SHMEM+: A Multilevel-PGAS Programming Model for Reconfigurable Supercomputing

VIKAS AGGARWAL, ALAN D. GEORGE, CHANGIL YOON, KISHORE YALAMANCHILI and HERMAN LAM
NSF Center for High-Performance Reconfigurable Computing (CHREC)
ECE Department, University of Florida

Reconfigurable computing (RC) systems based on FPGAs are becoming an increasingly attractive solution to building parallel systems of the future. Applications targeting such systems have demonstrated superior performance and reduced energy consumption versus their traditional counterparts based on microprocessors. However, most of such work has been limited to small system sizes. Unlike traditional HPC systems, lack of integrated, system-wide, parallel-programming models and languages presents a significant design challenge for creating applications targeting scalable, reconfigurable HPC systems. In this paper, we extend the traditional Partitioned Global Address Space (PGAS) model to provide a multilevel integration of memory, which simplifies development of parallel applications for such systems and improves developer productivity. The new multilevel-PGAS programming model captures the unique characteristics of reconfigurable HPC systems, such as the existence of multiple levels of memory hierarchy and heterogeneous computation resources. Based on this model, we extend and adapt the SHMEM communication library to become what we call SHMEM+, the first known SHMEM library enabling coordination between FPGAs and CPUs in a reconfigurable, heterogeneous HPC system. Applications designed with SHMEM+ yield improved developer productivity compared to current methods of multi-device RC design and exhibit a high degree of portability. In addition, our design of SHMEM+ library itself is portable and provides peak communication bandwidth comparable to vendor-proprietary versions of SHMEM. Application case studies are presented to illustrate the advantages of SHMEM+.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms: Design, Languages, Performance

Additional Key Words and Phrases: Reconfigurable computing, parallel programming, programming language, programming model, productivity, portability

1. INTRODUCTION

High-performance computing (HPC) is a critical enabling technology for the advancement of science and engineering, supporting multi-scale simulations and experiments that drive breakthroughs in an ever-broadening range of fields. The field

Author's address: NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, University of Florida, Gainesville, Florida 32611; email: aggarwal@chrec.org.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20?? ACM 1529-3785/20??/0700-0001 $5.00

of HPC is currently undergoing a major transformation brought on by advances in device technologies as well as new generations of fixed-logic, reconfigurable-logic, and/or heterogeneous multicore and many core devices. These technologies are driving systems to become ever more powerful and efficient but unfortunately also more complex to program, with multiple types and levels of hardware parallelism to be understood and exploited.

A special class of such systems featuring reconfigurable computing (RC), based on closely coupled microprocessors and FPGAs, offers an attractive solution for HPC. Numerous studies have demonstrated that RC systems can achieve performance improvements ranging from 10× [Shih et al. 2008] to more than 1000× [Storaasli 2008] over their microprocessor-based counterparts while concomitantly reducing energy consumption. Despite their superior performance, RC systems are yet to make a significant impact on the HPC market, largely because of increased complexity of application-design which has been the focus of much commercial activity. Although advances in device-level languages and tools for FPGAs have been the subject of much research and commercial activity, system-level design issues have largely been unaddressed. Such is the case with communication and synchronization between multiple devices in RC systems. Unlike traditional HPC systems, lack of integrated, system-wide, parallel-programming models and languages has limited most RC applications to small systems. The characteristic differences between RC systems and traditional HPC systems, such as additional levels of memory in the system and different execution models of heterogeneous devices present in the system, warrant a programming model which can address these differences. Currently, application developers employ ad-hoc methods and multiple libraries (and APIs) to incorporate inter- and intra-node communication and synchronization for large-scale RC systems. As a result, the development productivity for scalable, parallel RC applications has suffered.

