Effective use of the PGAS Paradigm: Driving Transformations and Self-Adaptive Behavior in DASH-Applications

Kamran Idrees
High Performance Computing Center Stuttgart (HLRS)
idrees@hlrs.de

Tobias Fuchs
Ludwig-Maximilians-Universität München (LMU)
tobias.fuchs@nm.ifi.lmu.de

Colin W. Glass
High Performance Computing Center Stuttgart (HLRS)
glass@hlrs.de

Abstract

DASH is a library of distributed data structures and algorithms designed for applications running on modern HPC architectures, composed of hierarchical network interconnections and stratified memory. DASH implements a PGAS (partitioned global address space) model in the form of C++ templates, built on top of DART – a run-time system with an abstracted tier above existing one-sided communication libraries.

In order to assist in application development exploiting the hierarchical organization of HPC machines. DART allows for reordering the computational units and in this paper we present an automatic, hierarchical units mapping technique (using a similar approach to the Hilbert curve transformation in order to reorder DART units) on the Cray XC40 machine Hazel Hen at HLRS. To evaluate the performance of new units mapping which takes into the account the topology of allocated compute nodes, we perform latency benchmark for a 3D stencil code. The technique of units mapping is generic and can be be adopted in other DART communication substrates and on other hardware platforms. Furthermore, high–level features of DASH are presented, enabling more complex automatic transformations and optimizations in the future.

Categories and Subject Descriptors D.1.2 [*PROGRAMMING TECHNIQUES*]: Automatic Programming; D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming

Keywords DASH, DART, PGAS, UPC, MPI, Self-Adaptation, Code Transformations

1. Introduction

Partitioned Global Address Space (PGAS) devises a new method of parallel programming by introducing a unified global view of address space (like purely shared memory systems) and presiding the distribution of data (similar to a distributed memory system), in order to provide a programmer with ease of use and a locality-aware paradigm. To provide such an abstracted view of shared memory programming over distributed memory systems, PGAS implementations use one–sided communication substrate, which is hidden from the application developer.

Unified Parallel C (UPC) is an implementation of the PGAS model. UPC is comprised of a single shared address space, which is partitioned among UPC threads, such that a portion of shared address space resides in the memory local to the UPC thread. UPC provides mechanisms to distinguish between local and remote data accesses, thus allowing to capitalize on data locality. However, a programmer needs to perform custom coding to exploit the locality efficiently and build advance data distribution schemes on top of very basic data distribution schemes provided by UPC.

DASH is a C++ library that delivers distributed data structures and algorithms designed for modern HPC machines, which are well-suited for hierarchical network interconnections and memory stratum (Fürlinger et al. 2014). DASH aims at various domains of scientific applications, providing the programmer with advanced data structures and algorithms, consequently reducing the need for custom coding.

DASH calls DASH Run-Time (DART), which provides basic functionalities to build a PGAS model using state of the art one-sided communication libraries (Zhou et al. 2014). These functionalities include:

- Global memory management and optimization for accessing data that reside on shared memory system (Zhou et al. 2015)
- Creation, destruction and management of teams and groups
- Collective and non-collective communication routines
- Synchronization primitives

This paper presents DASH as an alternative to traditional PGAS implementations like UPC. Short-comings of UPC and other traditional PGAS implementations are discussed, which in many cases prevent an effective use of the PGAS paradigm. Furthermore, features of DASH overcoming these short–comings are presented. The main contributions of this paper are:

1. We present advanced local copy feature in DASH
2. We present automatic hierarchical units mapping mechanism in DART for applications having nearest neighbor communication pattern
3. We evaluate the applicability of automatic hierarchical units mapping mechanism on a 3D stencil communication kernel on Cray XC40 Hazel Hen machine at HLRS
4. We evaluate the throughput of the local copy feature

2. Experiences with UPC

From our experience with UPC for our in–house molecular dynamics code, we highlighted three major issues, resulting in severe performance degradation (Idrees et al. 2013). These issues – and a further problem regarding hardware topology – are:

1. Requires manual pointer optimization for fast access to local data (using local pointer)
2. Non-trivial data distribution schemes need to be implemented by hand
3. Communication is performed at the same granularity as data access
4. No mechanism available for colocating strongly interacting units on the given hierarchical hardware topology

The first issue regards the failure of UPC compilers to automatically distinguish between shared and distributed memory data accesses, even though the complete data layout is available. This holds true both for static or dynamic allocation of data (as the block size of a distributed shared array needs to be a compile–time constant). This can lead to a significant performance degradation and can only be avoided by expert programmer intervention. Manual optimization requires checking all parts of the code where significant data accesses are performed and switching to local pointers for local memory access.

