Exploiting Hierarchical Locality in Deep Parallel Architectures

AHMAD ANBAR, OLIVIER SERRES, and ENGİN KAYRAKLIOĞLU,

The George Washington University, Washington DC

ABDEL-HAMEED A. BADAWY, The George Washington University, Washington DC,

and Arkansas Tech University, Russellville, AR

TAREK EL-GHAZAWI, The George Washington University, Washington DC

Parallel computers are becoming deeply hierarchical. Locality-aware programming models allow programmers to control locality at one level through establishing affinity between data and executing activities. This, however, does not enable locality exploitation at other levels. Therefore, we must conceive an efficient abstraction of hierarchical locality and develop techniques to exploit it. Techniques applied directly by programmers, beyond the first level, burden the programmer and hinder productivity. In this article, we propose the Parallel Hierarchical Locality Abstraction Model for Execution (PHLAME). PHLAME is an execution model to abstract and exploit machine hierarchical properties through locality-aware programming and a runtime that takes into account machine characteristics, as well as a data sharing and communication profile of the underlying application. This article presents and experiments with concepts and techniques that can drive such runtime system in support of PHLAME. Our experiments show that our techniques scale up and achieve performance gains of up to 88%.

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers, Optimization, Run-Time Environments; D.1.3 [Concurrent Programming]: Distributed Programming

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: PGAS, hierarchical locality exploitation, productivity, PHLAME, PHAST

ACM Reference Format:

Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed A. Badawy, and Tarek El-Ghazawi. 2016. Exploiting hierarchical locality in deep parallel architectures. *ACM Trans. Archit. Code Optim.* 13, 2, Article 16 (June 2016), 25 pages.

DOI: <http://dx.doi.org/10.1145/2897783>

1. INTRODUCTION

With deeper memory systems and more complex interconnection hierarchies, parallel computers are becoming more and more difficult to program by domain scientists. Handling data locality and reducing the amount of communication are complex tasks. It is now imperative and inevitable for the research community to appreciate the new

Extension of a conference article in PGAS 2015 proceedings [Anbar et al. 2015]. Preliminary results were presented at ICPADS 2014 [Anbar et al. 2014].

Authors' addresses: A. Anbar, The George Washington University, 20101 Academic Way, Suite 333, Ashburn, VA 20147, Washington DC; email: anbar@gwu.edu; O. Serres, The George Washington University, 20101 Academic Way, Suite 333, Ashburn, VA 20147, Washington DC; email: serres@gwu.edu; E. Kayraklioglu, The George Washington University, 20101 Academic Way, Suite 333, Ashburn, VA 20147, Washington DC; email: engin@gwu.edu; A.-H. A. Badawy, The George Washington University, 20101 Academic Way, Suite 333, Ashburn, VA 20147, Washington DC, and Arkansas Tech University, Russellville, AR; email: abadawy@atu.edu; T. El-Ghazawi, The George Washington University, 20101 Academic Way, Suite 333, Ashburn, VA 20147, Washington DC; email: tarek@gwu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1544-3566/2016/06-ART16 \$15.00

DOI: <http://dx.doi.org/10.1145/2897783>

Fig. 1. Bandwidth characterization between two threads at each level for different message sizes on a Cray XE6 supercomputer.

realities of data locality and engender new abstractions and methodologies that can enable handling and managing hierarchical locality.

As parallel computer architecture advances, systems are scaled in two dimensions. This is accomplished in one dimension by increasing the number of cores and nodes. Scaling is also achieved in another dimension by the creation of deeper memory hierarchies and interconnection networks. All the layers in the system hierarchy have different characteristics in terms of latency, bandwidth, and affinity. Special hardware features are also present to support certain functions such as Remote Direct Memory Access (RDMA), which allows data transfers to be very efficient. Thus, the cost that parallel activities incur for communication varies widely according to the communication layers and the type of communication. To illustrate the importance of understanding the characteristics of different hardware layers, we ran an experiment on a Cray XE6 supercomputer to measure the bandwidth between two threads at different relative distances. Unexpectedly, the bandwidth between two remote threads is higher than the bandwidth among local threads given that there is no network contention. This happens only at message sizes larger than 16KB. Figure 1 depicts the results of this experiment. This unforeseen behavior is attributed to the Block Transfer Engine (BTE) in the Gemini network chip of the Cray XE6, which speeds up such large messages. The experiment described here only uses point-to-point communication between a pair of cores, which is not representative of the general case and should not be relied on for system characterization.

Locality-aware parallel programming models, such as Partitioned Global Address Space (PGAS) [PGAS 2015], expose data locality to enable programmers and domain scientists to enforce data affinity, or first-order data affinity, that is, the association between a given computing activity/thread and the data that it needs the most. To fully exploit the capabilities of the underlying hardware, it is no longer sufficient to only consider “first-order locality.” In addition, the relative amount of interactions between every pair of activities should be used to decide how far they should be placed from one another given the system hierarchy.

While many agree that this is a pressing need, we strongly believe that viewing this as the programmer’s responsibility will not only limit productivity but also affect performance portability in many cases.

We demonstrate the importance and expected gains from leveraging hierarchical locality using a synthetic benchmark on a Cray XE6 as shown in Figure 2. The benchmark is written in UPC and used 144 threads in total (24 threads per node on 6 nodes). The benchmark distributes a shared array across threads. Each thread makes a fixed number of accesses that are divided between owned and non-owned data. The ratio

Fig. 2. A synthetic benchmark showing the gain of thread placement on the proper node (blue) and on the proper chip (red).

Fig. 3. Synthetic benchmark: Effects of scaling and remote communication ratio on gains from exploiting hierarchical locality.

between the owned and non-owned number of accesses is shown on the x -axis. The baseline is measured for each of these ratios while the affinity of the threads is randomly assigned to cores. The blue bars show the gains due to the best placement of the threads at the node level, that is, properly choosing which threads belongs to each node. The red bars show the gains of the best placement at the chip level by grouping the threads with higher communication on the same chip. Thus, each color represents the expected amount of performance improvement if the correct placement decisions are taken at the corresponding level.