Although shared-memory models have been prevalent in HPC for decades, recently, newer models providing a programmer with a partitioned, global address space (PGAS) view for abstraction have been gaining popularity, such as Unified Parallel C (UPC) [El-Ghazawi et al. 2001; Carlson et al. 1999], SHMEM [SGI], Co-Array Fortran [Numrich and Reid 1998], and Titanium [Yelick et al. 1998]. By extending the memory hierarchy to include an additional, higher-level global memory layer that is partitioned between nodes in the system, such languages and libraries allow for explicit or implicit one-sided data exchange (i.e. put, get) through reading and writing of global variables. Although designed for traditional HPC systems, the PGAS-based model has the requisite simplicity, syntax, and semantics to meet the needs of coordination amongst FPGA and CPU devices in reconfigurable HPC systems. However, the model needs to be adapted and extended to work with such systems. The virtual memory layer needs to be extended to incorporate and abstract multiple levels of memory present in RC systems. In particular, the SHMEM communication library stands out as a strong candidate for extending to RC systems due to its innate simplicity, low overhead, support for a partitioned global address space (PGAS), and emphasis upon explicit, fast one-sided communications. However, architectural differences introduced by incorporating RC devices in the system warrant re-examination of some concepts and semantics traditionally

associated with its common usage.

In this paper, we introduce a multilevel-PGAS programming model for RC systems, which abstracts the memory hierarchy available in the system, and presents the designer with a flattened, unified view of the system memory. Furthermore, we employ the model to develop SHMEM+ (i.e. an extended SHMEM library), the first known implementation of SHMEM that enables communication and synchronization between FPGAs and CPUs in a scalable, reconfigurable HPC system. Although our work focuses on HPC systems and applications, the proposed multilevel PGAS model and SHMEM+ library can be employed by high-performance embedded systems (HPEC) systems which employ a variety of embedded processors and accelerators. In addition, the ideas and concepts explored here can also be extended to systems based on other types of accelerators such as GPUs, many-core processors, etc. The potential uses and impact of SHMEM+ on such systems will be explored in future research.

Using SHMEM+, designers can create scalable, parallel applications that execute over a mix of microprocessors and FPGAs. The high-level abstraction provided by SHMEM+ can yield significant improvement in developer productivity. Concomitantly, for the decomposed tasks of a parallel application, developers of FPGA cores can employ high-level synthesis tools and languages (e.g. Impulse-C, Carte-C, Handel-C, etc.) for creating hardware designs for FPGAs to further improve productivity. We analyze the performance of our implementation of SHMEM+ and investigate its inherent strengths through multiple case studies.

The remainder of this paper is organized as follows. Section 2 describes previous work. Section 3 provides a description of the multilevel-PGAS programming model. Section 4 gives an overview of the design of SHMEM+. In Section 5, we benchmark the performance of data-transfer routines available in SHMEM+. We also present two case studies to illustrate the design methodology and evaluate the advantages of application design using SHMEM+. Finally, Section 6 summarizes the work with conclusions and directions for future work.

2. BACKGROUND AND RELATED RESEARCH

A variety of projects have attempted to simplify application design for FPGAs by employing High-level Languages (HLLs) such as Impulse-C, Handel-C, etc. While enabling faster hardware designs for an FPGA, HLLs typically do not address the system-level issues involved with parallel programming on reconfigurable HPC systems. Traditionally, developers of parallel programs have performed coordination between tasks using either message-passing libraries such as MPI [MPI] or shared-memory libraries such as OpenMP [OpenMP]. Recently, languages and libraries that present a partitioned global address space (PGAS) to the programmer, such as UPC [El-Ghazawi et al. 2001; Carlson et al. 1999] and SHMEM [SGI], have become more visible and popular. These languages provide a simple interface for developers of parallel applications through implicit or explicit one-sided data transfer functions, while providing comparable performance to message-passing libraries [Nishtala et al. 2009]. However, since such languages and libraries were developed for traditional HPC systems, they have been typically limited to homogeneous execution contexts of a cluster of microprocessors.

```
#include <stdio.h>
#include <shmem.h>
#include <intrinsics.h>

int me, npes, i;
int *source, *dest;
main()
{
 shmem_init();
 me = my_pe(); /* Get PE information */
 source = shmalloc(4*8); /* Allocate data in shared memory */
 dest = shmalloc(4*8);
 if(me == 1) /* Perform send on PE 1 */
  for(i=0; i<8; i++) source[i] = i+1;
   /* put source data at PE1 to dest at PE0*/
   shmem_putmem(dest, source, 8*sizeof(dest[0]), 0);
 /* Make sure the transfer is complete */
 shmem_barrier_all();
 /* Print from the receiving PE */
 if(me == 0) {
  printf(" DEST ON PE 0:");
  for(i=0; i<8; i++)
   printf(" %d%c", dest[i], (i<7) ? ',' : '\n');
 }
}
```

Fig. 1. Source code for a SHMEM application that transfers an array of data from PE1 to PE0.

