538, 2005.
11. G. Cong, S. Kodali, S. Krishnamoorty, D. Lea, V. Saraswat, and T. Wen. Solving irregular graph problems using adaptive work-stealing. In Proc. 37th Int Conf. on Parallel Processing (ICPP), Portland, OR, Sept. 2008.
12. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In Proc. Conf. on Prog. Language Design and Implementation (PLDI), pages 212223. ACM SIGPLAN, 1998.
13. Kumar, V., Grama, A.Y., Vempaty, N.R.: Scalable load balancing techniques for parallel computers. J. Par. Dist. Comp. 22(1) (1994) 6079
14. Chakrabarti, S., Yelick, K.: Randomized load- balancing for tree-structured computation. In: IEEE Scalable High Performance Computing Conf. (1994) 666673
15. K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challanges in dynamic load balancing. J. Appl. Numer. Math., 52(2-3):133152, 2005.
16. Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courtenay Vaughan. 2000. Design of dynamic load-balancing tools for parallel applications. In Proceedings of the 14th international conference on Supercomputing (ICS '00). ACM, New York, NY, USA, 110-118. DOI=10.1145/335231.335242 http://doi.acm.org/10.1145/335231.335242