The second issue is that UPC provides round robin and blocked data distribution schemes. These schemes are suboptimal for many applications featuring some soft of short range geometric data accessing patterns, e.g. stencil patterns. This will lead to a unnecessarily high amount of communication traffic and percentage of remote communication. To avoid this problem, the programmer has to write specific data mapping routines.

The third issue is associated with the communication granularity. In shared memory address space, the programmer can directly access and modify the shared data. The PGAS paradigm also provides these attributes for its global address space. However, accessing and modifying remote data is expensive, especially if it leads to many small communications. As UPC does not change the granularity, this often leads to a vast number of tiny communications. To avoid this problem, the programmer needs to take the underlying distributed memory architecture into account and perform the necessary optimizations for packing communications manually.

The fourth issue addresses the difficulty of adapting the behavior of an application to the machine topology. For example, reordering the placement of software units that may allow to reduce the communication cost by placing the interacting partners closer to each other on the physical hardware.

To summarize: in order to achieve a near optimal performance using traditional PGAS implementations, the programmer needs to take care of a variety of issues manually. This contradicts the driving idea behind PGAS: ease of programmability. The good news is, DASH is tackling these issues by providing automatic optimization for faster local data accesses (Zhou et al. 2015), advanced data distribution schemes (Fuchs et al. 2015), algorithm specific routines for pre-fetching and packing of data (evaluated in section 4.3) and automatic hierarchical units mapping (evaluated in section 4.3). The following section provides a detailed illustration of these advance features.

3. DASH as a Solution

DASH resolves the short–comings of the traditional PGAS implementations with its automatic optimizations, advanced data structures and algorithms. We will now explain few specific features of the DASH which address the problems highlighted in the previous section.

3.1 Fast Access to Local Data

The automatic detection of local vs. remote data access in DASH demonstrates an effective use of PGAS paradigm: every data access is performed in the most efficient way available. This automatic behavior is achieved by capitalizing on the shared memory window feature of MPI-3 (Hoefler et al. 2012) used in the MPI version of DART (DART–MPI). The shared memory window can be accessed directly by local MPI processes using load/store operations (zero–copy model), allowing the processes to circumvent the single-copy model of the MPI layer. DART-MPI maps both global and shared memory windows to the same shared memory region, thus allowing the DART units on shared memory to directly access the local memory region. The DART units that are not part of the shared memory window, perform RMA operations using a global window. Furthermore, the use of the zero copy model for intra-node communication in DART scales down the memory bandwidth problem. We have demonstrated in (Zhou et al. 2015), our optimization of mapping both shared and global memory windows to the same memory region on shared memory, enabling faster intra-node communication. This allows DASH programmers to iterate over distributed data structures without worrying about slow local data accesses, unlike UPC where man-

ual pointer optimizations are necessary for such perfomance (Idrees et al. 2013).

3.2 High-Level Data Distribution Schemes

As previously mentioned, DASH features several data distribution schemes (*pattern types*), each highly flexible in configuration. However, as no single distribution type could possibly be configured to be optimal for any thinkable communication strategy, the preferable pattern type depends on the specific use case. Algorithm variants strictly depend on domain decomposition as they are optimized for, or even require, data distributions that satisfy specific properties.

We therefore provide high-level functions to automatically optimize data distribution for a given algorithm and vice-versa. These mechanism are described in detail in (Fuchs et al. 2015). In this we present a classification of data distribution schemes based on well-defined properties in the three mapping stages of domain decomposition: partitioning, mapping, and memory layout. This classification system, briefly put, serves two purposes:

First, it allows to formally describe general data distributions by their semantic properties, also named *pattern traits* in DASH. As an example, the *balanced partitioning* property describes that data is partitioned into blocks of identical size.

