DASH: Data Structures and Algorithms with Support for Hierarchical Locality

Karl Fürlinger[1], Colin Glass[2], Jose Gracia[2], Andreas Knüpfer[4], Jie Tao[3], Denis Hünich[4], Kamran Idrees[2], Matthias Maiterth[1], Yousri Mhedheb[3], and Huan Zhou[2]

[1] Ludwig-Maximilians-Universität (LMU) Munich
Computer Science Department, MNM Team
Oettingenstr. 67, 80538 Munich, Germany
[2] High Performance Computing Center Stuttgart
University of Stuttgart, Germany
[3] Steinbuch Center for Computing
Karlsruhe Institute of Technology, Germany
[4] Center for Information Services and High Performance Computing (ZIH)
TU Dresden, Germany

Abstract. DASH is a realization of the PGAS (partitioned global address space) model in the form of a C++ template library. Operator overloading is used to provide global-view PGAS semantics without the need for a custom PGAS (pre-)compiler. The DASH library is implemented on top of our runtime system DART, which provides an abstraction layer on top of existing one-sided communication substrates. DART contains methods to allocate memory in the global address space as well as collective and one-sided communication primitives. To support the development of applications that exploit a hierarchical organization, either on the algorithmic or on the hardware level, DASH features the notion of teams that are arranged in a hierarchy. Based on a team hierarchy, the DASH data structures support locality iterators as a generalization of the conventional local/global distinction found in many PGAS approaches.

1 Introduction

High performance computing systems are getting bigger and bigger in terms of the number of cores they are composed of and the degree of parallelism that needs to be exploited to successfully use them is becoming higher and higher. Billion-way parallelism is envisioned for Exascale-class machines [22] and one of the consequences of this trend is that data movement is becoming a more significant contributor to computing cost (in terms of time and energy) than the arithmetic operations performed on the data [8].

At the same time, while data comes to the fore in many areas of science, technology, and industry in the form of data-intensive science and big data, the programming models in use today are still largely compute-centric and do not support a data-centric viewpoint well. Consequently, programming parallel systems is difficult and will only get more complex as the Exascale era approaches.

PGAS (partitioned global address space) languages have long been proposed as a solution to simplifying the process of developing parallel software, but traditional PGAS solutions are ill equipped to address the two trends outlined above. First, most PGAS approaches offer only the differentiation between local and global data, a more fine-grained differentiation that corresponds to hierarchical machine models often envisioned for Exascale computing is not straightforward. Second, many existing PGAS solutions only offer basic data structures of built-in data types such as one-dimensional arrays and users have to develop more complex abstractions from scratch.

To address some of these issues, we are developing DASH, a PGAS approach that comes in the form of a C++ template library, supports hierarchical locality, and focuses on data structures and programmer productivity. The rest of this paper gives an overview of the project and its current status and is organized as follows: In Sect. 2 we start with the discussion of the high-level layered structure of our project. Sect. 3 describes the foundation of the project, the DART runtime layer and its interface to the C++ template library, in some detail. In Sect. 4 we describe how the abstractions of DASH can be used by an application developer. In Sect. 5 we discuss research projects that are related to DASH and in Sect. 6 we summarize the current status and discuss the further direction for our project.

2 An Overview of DASH

DASH [9] is a data-structure oriented C++ template library under development in the context of SPPEXA [23], the priority program for software for Exascale computing funded by the German research foundation (DFG). The DASH project consortium consists of four German partner institutions (LMU Munich, KIT Karlsruhe, HLRS Stuttgart, and TU Dresden) and an associated partner at CEODE in Beijing, China. The layered structure of the project is shown in Fig. 1; each project partner is leading the efforts for one of the layers.

A DASH-enabled application makes use of the data structures, algorithms, and additional abstractions (such as the hierarchical team construct) that are provided in the form of a C++ template library. DASH relies on a one-sided communication mechanism to exchange data, residing in the memory of multiple separate nodes, in the background, while providing the programmer with a convenient, local view.