To further investigate the potential performance gain from our techniques as we scale out the number of cores in the machine, we use a second synthetic benchmark on a Cray XE6. The results of the benchmark are presented in Figure 3(a) and (b). The synthetic benchmark distributes data uniformly across the compute threads. Then, the compute threads are either placed randomly or intelligently on the system. The compute threads will then perform some accesses to their own data and some accesses to remote data that belong to other thread(s). The accesses patterns are constructed such that heavily communicating groups of threads are formed, that is, groups that have high degree of locality. The performance gain provided by the intelligent placement (based on our proposed model) is then evaluated under different conditions: read/write, different ratio of local to remote accesses (x -axis), and the number of threads (z -axis or depth axis). The y -axis represents the gain of running the benchmark with groups assigned to the same node compared to running with the default randomized placement of threads as decided by the system. It is clear that hierarchical locality becomes remarkably significant with the scaling of the number of threads and remote data access ratio.

Fig. 4. A Cray XC40 Dragonfly Interconnection Hierarchy.

Figure 3(a) shows gains of up to 37-fold for read accesses, while Figure 3(b) shows gains of up to 32-fold for write accesses.

Figure 4 shows the hierarchical dragonfly topology of the Cray XC40 supercomputer as an example of a modern architecture. There are five locality levels shown in the figure: four levels originate from the interconnect and one level emerges from having dual sockets within the node. According to the benchmark results above, heavily communicating groups of threads should be placed in a way to maximize their throughput bandwidth. For example, on the Cray XC40, this can be realized by placing these thread groups within the same node or within the same Processor Daughter Card (PDC). However, misplacing those threads can have a steep performance penalty. We believe strongly that the penalty will increase as systems grow deeper and wider.

In this work, we propose the Parallel Hierarchical Locality Abstraction Model for Execution (PHLAME, “flame”). Starting from the affinity exposed by the locality-aware programming models, PHLAME exploits the machine hierarchical characteristics through a runtime framework. The runtime framework combines machine characteristics and the data sharing/communication profile of the application in order to guide the automatic exploitation of hierarchical locality using different strategies. PHLAME is loosely referred to as an execution model, as its concepts, metrics, and algorithms are used to create a hierarchical locality-driven thread execution environment. We also propose the PHLAME Adaptive Selection Test (PHAST, “fast”) algorithm, which is used to pick the best strategy for a given application and machine combination.

The work described in this article makes the following contributions:

- Proposes PHLAME, the Parallel Hierarchical Locality Abstraction Model for Execution: a descriptive model that captures the deep hierarchical architecture characteristics of parallel systems. The model is utilized by a runtime framework, along with the application’s communication profile, to produce a hierarchical thread mapping according to different placement strategies.

- Introduces PHAST, the PHLAME Adaptive Selection Test algorithm: an adaptive algorithm that employs the hierarchical communication costs to test and predict the best placement strategy.
- Demonstrates the effectiveness of PHLAME and PHAST for large systems, where the performance gains increase as the number of cores scales up for a varying benchmark suite of varying communication patterns and complexity.

The rest of this article is organized as follows: Section 2 discusses previous related work relevant to hierarchical locality. Section 3 presents the PHLAME model. Section 4 presents our experimental system and our results in detail. We will conclude the article and discuss future work in Section 5.

2. RELATED WORK

Hierarchical locality exploitation has been gaining attention, and many recently started projects are underway. One such initiative, Dynamic Exascale Global Address Space (DEGAS) [Yelick et al. 2015], is a project to research a unified programming model for Exascale systems. It proposes the use of a hierarchical programming model both for the hierarchical locality exploitation (for the memory stack and the network) and for the control (creation of new threads in a hierarchical fashion). Similarly, Habanero-C is a hierarchical PGAS language [Majeti et al. 2014; Habanero-C 2015] that allows the programmer to dynamically create new threads and control their locality by using hierarchical place trees. The programmer specifies where in the tree the new threads should be spawned. Another approach is to use specific libraries like the Zoltan toolkit [Devine et al. 2002]. The program is expected to be written in MPI and the programmer is expected to describe the problem to be partitioned through Zoltan API.

We believe that while all the above approaches tackle the problem in one good possible way and can yield good performance, they leave programmers and domain scientists with the big burden of reasoning and writing code that specifies many levels of parallelism and locality at all machine levels.

In our approach, the user specifies only the first level of locality. The knowledge generated from that step is then used to extract the hierarchical locality information automatically through straightforward profiling of the application. Then, in conjunction with the underlying PHLAME machine model, the runtime system can make informed placement decisions for exploiting locality at the other levels of the hierarchy. This considerably increases the ease of use of the system.

A large body of related research targeted topology-aware communication optimization. A closely related effort was that of Jeannot et al. [2014]. They presented the TreeMatch algorithm, which is somewhat similar to our clustering algorithm. However, their clustering does not take into account the hierarchical characteristics of the machine as they did not consider the various communication costs at each level, whereas hierarchy is an essential element in our machine abstraction model. Thus, their technique does not cater to deeply hierarchical modern architectures, which is our primary concern here. Another distinction is that their technique is based on the aggregate communication size and does not account for the underlying message sizes and number of messages, whereas, in our approach, we account for the different message sizes, which captures the communication cost more precisely. Thus, we make a distinction between patterns that have many small messages as opposed to patterns with few large messages. Jain et al. [2014] presented a model that can predict traffic on a dragonfly network, based on the process placement and the application communication pattern. Nevertheless, their model did not aim at improving the placement for the target applications. Zheng et al. presented a heuristic in Zheng et al. [2015]

Fig. 5. PHLAME implementation flow.

to maximize the throughput of parallel applications in cloud/grid environments. However, their work was targeting parallel applications that are represented as directed acyclic graph (DAG). Wu et al. proposed a hierarchical task mapping for adaptive mesh refinement (AMR) workloads [Wu et al. 2012]. Several efforts targeted topology-aware optimization for collective communication operations only such as in Prisacari et al. [2013], Sack and Gropp [2012], Subramoni et al. [2011], and Bruck et al. [1997].

A lot of work [Molina da Cruz et al. 2011; Cruz et al. 2012] considered locality exploitation under locality-unaware shared memory programming models. This, however, is a different problem, as the core of these efforts was to guess the first level of locality and not the hierarchical grouping.

Molina da Cruz et al. [2011], on the other hand, presented a framework for detecting shared accesses in an Symmetric Multiprocessing (SMP) node running parallel programs written in OpenMP [Dagum and Menon 1998]. Their work was simulation based. They used the Simics simulator [Magnusson et al. 2002] to be able to record all memory sharing patterns among threads. This is neither a software nor a programming model technique and therefore it does not address the issues at hand.