Furthermore, pattern traits are used to specify constraints on expected data distribution semantics. For example, if an algorithm is optimized for containers that are evenly distributed among units, it could declare the *balanced partitioning* and *balanced mapping* properties as constraints, the latter describing that the same number of blocks is mapped to every unit. When applying the algorithm on a container, its distribution is then checked against these constraints already at compile time, preventing inefficient usage.

Finally, pattern traits also allow to implement high-level functions that resolve data distribution automatically. For this, we use simple constrainted optimization based on type traits in C++11 to create an instance of a initially unspecified pattern type that is optimized for a set of property constraints. To be more specific, the automatic resolution of a data distribution involves two steps: at compile time, the pattern type is deduced from constraints that are declared as type traits. Then, an optimal instantiation of this pattern type is resolved from distribution constraints and run-time parameters such as data extents and team size.

Deduction of data distribution also can interact with team specification to find a suitable logical Cartesian arrangement of units. As for domain decomposition, DASH provides traits to specify preferences for logical team topology such as ”compact” or ”node-balanced”. As a result, application developers only need to state a use case, such as DGEMM, and let unit arrangement and data distribution be resolved automatically.

Using deduction instead of specifying explicit types decouples distribution semantics from concrete implementations. This in effect increases the robustness of application code against changes between DASH versions, but more important allows to benefit from future improvements in DASH effortlessly, without modifying application code. When switching to a newer version of DASH where an improved pattern implementation is introduced, the new pattern is employed automatically if it satisfies the data distribution constraints specified in the application.

We are currently working on extending these methods to data flow scenarios, however automatic optimization in many data flow use cases is conceptually equivalent to integer programming and thus proven to be NP-hard. We assume that solutions for a useful subset of scenarios can be found using linear programming techniques like the Simplex algorithm.

While no automation can possibly do away with the need for manual optimization in general, automatic deduction as provided by DASH greatly simplifies finding a configuration that is suitable as a starting point for performance tuning. In comparison, finding practicable blocking factors and process grid extents for ScaLAPACK – even in seemingly trivial use cases – is a challenging task for non-experts and evaluating achieved performance is strongly advised from the beginning.

3.3 Creating Local Copies

TODO: Explanation regarding packing the communication is missing.

Local Copy – Objectives:

- Provide asynchronous variants of data movement operations to enable overlap of communication and computation.
- Provide semantics that are familiar from the STL, i.e. based on iterator ranges.
- Reduce calls to the communication backend to the unavoidable minimum, use shared windows or std::copy for data ranges that are located on the same node.
- Optimize for transmission buffer sizes by splitting the data movement into chunks. Adequate chunk sizes can be obtained from auto tuning or environment variables.
- Exploit parallel transmission capacity where possible: If the data source range spans the address space of multiple units, use separate asynchronous transmissions for every unit instead of consecutive blocking transmissions.

The Utility Function dash::local_range

- Partitions a global iterator range into local and remote subranges.
- Iterator range may be multidimensional.
- Local subranges are copied with std::copy.

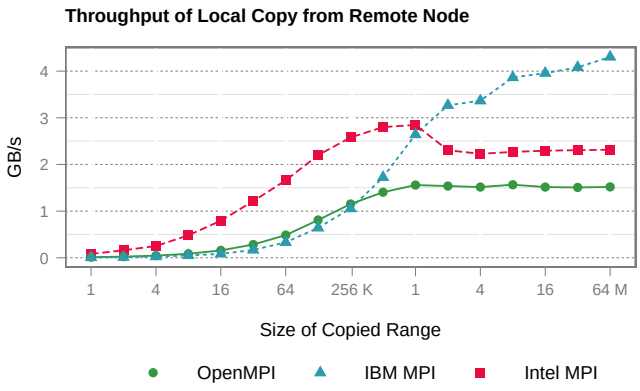

Figure 2: Throughput of `dash::copy` for local copies of blocks on a remote processing node.

- When requesting data from non-local subranges mapped to units on the same node, the DASH runtime automatically resorts to shared window queries and memcpy instead of MPI communication primitives for data movement.
- Automatically distinguishes between shared and distributed memory data accesses either already in DASH, or in the DART backend.