As an example, Fig. 2 shows a simple stand-alone *hello world* DASH programme that allocates a small 1D array of integer keys and stores them over all available nodes. DASH follows the SPMD (single program, multiple data) model and the execution environment is initialized by the `dash::init()` call in line 3. Subsequently, `size` gives the number of participants in the program (denoted units) and `myid` identifies an individual unit. As an extra benefit of using DASH, rather than a local container such as an STL vector or array, the storage space is not limited by the locally available memory, but is extensible by adding more resources in a distributed memory setting. In the example code (Fig. 2), the DASH array allocated in line 8 is used to communicate a single in-

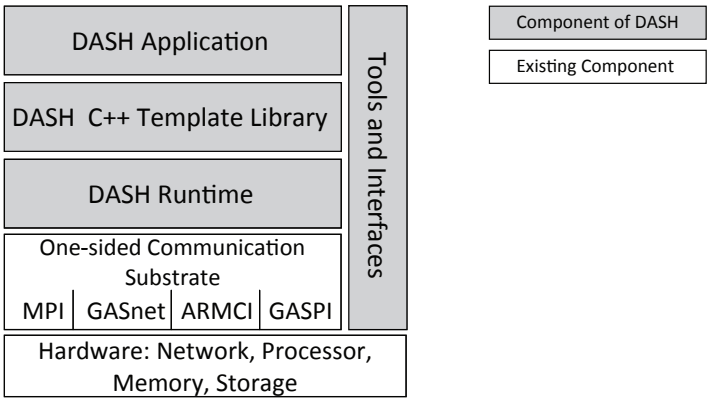

Fig. 1. The layered structure of the DASH project.

teger key from unit `0` to every other unit in the application. The communication is accomplished by overloading the subscript (`[]`) operator of the `dash::array` container and in lines 11–13 unit `0` stores the key at every (distributed) memory location of the array. The default layout for DASH one-dimensional arrays is blocks of elements over the available units. In our example this mapping implies that `key[`i`]` is stored on unit i and hence the access in line 18 (`key[myid]`) does not generate a communication event, since every unit reads its own local data item.

DASH builds upon existing one-sided communication substrates. A variety of one-sided communication solutions such as GASNet [4], ARMCI [18], OpenSHMEM [21], GASPI [12], and MPI exist, each with various features, restrictions and levels of maturity. DART (the DASH runtime), aims at abstracting away the specifics of a given substrate and provides services to the upper levels of the DASH stack. Most importantly, global memory allocation and referencing, as well as one-sided puts and gets, are provided by DART. In principle, any communication substrate can form the basis for DASH. However, since interoperability with existing MPI applications is among our design considerations, we chose MPI-3 one-sided (RMA, remote memory access) operations as the foundation for our scalable runtime implementation.

A DASH-enabled application can use the data structures and programming mechanisms provided by DASH. An application can be written from scratch using DASH, but we envision that more commonly existing applications will be ported to DASH, one data-structure at a time. In our project, two application case studies guide the development of the features of DASH. One application is a remote sensing Geoscience application from CEODE (China), the other is a molecular applications code contributed by HLRS Stuttgart. Finally, the tools and interfaces layer in Fig. 1 encompasses the integration of parallel I/O directly

to and from the data structures as well as the inclusion of a tools interface to facilitate debugging and performance analysis of DASH programs.

```
#include <libdash.h>

int main(int argc, char* argv[]) {
  dash::init(&argc, &argv);

  int myid = dash::myid();
  int size = dash::size();

  dash::array<int> key(size);

  if(myid==0) {
    for(i=0; i<size; i++) key[i]=compute_key(...);
  }

  dash::barrier();

  cout<<"Hello␣from␣unit␣"<<myid<<"␣of␣"
      <<size<<"␣my␣key␣is"<<key[myid]<<endl;

  dash::finalize();
}
```

Fig. 2. A *hello world* stand-alone DASH program that makes use of a small, shared 1D array for passing an integer key from unit `0` to all units in the program.