It should be noted that none of these later techniques would be applicable in locality-aware programming languages such as UPC [El-Ghazawi et al. 2003], MPI [Gabriel et al. 2004], SHMEM [Poole et al. 2011], or Chapel [Chamberlain et al. 2007].

Experimental characterization of the target-machine-relevant properties is another area of related work. Broquedis et al. [2010] presented Portable Hardware Locality (hwloc), a framework to detect and report portable abstractions of the hierarchical architecture of modern hardware, such as NUMA nodes, sockets, shared caches, cores, and so on. However, it does not consider some critically needed information like the memory bandwidth. Our PHLAME, on the other hand, captures and abstracts all the machine characteristics from the cores to the shared memory or interconnection network in relationship to hierarchical locality. This allows us to accurately rank the different placement choices.

3. THE PARALLEL HIERARCHICAL ABSTRACTION MODEL FOR EXECUTION (PHLAME)

3.1. Design Philosophy

PHLAME is an execution model that is designed to facilitate the hierarchical locality exploitation, shown in Figure 5. As an execution model, PHLAME defines at a high level the strategy for performing computations. It also determines the functionality of different components of the system stack. PHLAME focuses on reducing the inter-communication overhead in deeply hierarchical architectures by exploiting hierarchical locality. The functionalities of PHLAME map into a locality-aware programming model

Fig. 6. PHLAME division of responsibilities.

and a runtime system. The objectives of PHLAME are to (1) organize activities (e.g., threads) such that each activity is co-located with the data that it requires and (2) place every pair of activities at a hierarchical distance that is consistent with the degree of interaction between them.

PHLAME's philosophy is built around two fundamental concepts, as shown in Figure 6: (1) trust the programmers or the domain scientists to have full control over the data affinity to the parallel threads (this is what we call “first-order locality”) and (2) refrain from asking the programmer to also handle the higher-order locality in a hierarchical architecture as this would be a great hindrance to productivity.

For the first point, the rationale is that the programmers or domain scientists are the experts in knowing the problem they are trying to solve, and they should be granted the ability to decide the best way to distribute the data across threads. Therefore, a locality-aware programming model is in the core of PHLAME. The two widely accepted programming models that expose locality-awareness are message passing and PGAS. While PHLAME does not mandate a certain programming model, we believe that with shared address space and one-sided communication constructs, PGAS languages will potentially (as they reach maturity) provide higher performance gains and productivity with PHLAME.

To be able to produce rational mappings, PHLAME utilizes descriptive models that capture the machine and application characteristics. The machine model captures the characteristics of the different layers in the machine hierarchy. Similarly, the application model captures the application communication profile. The characterization is done completely offline and the associated overhead will be therefore transparent to the user. Such offline overhead is also incurred only once for a given machine as long as the machine does not change (e.g., adding more nodes or more memory to the existing nodes) and only once for a given problem size of an application. PHLAME fuses these two models to obtain mappings using different strategies that rely on different graph partitioning methodologies. PHLAME requires testing and evaluation of the different mappings. The testing and evaluation should be performed analytically without having to run the application under each mapping. Finally, the PHLAME runtime system will adaptively select and apply the best mapping for a specific parallel task.

Our model here is not a structural/architectural model that differentiates between the details of how a cache versus network router work. It is, rather, a behavioral, performance characterization that focuses on the cost of the data movement (latency) as seen by the underlying programming model, regardless of the specifics of the hardware units involved. For example, for MPI workloads, this characterization is based on sends and receives, for UPC workloads, this characterization is based on puts and gets. In establishing such a characterization, we deal with such hardware units as black boxes:

Fig. 7. Levels and locales of a Cray XC40.

They are given an excitation (a data size to move) and latency is measured. What exactly happens inside of a cache or a network interface has no bearing on the decision; only the associated total latency across all levels involved is what matters.

While in this article we do not present a full end-to-end working realization of the PHLAME stack, we provide a robust proof of concept as we established the core components at all the levels to experimentally evaluate the effectiveness of the concepts and techniques proposed as reported in the next section.

3.2. Basic Definitions

- Locale: A locale is a group of execution modules (or cores in the first level of the hierarchy) where any pair of cores within the same locale can interact with the same consistent latency. A locale in the context of PHLAME is a hardware feature. A locale at Level₀ just consists of a single core. We define locales recursively from the bottom-up. Level _{l} locale is formed by grouping N locales from the lower level _{$l-1$} .
- Level: A level _{l} is formed from all locales that are at the same vertical distance from the root. For example, level₀ is usually formed of all cores. Levels are generally delineated by the communication hardware, which can be through shared memory or an interconnection network. As an example, the levels and locales of XC40 are shown in Figure 7.
- Locale Size at Level _{l} : the maximum number of level _{$l-1$} locales that can fit in a level _{l} locale. For example, the maximum locale sizes at level₂ (Node Level) in XC40 is two, as each node locale can fit two chip locales from level₁, as shown in Figure 7.
- Locale Mass at Level _{l} : Locale Mass is the total number of cores that belong to a locale at level _{l} . For example, in the XC40, shown in Figure 7, if the chip contains 12 cores, then the mass of a chip locale is 12 and the mass of a node locale is 24, and so on.
- Message Cost at Level _{l} : the time to deliver a message of size b bytes between two cores that belong to the same locale at level _{l} and belong to different locales at lower levels. For instance, to measure the cost at the blade level, the measurement should be done between cores that belong to different nodes.
- Partition: Partitions are used to describe groupings of parallel activities that can fit in one locale at a given level. A partition at level _{l} cannot exceed the locale mass at the same level _{l} . Thus, partitions at level₀ can only contain a single task, assuming no over-scheduling is allowed.
- Hierarchical Locality Exploitation: creating partitions of parallel activities at different levels that ideally will result in reduction of the overall communication cost.

3.3. Formal Treatment

Being an execution model, PHLAME establishes the functionality of different components of the system stack [Lucas et al. 2014]. In this section, we define a general framework that can be used to realize different PHLAME components.

3.3.1. Machine Description and Characterization. The goal of this step is to extract the hardware properties of the underlying machine architecture. This is achieved by determining the different parameters specific to each machine:

- Machine Description: captures general parameters, such as the total number of cores and the total number of levels in the machine. Hierarchical description of the levels in terms of locale sizes and locale masses will also be included.
- Machine Characterization: captures the costs of different message sizes at each level. This can be described as the cost vector for level l and is given by the following equation:

$$\mathbf{C}_l = \{c_1, c_8, c_{16}, \dots, c_s\}, \quad (1)$$

where $\mathbf{C}_l$ is the vector storing the message costs for different message sizes at level l and c_s is the cost of sending a message of size s bytes on that level.