Use Cases

```
dash::Matrix<2, double> matrix;
// First block in matrix assigned to
// active unit:
auto l_block = matrix.local.block(0);
// Map local block to global index domain,
// shift by -1 in column dimension (west
// neighbor):
auto block_w = l_block.global.shift<1>(-1);
// Copy block to second block assigned to
// active unit:
dash::copy(block_w.begin(),
           block_w.end(),
           matrix.local.block(1).begin());
// Specify matrix block by process grid
// coordinates:
auto m_block = matrix.block({ 3, 5 });
// Copy matrix block to local array:
double * l_submat = new double[m_block.size()];
dash::copy(m_block.begin(),
           m_block.end(),
           l_submat);
```

3.4 Automatic Hierarchical Units Mapping

The mapping of an application's software units (or threads/processes/tasks) to the physical cores on an HPC machine is becoming increasingly important due to the rapid increase in the number of cores on a machine. This also leads to increasingly hierarchical networks and – depending on the underlying system and the submitted job – sparse core allocations. Therefore, if two units of an application which are interacting partners (or communicating more frequently than

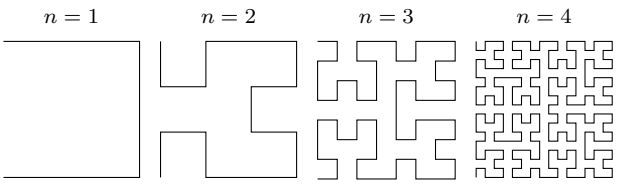

Figure 3: Hilbert Space Filling Curve Example

average) are placed far from each other in the network, they will have to communicate over several levels in the network hierarchy. The placement of these units not only has repercussions in their communication latency and bandwidth, but may also result in the congestion of the network links.

As the units mapping plays a vital role, DART-MPI provides DASH with the mean to automatically reorder the units which respect both to the communication pattern of an application and the topology of allocated nodes on an HPC machine. This results in reduced communication and overall execution time of an application. Currently we counter this problem for a specific set of applications which are based on nearest neighbor communication. The programmer only needs to inform DASH that the application to be executed has a nearest neighbor communication pattern. The automatic mapping routine then gathers the required hardware topology, computes a new hierarchical unit mapping and registers it in the system. The new mapping is determined based on an approach similar to a Hilbert Space Filling Curve (HSFC) partitioning (Moon et al. 2001). An example of HSFC is shown in figure 3. HSFC is chosen due to its property of preserving the locality. The algorithm however does not fix the order of HSFC for multiple levels as the order can be varied depending upon the number of nodes corresponding to each level in the network hierarchy.

Before discussing the steps in the automatic hierarchical unit mapping algorithm, we briefly explain network hierarchical levels of the Cray XC40 Supercomputer Hazel Hen in the following.

3.4.1 Off-Node Network Hierarchy on Hazel Hen

The first off-node network hierarchical level is a *compute blade*. A compute blade is composed of four nodes which share an Aries chip. The Aries chip connects these four nodes with the network interconnect. This is the fastest connection between nodes.

The second level is the *rank 1 network* (or backplane). The rank 1 network is used for inter-compute blades communication within a chassis – set of 16 compute blades (64 nodes). The rank 1 network has adaptive routing per packet. Any two nodes at this level communicate with each other by going through their Aries chip, through the backplane, to the Aries chip of the target node. This is an all-to-all PC board network.