3 DART: The DASH Runtime Layer

DART is a plain-C based runtime that defines and implements central abstractions governing the development and usage of the DASH library and DASH applications. This section describes some of the key concepts that have been included in the first realization (v1.0) of the DART interface. In this first iteration of the interface we have been intentionally conservative and have limited ourselves to the necessities required to implement a functional version of the DASH library. A future iteration of the DART interface is likely to relax some of the restrictions and allow for a more expressive execution model. Specifically, DART v1.0 does not contain a tasking or explicit code execution model. Instead, data can be transparently accessed and computed on by regular operating system threads. Work is currently in progress to identify the requirements for extending DART to GPUs and in the context of this work a DART task execution model will be developed.

In the DART execution model, the individual participants of a DASH/DART program are called *units*. The generic name unit was chosen because other related terms such as process or thread already have a specific meaning in a variety of contexts and with DART we would like to have the conceptual freedom to map a unit onto any operating or runtime system concept that fits our requirements. A DASH application follows the SPMD programming model and the total number of units that exist is fixed at program start and does not change in the course of the program execution. Units are organized into *teams* and one team is referred to as `DART_TEAM_ALL`, comprising all existing units. Every unit in a team has an integer identifier (ID) which remains unchanged throughout the lifetime of the team; a unit's ID with respect to `DART_TEAM_ALL` is referred to as the unit's *global ID*. Like units, teams are identified by integer IDs, but teams can be created and destroyed dynamically. A unit's ID with in a team other than `DART_TEAM_ALL` is referred to as a local id.

A new team in DART is formed by specifying a subset of an existing parent team. The team creation routine `dart_team_create()` is a collective operation on the parent team and returns an integer identifier for the new team. Since we want to support large hierarchical machines and a localized sub-team creation that requires the involvement of the whole application would be prohibitively expensive, the new team ID does not have to be globally unique. However, the following localized uniqueness guarantees are provided:

- The same team ID is returned to all units that are members of the new team.
- The team ID is unique with respect to the parent team.
- If a unit is participating in two teams, t_1 and t_2, then it is guaranteed that t_1 and t_2 will receive different identifiers.

Teams are a mechanism for representing the hierarchical structure of algorithms and machines in a program [16]. An example for a team hierarchy representing the machine hierarchy of a system like SuperMUC (which has the notion of interconnected *islands* [24]) is shown in Fig. 3. Clearly it is not desirable for every team creation operation to require global synchronization – creating the sub-teams of team $t1$ (island 1) should only involve team $t1$ and not require any involvement from the rest of the machine. A straightforward algorithm that we use in our implementation to guarantee the above requirements, while avoiding global communication, keeps a unit-local *next_team_id* counter and performs a maximum reduction among all members of the parent team. After creating the new team, the *next_team_id* counter on all units of the new team is set to $max + 1$.

An important abstraction provided by DART is the virtual global memory space and a mechanism to refer to data items residing in it (i.e., a global pointer). A DART global pointer is a structure of 128 bits which has a 32 bit field for identifying the unit providing the memory, a 64 bit offset or local address field and 32 bits for flags and a segment identifier. Importantly, the global pointer on the DART level has no phase information associated with it. However, a similar construct is provided on the C++ (DASH) level, which then does contain appropriate phase information needed to decide when to switch between units.

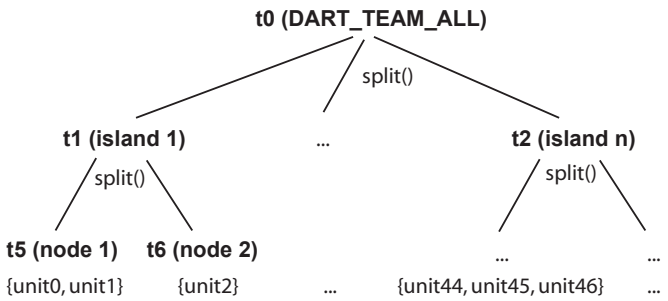

Fig. 3. An example team hierarchy for an execution of an application on a machine like SuperMUC, with a hierarchical interconnect architecture.