The overhead of describing and characterizing a machine is only incurred once only for a particular machine. The output can be stored in a file that is made available to other PHLAME components.

3.3.2. Application Profiling. The application profile is done by extracting the communication characteristics between each pair of activities. As PHLAME relies on locality-aware programming models, the runtime associates each piece of data to an activity. Therefore, the communication can be precisely inferred by the runtime, even when it happens across the shared memory and caches. In order to reduce both the memory and the computational requirements, two vectors are used to represent the communication patterns. The message size spectrum is partitioned into bins, such that each bin corresponds to a sub range. For example, bin 0: (1 byte $\rightarrow$ 63 bytes), bin 1: (64 bytes $\rightarrow$ 255 bytes), and so forth. And for each bin, two values are collected: (1) the average message size in the bin, to be captured in the first vector, and (2) the total number of messages between the pair of activities that belong to that bin, to be captured by the second vector. The two vectors can be represented by the following equations:

$$\mathbf{A}_{ij} = \{a_0, a_1, \dots, a_B\}, \quad (2)$$

$$\mathbf{N}_{ij} = \{n_0, n_1, \dots, n_B\}, \quad (3)$$

where i and j are the ranks of threads/activities, whereas $\mathbf{A}_{ij}$ and $\mathbf{N}_{ij}$ are the vectors storing the average message sizes and the number of messages in each bin, for bins $0 \rightarrow B$, for communication between activities i and j . Thus, a_b and n_b are the average message size and number of messages in bin b , respectively.

3.3.3. Fitness for Integrating Threads (FIT). Based on the characterizations of the machine and application, a measure is needed to assess the importance of how a pair of threads should be placed relative to each other.

We calculate vector $\mathbf{T}_{ij}$, which captures the different costs incurred by a pair of threads/activities i and j , if they are placed at different levels. $\mathbf{T}_{ij}$ is of length L , where L is the number of levels in the target machine.

$$\mathbf{T}_{ij} = \{t_1, t_2, \dots, t_{L-1}\}, \quad (4)$$

where each element t_l in the vector $\mathbf{T}_{ij}$ represents the total expected communication cost that the two threads being considered would incur if placed at level l , ignoring

t_0 , which corresponds to placing i and j on the same core. We store the message costs for the bin limits when performing the machine characterization step. During the execution of an application, messages are not restricted to any size and can fall in the middle of the range in any of the bins. Hence, we perform the interpolation function to get a more accurate cost. Thus, t_l can be calculated as follows:

$$\forall t_l \in \mathbf{T}_{ij}, t_l = \sum_{b=0}^B (\text{Interpolate}(A_{ij}[b]) \times N_{ij}[b]), \quad (5)$$

where

$$\text{Interpolate}(x) = (\mathbf{C}_l[s(x)] + \mathbf{C}_l[s(x) + 1])/2, \quad (6)$$

where $s(x)$ is the index of the first element in $\mathbf{C}_l$ that represents message sizes less than or equal to x , as defined in Equation (1). Finally, we derive the **FIT** measure to assess how a level l is suitable for two threads i and j . As the placement of one pair of threads may impact the others, **FIT** is based on the total gain from placing a pair of threads at a better level, minus the total loss due to placing them at levels that are worse, as compared to the current level l :

$$\begin{aligned} \mathbf{FIT}[i, j][l] = & \\ & \sum_{w \in \text{Worse}} (\mathbf{T}_{ij}[w] - \mathbf{T}_{ij}[l]) - \sum_{b \in \text{Better}} (\mathbf{T}_{ij}[l] - \mathbf{T}_{ij}[b]) = . \end{aligned}$$

With simple rearrangement of the terms of the second summation, this becomes

$$\sum_{w \in \text{Worse}} (\mathbf{T}_{ij}[w] - \mathbf{T}_{ij}[l]) + \sum_{b \in \text{Better}} (\mathbf{T}_{ij}[b] - \mathbf{T}_{ij}[l]) = .$$

This can be further simplified into

$$\sum_{o \neq l} (\mathbf{T}_{ij}[o] - \mathbf{T}_{ij}[l]), \quad (7)$$

where $\mathbf{T}_{ij}[l]$ is the total messaging cost associated with having threads i and j placed at level l , as shown in Equation (4), and *Better* is the set of all better mappings, *Worse* is the set of all worse mappings, and o represents all other levels that belong to *Better* and *Worse* less the current level l . **FIT** measure is computed based on the benefit-loss tradeoff caught by Equation (7). A **FIT** value is calculated at each level for each pair of interacting threads. The **FIT** measure simply assesses how a given level is a good or bad choice for any two threads. The first summation calculates the gain by not being at worse placement levels, and the second summation calculates the loss by not going to better level placements. By getting the difference of the two values, we can get an idea about how good a given level is for a pair of threads. The complexity of computing the **FIT** measure for all threads at all levels is $O(b.l.n^2)$, where b is the number of bins, l is number of levels, and n is the number of threads.

A full explanation of computing the **FIT** measures between a pair of threads at the different levels is shown in Figure 8. The rounded boxes in the figure depict the different placement levels available. For example, the innermost represents a die, then a socket, and, finally, a node. The blue thread is shown at different levels in each sub-figure, corresponding to different relative placement of the two threads. The red bars on the bar graph below show the cost of the communication at each level. To calculate the **FIT** $[i, j][1]$ metric at the die level (level1), as in Figure 8(a), we start from the die level as the baseline and calculate whether mapping those threads to other levels will results in benefit or loss. Differences above the dotted line correspond to savings

Fig. 8. Calculating FIT for a pair of threads at different levels.

Fig. 9. Graph partitioning.

by placing the pair of threads at the baseline level. Differences below the dotted line correspond to losses from placing the pair of threads at the baseline level. The **FIT** metric is the sum of all the gains ($w \in Worse$ in Equation (7)) minus the sum of all the losses ($w \in Better$ in Equation (7)). This has to be repeated for the rest of the levels. The number of levels will depend on the target machine.