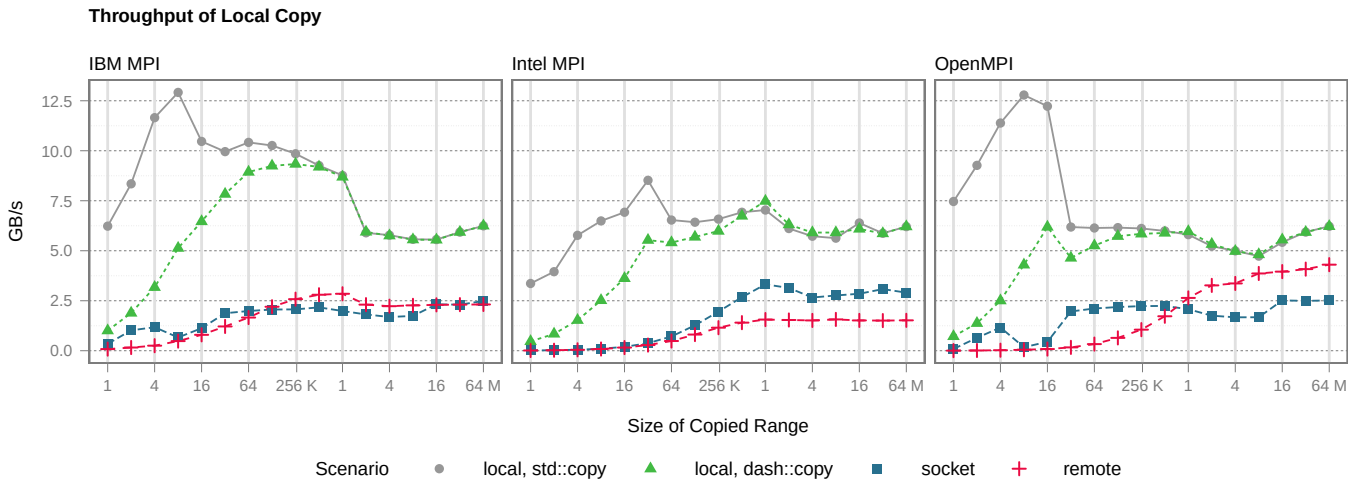

Figure 1: Throughput of `dash::copy` for local copies of blocks within a single processing node. Throughput of `std::copy` and remote copy scenario included as reference.

The *rank 2 network* is used for inter-backplane communication (nodes on distinct chassis) in a two cabinet group (a group is composed of 384 nodes). The backplanes are connected through copper cables. All copper and backplane signals run at 14 Gbps. The minimal route between two nodes on distinct chassis is two hops, whereas the longest route requires four hops. The aries adaptive routing algorithm is used to select the best route from four routes in a routing table. The rank 2 network also has an all-to-all connection, connecting 6 Chassis (two cabinets).

The last off-node network level is the *rank 3 network*, which is used for communication between different groups. The rank 3 network has all-to-all routing using optical cables. If minimal path between two groups is congested, traffic can be hopped through any other intermediate group (1 or 2 hops).

The layered layout of network hierarchy of Hazel Hen is show in figure 4.

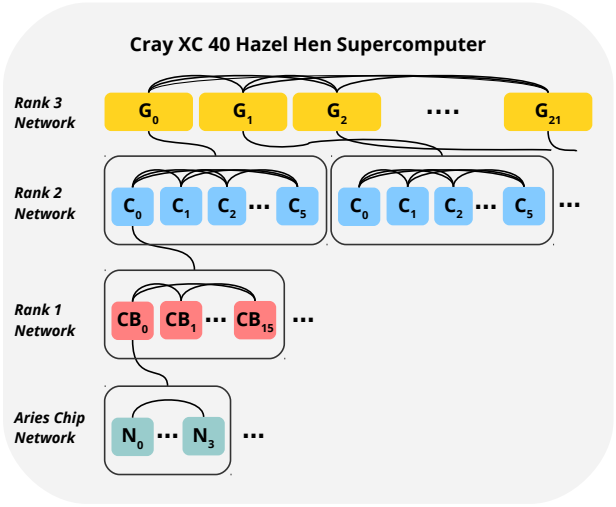

Figure 4: Layered Layout of the Network Hierarchy of `Hazel Hen`. G, C, CB and N are abbreviated for Group, Chassis, Compute Blade and Node respectively.

3.4.2 Algorithm

The automatic hierarchical units mapping algorithm is executed by every DASH unit and assumes that the user has performed the binding of the units to the CPU cores [1], such that the units do not migrate from one CPU to another. Every unit performs the following steps:

1. Acquires the *total number of units* in the team (to be mapped on the hierarchical topology).
2. Acquires its processor name and parses it to obtain the *Node ID* on which the unit resides.
3. Participates in the collective allgather operation to obtain the Node IDs of all units (units on the same node have the same Node ID) and uses the Node ID as a key to search the placement information string of the node inside the topology file [2] of the machine.
4. Reads topology file of machine to acquire placement information string, number of sockets and number of cores per socket, for each allocated node.

[1] On a Cray machine, unit or process binding can be easily performed by adding the argument *-cc cpu* to the *aprun* command.