The DART virtual global memory space is composed of the memory segments contributed by the units of an application on demand. Visibility of and accessibility to memory is based on the team concept. The team-collective operation `dart_team_memalloc_aligned`(t, `nbytes`) allocates `nbytes` in the memory of every unit in team t. This memory is accessible only by the members of team t and is said to be *team-aligned* and *symmetric*. Symmetric refers to the property that all units allocate the same amount of memory, while team-aligned denotes that every unit can compute the global pointer to any location in the global memory by simple arithmetic. A second memory allocation function supported in DART is `dart_memalloc`, which allocates a "local global" memory that is accessible by any unit (the memory has implicit associativity with `DART_TEAM_ALL`), but the call is local. The two memory allocation functions are depicted in Fig. 4.

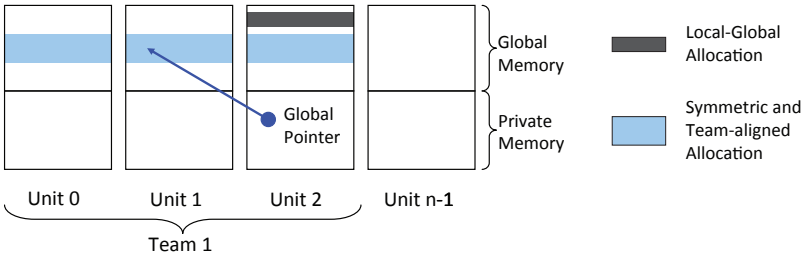

Fig. 4. The two types of memory allocation functions supported by DART.

4 Using DASH in Applications

The overall goal of DASH is to provide a programmer with data structures that can be used productively on large, parallel machines. C++ was chosen as a host language for our project, because it is used in an increasingly large number of HPC and data-science applications [27] and it has powerful features that allow us to realize PGAS semantics efficiently. Specifically, we use templates to provide efficient implementations of containers for user defined types and operator overloading, thus achieving a PGAS abstraction without relying on a custom compiler.

Applications can be written from scratch using DASH and existing applications can furthermore be adapted to use DASH data structures. A stand-alone application is shown in Fig. 2.

PGAS approaches are often classified into local-view and global-view solutions, where global-view describes a situation in which the programming entities are global objects and it is not syntactically obvious, whether accessed data is local or remote. In local-view approaches, this syntactic visibility is always guaranteed, often in the form of an explicit co-index that explicitly states the location of data. Since this distinction is important for performance and energy efficiency, there is always some way of telling whether data is remote or local (say, by computing an affinity expression), but global-view does not force this differentiation to be syntactic.

With DASH we largely follow the global-view approach. The constructors of our data containers are collective operations on a team and every participating unit receives an object representing the entire data structure. In several cases, this global-view approach allows us to use a DASH container (instead of a standard STL container) in a straightforward manner. An example is shown in Fig. 5. A 1D array, stored over several units, is used in combination with the standard library's sort algorithm in line 11. Naturally, this approach has several drawbacks: `std::sort()` is a sequential sorting procedure that only engages one unit at a time, resulting in fine-grained communication, as the sorting algorithm fetches data items to compare with. However, despite these drawbacks, we envision that the ability to seamlessly replace STL with DASH containers can be useful in some situations for prototyping and removing memory limitations.

Accompanying the DASH data structures, we are investigating algorithms analogous to those found in the STL to take into account data distribution and parallelism (i.e., a parallel `dash::sort()`). Additionally, the standard *owner-computes* paradigm is supported by DASH in the form of local iterators (`lbegin()`, `lend()`), as shown in line 17 of Fig. 5. These local iterators allow each unit to access its local portion of the data and they correspond to the classic two-level affinity model (local/remote) of PGAS. As a generalization of this concept we are investigating hierarchical locality iterators by leveraging the hierarchical team concept in the DASH data containers.

```
// split the units into 8 teams (e.g., one per node)
dash::team nodeteam = dash::TeamAll.split(8);

// allocate an array over the node team
dash::array<double> b(100000, nodeteam);

// use the DASH container in place of an STL container
// note sequential sort and perf. implications
int myid=nodeteam.myID();
if(myid==0) {
  std::sort(b.begin(), b.end());
}

// to use containers with standard algorithms in parallel,
// local iterators lbegin(), lend() are provided
// this fills the array in parallel (aka. 'owner computes')
std::fill(b.lbegin(), b.lend(), 23+myid);
```

Fig. 5. A small example that shows teams and DASH containers used with global-view and local-view semantics.