3.3.4. Hierarchical Partitioning of Thread Interaction Graphs. PHLAME represents the application as an interaction graph, where vertices represent threads and edges represent interactions between threads. The edges have multiple weights that represent the **FIT** measure between adjacent threads at multiple levels as in Figure 9(a). The partitioning step will hierarchically create partitions that can be mapped onto locales, as shown in Figure 9(b). There are several strategies to produce hierarchical partitions. A bottom-up strategy (Clustering) is to begin with isolated partitions and create grouping recursively until reaching one partition. A top-down strategy (Splitting) would be beginning with a single partition and start splitting it recursively until reaching partitions consisting of only one thread/activity. A preliminary implementation of these two strategies was presented in Anbar et al. [2014]:

- Clustering, Bottom-Up Strategy (Algorithm 1) consisting of three phases:
 - (1) Grouping Phase: can be performed by any known graph partitioning algorithm. The priority should be given to the pairs with highest $\mathbf{FIT}[l]$ at the current level l . This means that the algorithm will tend to place pairs that are benefiting most of being grouped at the current level together.
 - (2) Packing Phase: ensures that the mapping is done using the minimum number of resources.
 - (3) Aggregation Phase: prepares the graph for the next grouping episode at level $l+1$ by combining the $\mathbf{FIT}[l + 1]$ values of the sub-partitions of any pair of recently created partitions.
- Splitting, Top-Down Strategy (Algorithm 2) consisting of two phases:
 - (1) Splitting Phase: can be performed by any known graph partitioning algorithm. The priority should be given to the pairs with lowest $\mathbf{FIT}[l]$ at the current level l . This means that the algorithm will tend to group pairs that will suffer most if being separated in different partitions at the current level.
 - (2) Recursive Descent Phase: step down to the lower level $l - 1$ and repeat the splitting for all the produced partitions.

The Splitting algorithm comes in two variants:

 - (1) Restricted - ensures that the mapping is done using the minimum number of resources (nodes).
 - (2) Non-restricted - with no resource restrictions, allows us to investigate possible mappings when resources are available. To obtain unrestricted mapping, the constraint of creating maximum N partitions in Algorithm 2 is just removed.

3.3.5. PHlame Adaptive Selection Test (PHAST) Algorithm. It is impractical to try all partitioning strategies for all applications. To overcome this big overhead, we are proposing the PHAST algorithm, which analytically predicts the best partitioning strategy based on the predicted total communication cost. The total communication cost of a partition can be calculated as follows:

$$\text{Total Cost}(\mathbf{T}, M) = \sum_{i=0}^{X-1} \sum_{j=i}^X \mathbf{T}_{ij}[lvl(i, j, M)], \quad (8)$$

where $\mathbf{T}$ is a two-dimensional (2D) array holding all cost vectors $\mathbf{T}_{ij} \forall$ threads i, j and X is the total number of threads and the function $lvl(i, j, M)$ returns the level that separates i and j according to the given mapping M .

The PHAST algorithm neither computes new maps nor does it search the entire space for best one. The objective of PHAST is to pick, from the mappings that are generated by different scheduling algorithms, such as the ones discussed earlier, the one that will give the best performance. The inputs to PHAST are the set of mappings and the cost of the communication between thread pairs at different levels as determined in the machine characterization step. The level that separates a pair of threads is determined by a given mapping, which in turn imposes a certain communication cost for that pair. PHAST computes the total communication cost by summing all communication costs for all thread pairs for different mappings. PHAST then picks the mapping that leads to minimum total communication cost. The intuition is straightforward, and the gain from a placement will correlate to the amount of reduction in total communication cost. Thus, PHAST picks the mapping that will produce the minimum total communication cost as shown in Algorithm 3.

The complexity of the PHAST algorithm is $O(m.n^2)$, where m is the number of available mappings and n is the total number of threads when cost matrix $\mathbf{T}$ is pre-computed,

ALGORITHM 1: Hierarchical Threads Clustering**Procedure:** Cluster**Input:** Thread Interaction Graph - edges labeled by **FIT****Output:** Partitioned graph**begin**

// Each thread is considered a group at level 0

// Starting from level 1

 $l \leftarrow 1$ **repeat**

// Grouping Phase

 Group partitions from level l giving priority to pairs with highest **FIT**[l]
 to be in the same partition

// Packing Phase

while # of formed partitions > max # of locales **do** Break weakest partition // with smallest total internal **FIT**[l]

Distribute it on other partitions

end while

// Aggregation Phase

for each pair of newly formed partitions p_x, p_y **do** **FIT**[p_x, p_y][$l + 1$] = 0 **for each** sub-partition $i \in p_x$ **do** **for each** sub-partition $j \in p_y$ **do** **FIT**[p_x, p_y][$l + 1$] += **FIT**[i, j][$l + 1$] **end for** **end for** **end for**

// Step one level up

 $l \leftarrow l + 1$ **until** $l == L - 1$ **return** Partitioned Graph**end**

or $O(b.m.n^2)$, where again m is the number of available mappings, n is the total number of threads, and b is the total number of bins when cost matrix **T** is computed on the fly.

4. EXPERIMENTAL RESULTS**4.1. Experimental Testbed**

To perform the experiments, we used George, a Cray XE6m supercomputer. George is composed of 56 compute nodes, and each node is composed of 32GB of RAM and 2 12-core AMD Magny Cours chips. Each core is an out-of-order, three-way superscalar processor. Each core has separate private instruction and data L1 caches each is 64KB and a private 512KB L2 cache. The L1 and L2 caches use an exclusive layout—that is, the L2 cache is a victim cache for the L1 instruction and data caches. Each processor consists of two dies (6 cores each), where the cores on each of these dies have a shared 6MB L3 cache. Every two nodes in the system share a Gemini network chip. The network chips are interconnected using a 2D torus topology network. The system has a peak performance of 10.6TFlops.

In the sense of PHLAME, a level is identified based on sharing some interaction medium. This interaction medium can be either a shared network router (such as the

ALGORITHM 2: Hierarchical Threads Splitting

Procedure: Split**Input:** Thread Interaction Graph - edges labeled by **FIT**, Level l **Output:** Partitioned graph**begin** // L is the number of levels // Initial call is done with a single partition at level $L - 1$ **if** $l == 0$ **then** // At l_0 , a graph will contain only one vertex **return** Graph **end if**

// Splitting Phase

 $N \leftarrow$ Locale Size at level l Split input into N partitions giving priority to pairs of *lowest* **FIT**[l] to be assigned the same sub-partition

// Recursive Descent Phase

for each newly formed sub-partition p **do** // Use **FIT**[$l - 1$] for the next level of partitioning Split($p, l - 1$) **end for** **return** Partitioned Graph**end**

ALGORITHM 3: PHlame Adaptive Selection Test (PHAST)

Procedure: PHAST**Input:** Thread Costs Matrix **T**, Set of Mappings $Maps$ **Output:** Predicted Best Mapping**begin** $M = \underset{m \in Maps}{argmin}(TotalCost(\mathbf{T}, m))$ **return** M **end**

Gemini chip in our testbed) or a shared memory. A level under PHLAME is therefore recursively defined by having a shared medium of interaction to the units belonging to a lower level. In that sense, our characterization is targeting only levels with no distinction between memory and network, as long as they provide a shared medium for interaction. The difference between a memory and a network router is caught in the characterization as message cost, thereby aiding effective placement of the threads. If memory or networking behavior, from one system to another, differs, then those differences will be captured in the characterization of that target machine and applied accordingly.