[2] The topology file on a Cray machine can be created using Cray's *xtprocadmin* utility. The placement information string of a node looks like *c11-2c0s15n3*, which means the position of the node in the machine hierarchy is: column 11, row 2, chassis 0, compute blade 15 and node 3.

5. Parses the placement information string of each node in order to obtain the value of each hierarchical level of the machine corresponding to each node.
6. Sorts the nodes with respect to all levels in *network hierarchy* i.e. at first performing the sorting according to Level[4] values of nodes, then for each Level[4] value (column), the nodes are sorted according to Level[3] value (row) and so on. For example:

   Level(4, 3, 2, 1, 0) = (0, 0, 1, 12, 3)
   Level(4, 3, 2, 1, 0) = (0, 0, 1, 13, 0)
   Level(4, 3, 2, 1, 0) = (0, 0, 1, 13, 1)
   ⋮
   Level(4, 3, 2, 1, 0) = (9, 1, 2, 1, 1)
7. Performs balanced distribution of total number of units in a cartesian grid
8. Performs balanced distribution of units per node, to form multicore groups in order to reduce inter-node communication (For example: 24 cores Hazel Hen node has balanced distribution of $(4 \times 3 \times 2)$. The lengths of coordinate directions of a 3D Cartesian grid (x,y,z) of total number of units should be divisible by the permutation of lengths of cartesian grid of units per node (permutations of (4,3,2)). This is necessary as our reordering method (Algorithm 1) performs multicore grouping at the node level and therefore the number of groups in each coordinate direction should fit Cartesian grid of total number of units.
9. Assigns new unit ID to each unit respecting the multi level network hierarchy, i.e. multicore groups of units are mapped as close as possible in the network hierarchy in order to reduce communication between distinct network hierarchy levels.
10. Finally, the reordered unit IDs are registered in the system.

After the last step, the new mapping, which takes multiple network hierarchy levels into account, is completed. The algorithm will result in perfect units mapping if the nodes are allocated in contiguous blocked manner, having best surface-to-volume ratio that will result in as little communication traffic as possible on all network hierarchy levels. If the nodes are allocated in a sparse manner, the algorithm attempts to preserve the locality through its HSFC like implementation.

At the very start, one spatial block of units (multi-core group) is mapped to the first node of the compute blade. Then, four spatial blocks of multi-core groups are mapped to the nodes on the same compute blade (if nodes are allocated in contiguous block manner). Similarly, sixteen spatial blocks of compute blades are mapped to the Rank 1 network (Chassis), and 6 spatial blocks of Chassis are mapped to the Rank 2 network (Group) and so on. If the nodes are allocated in a sparse manner (e.g. fewer than four nodes on a compute blade, fewer than sixteen compute blades on a Chassis and so on.), then nodes are mapped in a manner similar to HSFC, such that first available node in hierarchy is used to form a spatial block. Figure 5 shows an example of automatic hierarchical tasks mapping for a 2D nearest neighbor communication pattern.

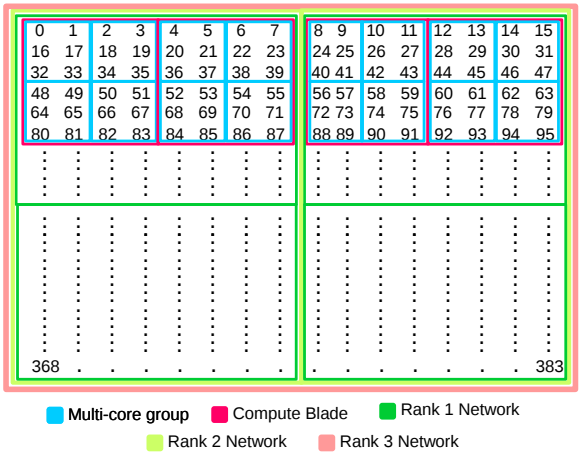

Figure 5: Example of automatic hierarchical units mapping for a 2D nearest neighbor problem of size (24×16). Please note that the example is just for elaborating the hierarchical units mapping and does not reflect the actual number of hardware instances at each network hierarchy level.

4. Performance Evaluation

We now evaluate the performance of 3D stencil communication kernel with and without using the automatic hierarchical units mapping feature. In order to see solely the impact of hierarchical units mapping on the performance, we have disabled the shared memory window feature of of DART-MPI for the benchmark shown later.