5 Related Work

A number of realizations of the PGAS concept exist. UPC [26] is an ANSI C dialect that extends C with the ability to declare shared pointers and data items. The portable Berkeley UPC implementation relies on GASNet [4] for communication, while some vendors directly target their own low-level interconnect API. Co-array Fortran [20, 17] extends the notion of standard Fortran arrays with a co-index to specify the process holding the array. The molecular dynamics application has already been ported to UPC which can be used for performance comparison with DASH porting in future [14]. The DARPA sponsored HPCS (High Productivity Computing Systems) languages X10 [7], Fortress [1], and Chapel [6] followed the PGAS model, of which Chapel remains the most actively developed and used.

PGAS has been realized in the form of a library in the past. Global Arrays [19] is an early example of an API for shared memory programming on distributed memory machines, primarily used in the context of quantum chemistry applications. GASPI [12] is an effort to standardize an API for PGAS programming developed by Fraunhofer, it features support for fault tolerance, by supporting timeouts for all non-local operations. OpenSHMEM [21] is a community effort to standardize the various dialects of SHMEM, which provides a strongly typed API for shared memory programming on distributed memory machines.

Recently, C++ has been used as a vehicle for realizing a PGAS approach in the UPC++ [29] and Co-array C++ [15] projects. While the DASH runtime is based on MPI, UPC++ is based on GASNet. Porting an existing MPI application will therefore be more straightforward using DASH. Co-array C++ follows

a strict local-view programming approach and is somewhat more restricted than DASH and UPC++ in the sense that it has no concept of teams.

STAPL [5, 13, 25] is a C++ template library for distributed data structures supporting a "shared view" programming model that shares several goals with DASH. The library provides a local view on data, while it can be physically spread over several nodes. The authors of STAPL mention PGAS as related work, but don't seem to consider their own work a PGAS solution. STAPL does provide a large set of data containers and places a lot of emphasis on extensibility and configurability – it does however not seem to be intended for classic HPC applications.

Hierarchical computation and data structure layout have been explored in several approaches before. Sequoia [10, 2] is a programming approach (language, compiler, and runtime system) for exploiting the memory hierarchy of modern machines in a portable way. Sequoia provides tasks that are restricted to access only local memory and the only supported way of communication between tasks is through parameters passed to tasks and the return values. Thus, a programmer expresses an application as a hierarchy of tasks, and this abstract hierarchy is later mapped to a concrete machine hierarchy. In Sequoia this mapping is done by the compiler, in Hierarchical Place Trees (HPT) [28] the mapping is done by the runtime. HPT are an extension to the flat place concept of X10. Hierarchically Tiled Arrays (HTA) [3, 11] are data structures that enable locality and parallelism of array intensive computations, by using a block-recursive storage scheme. Several implementations of HTA exist, including one for C++. Finally, the work of Kamil et al. [16] explores additions and modifications to the SPDM programming model to support a hierarchical concept of teams. The DASH concept for hierarchical teams is inspired by his work.

6 Conclusion and Future Work

We have presented an overview of the DASH project. One goal of DASH is to make the PGAS (partitioned global address space) concept available to a wider range of application developers. PGAS languages often suffer from limited acceptance, because existing applications have to be ported to the new language as a whole. With DASH we offer a way to port C++ MPI applications incrementally (one data structure at a time). DASH has the advantage that it is not a new language to learn and does not require a custom compiler or pre-processor. Instead, DASH is realized as a C++ template library and operator overloading is used to provide the PGAS semantics on the data containers.