We identified the following locality levels in George:

—Four levels belong to the memory hierarchy:

(1) *Core level* ($level_0$): A total of 1,344 cores are available.

(2) *Die level* ($level_1$): Each die is composed of 6 cores, sharing an L3 cache and a memory controller.

- (3) *Socket level (level₂)*: Each socket is composed of two dies, the dies are interconnected with HyperTransport links.
- (4) *Node level (level₃)*: Each node is composed of two sockets for a total of 24 cores.
- Two levels belong to the interconnection network:
- (5) *Network chip level (level₄)*: Each two nodes are sharing a Gemini network chip.
- (6) *System level (level₅)*: The interconnect for the entire system.

We adopted a variant of the crossfire bandwidth (Xfire) benchmark [Conway et al. 2010]. Recall that our system characterization is focused on level-to-level communication for a given pair of threads using different message sizes. In order to assess the bandwidth at a given level, we measure the bandwidth available to each core when all cores are accessing the memory of other cores at that level. For example, at the socket level, to measure the bandwidth between 12-core sockets on the same node, we run 24 threads, each is pinned to one core. Each thread then accesses the data that belong to the threads pinned to the other socket. At the node level, the same is done where cores from one node access the memory of cores from other nodes at the same level. This is also repeated at the chassis level, and so on. In each of the cases, different data sizes are used and all cores are participating while bandwidth measurements are taken.

4.2. Experimental Setup

To implement and experimentally verify the concepts presented in this article, we used MPI and UPC as it is the most mature PGAS language today. In the “Grouping Phase” of the clustering algorithm and “Splitting Phase” of the splitting algorithm, we relied on METIS [Karypis and Kumar 1998] v5.1.0 to produce the graph partitions. The mappings generated by our algorithms resulted in different numbers of threads assigned to each node. This is not generally allowed in this environment. To be able to launch variable numbers of threads per node and at the same time to control the mapping of the threads, we used the most recent Berkeley UPC compiler v2.20.0 and we modified the GASNet [Bonachea 2002] Gemini conduit, as described in Anbar et al. [2014].

For the UPC application characterization, we modified the Parallel Performance Wizard v3.2 (PPW) [Su et al. 2008]. PPW is a performance analysis tool designed for UPC, MPI (C Bindings Only), and SHMEM programs. To be able to characterize Fortran MPI applications, we modified the Tuning and Analysis Utilities (TAU) v2.24 [Shende and Malony 2006]. We customized PPW and TAU tools to be able to generate communication profiles with fine bin resolution. This entailed making changes to PPW and TAU front and back ends.

4.3. Benchmarks

As presented in Section 1, we first used a synthetic benchmark to evaluate the possible gains of PHLAME (Figure 3(a) and (b)). Therefore, we are expecting the gains to grow with the ratio of remote communication and the number of threads.

For real-world workloads, we used all five NAS Parallel Benchmark kernels (Class C), written in UPC [Bailey et al. 1991; El-Ghazawi and Cantonnet 2002], and the corresponding MPI versions. They represent various computation workloads with a variety of access and communication patterns, as follows:

- NAS CG - Conjugate Gradient: irregular memory access and communication
- NAS EP - Embarrassingly Parallel: minimal communication
- NAS FT - Discrete 3D Fast Fourier Transform: large, all-to-all communication
- NAS IS - Integer Sort: performs many random accesses
- NAS MG - Multi-Grid: both short- and long-range communication

For completeness, we also included heat diffusion as the sixth benchmark. This benchmark performs a six-point stencil operation in 3D across a large 3D matrix. The benchmark is described by El-Ghazawi et al. [2005].

4.4. Results and Discussion

For each benchmark, we present the results in eight graphs. The UPC-related graphs are on the left, and the MPI-related graphs are on the right. First, the Communication Matrices are shown in parts (a) and (b) of each figure. We obtain it by profiling the application. The communication matrix represents the weighted sum of all the bins used by PHLAME.

Second, the Performance Gains are shown in part (c) and (d) of each figure while scaling the number of threads from 64 to 1024. The baseline timing is generated by the default partitioning of the underlying Cray testbed, which is representative of the current state-of-the-art driven by common practice, rather than hierarchical locality exploitation. Currently, there is no automated system that provides such a hierarchical locality inspired partitioning, which is introduced in this article. The Cray testbed has three algorithms controlled by the `MPICH_RANK_REORDER_METHOD` environment variable. The default algorithm that is used by both UPC and MPI runtimes sequentially places threads according to their rank, with disregard to locality, assigning all cores in a node before using another. The other two algorithms are round-robin and folded-rank. We have used the default algorithm, which gave the best performance among the available options, as the baseline. The graphs have four bars representing different placements for each thread count. The y -axis represents the gain as calculated by Equation (4) over the baseline. The performance gain is calculated using following equation:

$$\text{Performance Gain} = \frac{T_{\text{Default}} - T_{\text{PHLAME}}}{T_{\text{Default}}} \%, \quad (9)$$

where T_{Default} (our baseline) and T_{PHLAME} are the real runtimes of the actual programs on the target machine for the default mapping and one of the PHLAME mappings, respectively. T_{Default} is the default execution time on the Cray XE6m testbed (George) without any improvements. T_{PHLAME} is the execution time on the same testbed with PHLAME incorporated.

The Relative Communication Costs are shown in parts (e) and (f) of each figure, as calculated in the PHAST algorithm. This graph estimates the reduction in communication cost relative to the default placement. It is used by PHAST to select the best partitioning strategy. Again, the graphs have four bars representing different placements for each thread count. The y -axis represents the ratio of total communication costs of each placement to that of the baseline (default placement).

For the EP kernel, the graphs are not shown as EP is designed to have minimal communication and, as expected, no significant gain is obtained by PHLAME.