4.1 Evaluation Metric

In this stencil communication kernel, each DART unit communicates with six neighbors (left, right, upper, lower, front, and back). We use the blocking DART put operation for transferring the messages from one unit to another and the size of the messages is varied exponentially from 1 byte to 2 megabytes.

We are interested in the *relative performance improvement factor*. So the execution time of the stencil communication kernel using the automatic hierarchical units mapping feature in DART with respect to the default units mapping (as performed by job launcher on the Cray machine). The relative performance improvement factor is computed by taking the ratio of average execution time (over ten–thousand iterations) of stencil communication kernel using default against hierarchical units mapping.

Algorithm 1: Pseudocode of algorithm to compute new unit IDs which respect underlying machine's network hierarchy

Input : Balanced distribution of total number of units, number of units per node, number of one level below blocks at each network hierarchy level

Output: Unit IDs respecting network hierarchy

```
1  unitNumber ← 0
2  N4 ← numThirdLevelBlocksInFourthLevel[3]
3  N3 ← numSecondLevelBlocksInThirdLevel[3]
4  N2 ← numFirstLevelBlocksInSecondLevel[3]
5  N1 ← numNodesAtFirstLevel[2]
6  N0 ← numUnitsPerNode[3]
7  NT ← numTotalUnits[3]
   /* Given that the nodes are sorted according
      to multiple network hierarchy levels,
      compute the new unit IDs                    */
8  foreach x, y, z: 0 → N4 (0, 1, 2) do
9      foreach a, b, c: 0 → N3 (0, 1, 2) do
10         foreach d, e, f: 0 → N2 (0, 1, 2) do
11             foreach g, h: 0 → N1 (0, 1) do
12                 foreach i, j, k: 0 → N0 (0, 1, 2) do
13                     if unitNumber > totalUnits then
14                         break
15                     end
16                     fourthLevelOffset ← ..
17                     thirdLevelOffset ← ..
18                     secondLevelOffset ← d × N0[0] ×
                         NT[1] × NT[2] + e × N1[0] × N0[1]
                         × NT[2] + f × N1[1] × N0[2];
19                     firstLevelOffset ← g × N0[1] × NT[2]
                         + h × N0[2];
20                     unitID[unitNumber] ←
                         fourthLeveOffset + thirdLevelOffset +
                         secondLevelOffset + firstLevelOffset
                         + i × NT[1] × NT[2] + j × NT[2] +
                         k;
21                     unitNumber++;
22                 end
23             end
24         end
25     end
26 end
   /* Register the new unit IDs in run time
      system -- reordering                        */
```

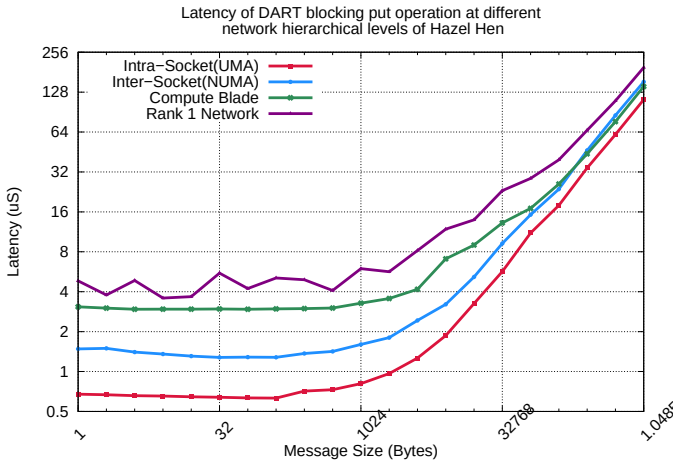

Figure 6: Average latency of transferring a message between two nodes at different hierarchical levels of the network. The message size is varied from 1 Byte to 1 MegaByte.