As high performance computing machines are getting bigger and more hierarchical on the way to Exascale, we plan to exploit the flexibility of the library-based approach DASH to address this trend and include support for hierarchical locality in our data structures. To this end, we are supporting the concept of hierarchical teams. Teams determine visibility and accessibility of the DASH data structures and allow for the realization of hierarchical locality iterators.

We are presently in the process of putting together a first public release of our DASH software stack. This first release will contain a generic 1D distributed array as the basic data structure, and it will be based on the first realization of our MPI-based DART runtime. The next steps for the projects will be to include additional data structures, such as multi-dimensional arrays, and distributed lists. We will continue our work on flexible data layout mappings and explore concepts to support hierarchical locality. With these data structures and concepts in place, work on the DASH-enabled molecular dynamics and remote sensing applications can proceed and thereby guide the next iteration of DASH features.

References

1. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, and Sam Tobin-Hochstadt. The fortress language specification. sun microsystems. *Inc., September*, 2006.
2. Michael Bauer, John Clark, Eric Schkufza, and Alex Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in Sequoia. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2011)*, pages 13–24, February 2011.
3. Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006)*, PPoPP '06, pages 48–57. ACM, 2006.
4. Dan Bonachea. GASNet specification, v1. *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.
5. Antal Buss, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M Amato, Lawrence Rauchwerger, et al. STAPL: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, page 14. ACM, 2010.
6. Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, August 2007.
7. Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM Sigplan Notices*, 40(10):519–538, 2005.
8. Bill Dally. Power, programmability, and granularity: The challenges of exascale computing, 2011. IPDPS 2011 Keynote Address.
9. DASH project webpage `http://www.dash-project.org/`.
10. Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'06)*, 2006. Tampa, FL, USA.
11. Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua, and Christoph Von Praun. Optimization techniques for efficient HTA programs. *Parallel Comput.*, 38(9):465–484, September 2012.

12. Daniel Grünewald and Christian Simmendinger. The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*, 2013. Edinburgh, Scotland.
13. Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL parallel graph library. In *LCPC*, pages 46–60, 2012.
14. Kamran Idrees, Christoph Niethammer, Aniello Esposito, and Colin W. Glass. Evaluation of unified parallel C for molecular dynamics. In *Proceedings of the Seventh Conference on Partitioned Global Address Space Programing Models (PGAS '13)*, New York, NY, USA, 2013. ACM.
15. Troy A Johnson. Coarray C++. In *7th International Conference on PGAS Programming Models*, 2013. Edinburgh, Scotland.
16. Amir Ashraf Kamil and Katherine A. Yelick. Hierarchical additions to the SPMD programming model. Technical Report UCB/EECS-2012-20, EECS Department, University of California, Berkeley, February 2012.
17. John Mellor-Crummey, Laksono Adhianto, William N. Scherer, and Guohua Jin. A new vision for coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09)*, New York, NY, USA, 2009. ACM.
18. Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, pages 533–546. Springer, 1999.
19. Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.
20. Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.
21. Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. OpenSHMEM - Toward a unified RMA model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer US, 2011.
22. John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.
23. SPPEXA webpage `http://www.sppexa.de/`.
24. SuperMUC system description `http://www.lrz.de/services/compute/supermuc/systemdescription/`.
25. Gabriel Tanase, Antal A. Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL parallel container framework. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2011)*, pages 235–246, February 2011.
26. UPC Consortium. UPC language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
27. Simon Wong, Galway Danica Stojiljkovic, Stelios Erotokritou, George Tsouloupas, Pekka Manninen, David Horak, and Georgi Prangov. PRACE training and education survey. Technical report, PRACE, December 2011. Available online at `http://prace-ri.eu/IMG/zip/d4.1_2.zip`.
28. Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A portable abstraction for task parallelism and data movement. In *Proceedings of*

the 22nd international conference on Languages and Compilers for Parallel Computing, LCPC'09, pages 172–187. Springer-Verlag, 2010.
29. Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: A PGAS extension for C++. In *28th IEEE International Parallel & Distributed Processing Symposium*, 2014.