For FT (Figure 10), the large all-to-all communications are difficult to optimize but the performance gain still ranges from 6% to 10% for the UPC case and 2% to 7% in the MPI case. The communication matrices for both UPC and MPI show that the differences in messages between different pairs of threads are small, resulting in almost a solid red color for both versions. Consequently, the reduction in total communication cost is quite small. The UPC achieves higher gain as the baseline performance of MPI is better than that of the UPC.

For MG (Figure 11), PHLAME is able to generate a good hierarchical placement that improves the performance by up to 60% in the case of UPC and 35% in the case of MPI. The communication matrices for both UPC and MPI show more locality than the matrices of the FT. The communication matrix of the UPC version shows more

Fig. 10. NAS Parallel Benchmark: FT kernel.

opportunities for locality exploitation. Accordingly, the reduction in total communication cost in the UPC case are more than in the MPI version, which in turn leads to more performance gains in UPC than MPI. Like the case of FT, PHAST adaptively selects different algorithms for different number of cores.

For IS (Figure 12), although the communication pattern appears to be random, using PHLAME can increase the performance by up to 12% in the UPC case and 9% in the MPI case when using 1024 threads. The reduction in total communication cost is not much in both cases due to the poor degree of locality. The UPC and MPI baseline version performances are close, with that of MPI slightly better.

Fig. 11. NAS Parallel Benchmark: MG kernel.

For heat diffusion (Figure 13), the gains increase from 7% to 22% in the UPC case as the number of threads is increased. The MPI gains range from 4% to 12%. The communication matrix of the UPC shows a high degree of locality, leading to better performance gains. However, the communication matrix of MPI is showing a very poor degree of locality. The reason for having different communication patterns is that in the MPI version every thread does a max-reduce operation after each step to find the maximum change from the previous step and stops if it reaches some threshold. This leads to an all-to-all communication.

Fig. 12. NAS Parallel Benchmark: IS kernel.

Figure 14 presents the results obtained with the NAS CG Benchmark. The performance gains range from 3% to 88% for the UPC version and range from 3% to 40% for the MPI version. The communication matrices of both versions shows high degrees of locality that clearly appear as clusters of communicating threads around the diagonal. The communication within the groups in the UPC version is more dense than the MPI version, which translates to higher degrees of locality in case of UPC.

The significantly larger gains obtained with the non-restricted splitting algorithm are attributed to the particular communication pattern of CG. In Figure 14(a), we see that clusters or groups of 16 threads communicate intensively with each other. This tight intragroup interaction is due to the high frequency of collective all-reduce

Fig. 13. Heat diffusion benchmark.

operations within each group. The default and the two restricted mappings cause some of these groups to be split and distributed across different nodes, as shown in Figure 15. Thus, all the extensive communication within a group now goes across the network, leading to high network overhead and, in turn, performance degradation, limiting the amount of improvement. The non-restricted splitting algorithm does not break those groups by using more nodes (see Table I).

The less dense communication within the groups of the MPI version does not lead to congested networks as in the case of UPC. Accordingly, the MPI version does not have the scalability problem of the UPC version. However, using the non-restricted version and keeping the communication within the groups off the network still helped in achieving high performance gain.

Fig. 14. NAS Parallel Benchmark: CG kernel.

Even though more nodes are used, the performance gain is so important that the system usage is greatly reduced (see Figure 16). With 1024 threads, it is not possible to preserve these clusters of 16 threads anymore as it would require 64 compute nodes and we only have 56 in our machine (George); this is why the performance gain obtained at 1024 threads is *only* 30%.

Across all the benchmarks, it is important to note that PHAST achieves 100% accuracy in choosing the best partitioning algorithm always based on the computed relative communication cost.

We also demonstrate that through all the benchmarks (with the only exception of CG at 1024 threads due to the insufficient number of nodes in George), the performance

Fig. 15. Inefficient placement of UPC threads for the CG benchmark.

Table I. Number of Nodes Allocated for Each Mapping Strategy by PHLAME for the CG Kernel

# of threads	Default Run	Restricted Run	Non-restricted Run
64	3	3	3
128	6	6	8
256	11	11	16
512	22	22	32
1024	43	43	56*

*Would have been 64 if a greater number of nodes was available.

(a) UPC Version

(b) MPI Version

Fig. 16. The required “node seconds” needed for CG by different partitioning strategies.

gains increase with the number of cores. This clearly shows that exploiting hierarchical locality becomes more and more important as systems grow, which is critical for extreme scale computing.

5. CONCLUSION AND FUTURE WORK

Large parallel systems exhibit deeper hierarchies as the technology advances. This introduces different levels with different access latencies and bandwidths across the system, making data locality critical for performance and very complex to handle by the programmer directly through hierarchical programming models. In this work, we present PHLAME, an execution model whose runtime system is designed to exploit the machine’s underlying hierarchy by providing a better mapping between activities and processing elements.

Under PHLAME, the programmer is only responsible for the first level of locality, which is afforded by existing locality-aware programming models. A PGAS programming model would be an excellent fit for our approach. The higher levels of locality are handled by the runtime system without hindering the programmer's productivity.

In this article, we presented PHLAME, along with extensive benchmarking and experimentation of the underlying concepts and techniques using a variety of workloads that included all of the UPC NAS Parallel Benchmark kernels and the corresponding MPI versions. We also included a 3D heat diffusion benchmark in both UPC and MPI. Our experiments have shown up to 88% improvement in performance gain. We further show that PHLAME can reduce the overall communication cost up to 15% compared to the default run(s). Even when the communication pattern is uniformly distributed across threads, as in NAS Parallel Benchmark kernel IS, PHLAME showed a performance gain of more than 12%. The performance gains are obtained automatically by the runtime system without any user intervention.

We also propose PHAST, which automatically selects the best hierarchical mapping available. In all of our experiments, PHAST provided 100% accuracy in predicting the mapping with the best performance.

The performance gains obtained by exploiting hierarchical locality increased with the number of cores in all of our experiments. This is why we believe that the PHLAME approach will be essential for extreme scale computing systems with deep hierarchies, as will be the case for Petascale systems and beyond.

Future work will include modeling that can enable studying extreme what-if scenarios and investigating the exploitation of hierarchical locality under dynamic parallelism.