4.2 Benchmark Environment

The benchmarks are carried out on the Cray XC40 machine Hazel Hen at HLRS. Each node on Hazel Hen is based on Intel Xeon CPU E5-2680 v3 (30M Cache, 2.50 GHz) processors and comprises 24 cores (12 cores per socket). Cray's Aries interconnect provides node-node connectivity with multiple hierarchical network levels. We have two on node memory hierarchy levels, which are Uniform Memory Access (UMA) – intra-socket communication – and Non-uniform Memory Access (NUMA) – inter-socket communication. It's easy to exploit the one node hierarchical levels. In this paper we are more interested in showing results by exploiting the off-node network hierarchical levels (Section 3.4.1). However, just to highlight the importance of exploiting memory and network hierarchical levels, we show the average latency of message transfer time up to the rank 1 network of Hazel Hen in figure 6.

4.3 Results

Figure 7 shows the relative performance improvement factor of automatic hierarchical units mapping over default units mapping using 3D stencil communication kernel on 384 nodes (9,216 units) of Hazel Hen. The nodes were allocated in a sparse manner by Cray's job launcher, with small contiguous blocks of nodes. It can be seen that our units mapping provides an performance improvement by an average factor between 1.4 to 2.2.

5. Conclusions

In this paper we have presented specific features of DASH which resolve the issues we observed in the traditional PGAS implementations. We have shown in section 4 that our automatic hierarchical units mapping provides a no-

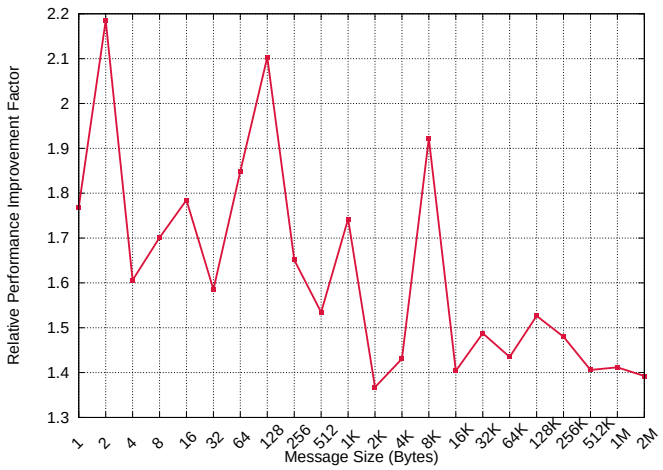

Figure 7: Relative performance improvement factor of 3D stencil communication kernel using default against automatic hierarchical units mapping, on 384 sparse nodes on Cray Hazel HazelHen XC40 machine. The message size for stencil communication is varied from 1 Byte to 1 MegaByte.

table performance improvement over default units mapping. A user can take advantage of this self adapting behavior without putting any effort to understand the complex machine hierarchy and perform custom coding to achieve such performance improvement. Furthermore, we plan to implement more high–level features in DASH by performing code transformations for set of scientific applications. These features will enable a user to represent the computation and communication patterns of scientific applications at very high–level abstractions in DASH, while DASH will take care of necessary code transformations.

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

This work was supported by the project *DASH* which is funded by the German Research Foundation (DFG) under the priority program "Software for Exascale Computing - SPPEXA" (2013-2015).

References

Karl Fürlinger, Colin Glass, Jose Gracia, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. Dash: Data structures and algorithms with support for hierarchical locality. In *Euro-Par 2014: Parallel Processing Workshops*, pages 542–552. Springer, 2014.

Kamran Idrees, Christoph Niethammer, Aniello Esposito, and Colin W Glass. Performance evaluation of unified parallel c for molecular dynamics. In *Proceedings of the 7th International Conference on PGAS Programming Models*, page 237.

Huan Zhou, Kamran Idrees, and José Gracia. Leveraging MPI-3 Shared-Memory Extensions for Efficient PGAS Runtime Systems In *Euro-Par 2015: Parallel Processing*, pages 373–384. Springer, 2015.

Huan Zhou, Yousri Mhedheb, Kamran Idrees, Colin W Glass, José Gracia, and Karl Fürlinger. DART-MPI: An MPI-based Implementation of a PGAS Runtime System In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 3. ACM, 2014.

Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian W Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. *Leveraging MPIs one-sided communication interface for shared-memory programming*. Springer, 2012.

Tobias Fuchs and Karl Fürlinger. Expressing and Exploiting Multidimensional Locality in DASH. *Springer Lecture Notes in Computational Science and Engineering*. Springer, November 2015.

Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.