REFERENCES

- Ahmad Anbar, Abdel-Hameed Badawy, Olivier Serres, and Tarek El-Ghazawi. 2014. Where should the threads go? Leveraging hierarchical data locality to solve the thread affinity dilemma. In *Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS'14)*. IEEE. <http://hpcl.seas.gwu.edu/eprints/38/>.
- Ahmad Anbar, Olivier Serres, Engin Kayraklioglu, Abdel-Hameed Badawy, and Tarek El-Ghazawi. 2015. PHLAME: Hierarchical locality exploitation using the PGAS model. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS'15)*. IEEE Computer Society, Washington, DC, 82–89. DOI : <http://dx.doi.org/10.1109/PGAS.2015.16>
- D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. ACM, New York, NY, 158–165. DOI : <http://dx.doi.org/10.1145/125826.125925>
- Dan Bonachea. 2002. *GASNet Specification, V1.1*. Technical Report. Berkeley, CA, USA.
- Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. Hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10)*. IEEE Computer Society, Washington, DC, 180–186. DOI : <http://dx.doi.org/10.1109/PDP.2010.67>
- Jehoshua Bruck, Ching-Tien Ho, Eli Upfal, Shlomo Kipnis, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst.* 8, 11 (Nov. 1997), 1143–1156. DOI : <http://dx.doi.org/10.1109/71.642949>
- B. L. Chamberlain, D. Callahan, and H. P. Zima. 2007. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. DOI : <http://dx.doi.org/10.1177/1094342007078442>
- P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. 2010. Cache hierarchy and memory subsystem of the AMD opteron processor. *IEEE Micro* 30, 2 (March 2010), 16–29. DOI : <http://dx.doi.org/10.1109/MM.2010.31>

- E. H. M. Cruz, M. Diener, and P. O. A. Navaux. 2012. Using the translation lookaside buffer to map threads in parallel applications based on shared memory. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*. 532–543. DOI: <http://dx.doi.org/10.1109/IPDPS.2012.56>
- Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. DOI: <http://dx.doi.org/10.1109/99.660313>
- Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. 2002. Zoltan data management services for parallel dynamic applications. *Comput. Sci. Eng.* 4, 2 (2002), 90–97.
- Tarek El-Ghazawi and François Cantonnnet. 2002. UPC performance and potential: A NPB experimental study. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, Los Alamitos, CA, 1–26.
- Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2003. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, New York, NY.
- Tarek El-Ghazawi, W. Carlson, T. Sterling, and Katherine Yelick. 2005. *UPC: Distributed Shared Memory Programming*.
- Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhajan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.
- Habanero-C. 2015. Homepage. Retrieved from <https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C>.
- Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J. Wright, and Laxmikant V. Kale. 2014. Maximizing throughput on a dragonfly network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Piscataway, NJ, 336–347. DOI: <http://dx.doi.org/10.1109/SC.2014.33>
- E. Jeannot, G. Mercier, and F. Tessier. 2014. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Trans. Parallel Distrib. Syst.* 25, 4 (April 2014), 993–1002. DOI: <http://dx.doi.org/10.1109/TPDS.2013.104>
- George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20, 1 (Dec. 1998), 359–392. DOI: <http://dx.doi.org/10.1137/S1064827595287997>
- R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, and others. 2014. Top ten exascale research challenges. *DOE ASCAC Subcommittee Report* (February 2014).
- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (Feb. 2002), 50–58. DOI: <http://dx.doi.org/10.1109/2.982916>
- Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. 2014. Compiler-driven data layout transformation for heterogeneous platforms. In *Euro-Par 2013: Parallel Processing Workshops*. Springer, Berlin, 188–197.
- E. H. Molina da Cruz, M. A. Zanata Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J. Mehaut. 2011. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. 551–558. DOI: <http://dx.doi.org/10.1109/IPDPS.2011.197>
- PGAS. 2015. Partitioned Global Address Space Languages. (2015). <http://pgas.org> Accessed: 7/2015.
- Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. 2011. OpenSHMEM - Toward a unified RMA model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, Berlin, 1379–1391. DOI: http://dx.doi.org/10.1007/978-0-387-09766-4_490
- Bogdan Prisacari, German Rodriguez, Cyriel Minkenberg, and Torsten Hoeftler. 2013. Bandwidth-optimal all-to-all exchanges in fat tree networks. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, NY 139–148. DOI: <http://dx.doi.org/10.1145/2464996.2465434>
- Paul Sack and William Gropp. 2012. Faster topology-aware collective algorithms through non-minimal communication. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*. ACM, New York, NY 45–54. DOI: <http://dx.doi.org/10.1145/2145816.2145823>
- Sameer S. Shende and Allen D. Malony. 2006. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.* 20, 2 (May 2006), 287–311. DOI: <http://dx.doi.org/10.1177/1094342006064482>
- Hung-Hsun Su, M. Billingsley, and A. D. George. 2008. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In *Proceedings of the IEEE International*

- Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.* 1–8. DOI: <http://dx.doi.org/10.1109/IPDPS.2008.4536476>
- H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. Mclay, K. Schulz, and D. K. Panda. 2011. Design and evaluation of network topology-/speed- aware broadcast algorithms for InfiniBand clusters. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER'11)*. IEEE Computer Society, Washington, DC, 317–325. DOI: <http://dx.doi.org/10.1109/CLUSTER.2011.43>
- Jingjin Wu, Zhiling Lan, Xuanxing Xiong, Nickolay Y. Gnedin, and Andrey V. Kravtsov. 2012. Hierarchical task mapping of cell-based AMR cosmology simulations. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society, Washington, DC, 1–10. DOI: <http://dx.doi.org/10.1109/SC.2012.63>
- Katherine Yelick, Vivek Sarkar, John Mellor-Crummey, James Demmel, Krste Asanovi, Armando Fox, Mattan Erez, Dan Quinlan, Surendra Byna, Marc Day, Tony Drummond, Paul Hargrove, Steven Hofmeyr, Costin Iancu, Khaled Ibrahim, Frank Mueller, Leonid Oliker, Eric Roman, John Shalf, David Skinner, Erich Strohmaier, Brian Van Straalen, Samuel Williams, and Yili Zheng. 2015. DEGAS: Dynamic Exascale Global Address Space; slides available online, retrieved 5/2015, <http://goo.gl/IrfsIs>.
- Wei Zheng, Lu Tang, and R. Sakellariou. 2015. A priority-based scheduling heuristic to maximize parallelism of ready tasks for DAG applications. In *Proceedings of the 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 596–605. DOI: <http://dx.doi.org/10.1109/CCGrid.2015.97>

Received September 2015; revised January 2016; accepted February 2016