In particular, the SHMEM communication library is currently experiencing a growth in interest in the HPC community due to its innate simplicity, low overhead, and emphasis upon explicit, high-bandwidth, one-sided communications. The SHMEM communication library consists of a set of routines that allow exchange of data between cooperating parallel processes (called processing elements or PEs). Programs developed using SHMEM follow the single-program, multiple-data model (SPMD) [Darema 2001], and are similar in style to programs based on MPI. SHMEM routines support remote data transfers through put (or get) operations, which transfer data to (or from) a different PE using remote pointers which allow direct references to data objects owned by the remote PE. Several other operations are also supported such as broadcast, collective reduction, synchronization operations, and atomic memory operations. Figure 1 shows the source code of an example application which uses the SHMEM library to transfer an array of data from “source” variable on PE1 to “dest” variable on PE0.

Owing to the emergence of a plethora of devices that are used for application acceleration and coupled with microprocessors in HPC, there has been a quest for exploring parallel-programming models that are better suited for heterogeneous, RC systems. TMD-MPI [Saldana et al. 2008] extends the MPI library to support message-passing between heterogeneous devices, such as a mix of FPGAs and microprocessors. [Aggarwal et al. 2009] adapts a message-passing model to generate efficient communication infrastructure between various devices of a heterogeneous system. Other research groups have shown interest in asynchronous execution in the PGAS model, leading to Asynchronous PGAS (APGAS) [APGAS 2009], which lays the foundation for active-message programming and fine-grained concurrency. However, APGAS is largely tailored towards spawning massively parallel, multi-

threaded kernel computations at run-time, on accelerators such as GPUs and not intended for FPGA devices.

Some researchers have attempted to build hybrid models using multiple models for a system [Farreras et al. 1997]. System-level libraries and languages such as MPI and UPC were used for coordination between tasks executing on different nodes of a cluster, and libraries such as OpenMP for coordination between tasks within each node. Hybrid models require the designer to partition their design into multiple levels and acquire expertise with multiple programming models, languages, libraries, and tools. By contrast, we attempt to abstract these details from an application designer and present an integrated programming model and library.

[El-Ghazawi et al. 2008] extends the UPC programming model to abstract a system of microprocessors and accelerators through a two-level hierarchy of parallelism. While their work shares the same goal of providing application developers with a unified programming model, their approach is quite different than ours. Their approach relies on identifying and extracting sections of code, specified in a UPC program, which are amenable to hardware acceleration, and re-directing them through a source-to-source translator and a high-level synthesis tool to generate hardware designs. Instead of providing a means for creating hardware designs, we provide a parallel programming model, amenable to HPC systems which have a hierarchy of computational devices and memory resources, and defer to and leverage the efficiency of existing and emerging high-level synthesis tools to raise the abstraction for device-level design and generate the hardware.

This paper extends our previous work in [Aggarwal et al. 2009] by refining the design of SHMEM+ to yield better performance, enable more functionality, and support multiple FPGAs on each node. In addition, SHMEM+ has been ported and evaluated on our new Novo-G RC supercomputer (details in Section 5). We further analyze the scalability, portability and productivity advantages of applications developed using SHMEM+ library through case studies on two different systems.

3. MULTILEVEL PGAS

Next-generation RC systems will be targeting FPGA devices in their system architectures in exotic ways to extract performance, ranging from closely coupled, in-socket accelerators to PCIe-based accelerator cards. Figure 2 depicts an example RC system, where every node contains a set of processing units (PUs), each a microprocessor or FPGA. With FPGA devices and multicore CPUs, each with one or more associated memory modules, all within a single node, becoming pervasive in high-end computing systems, the existence of multiple levels of memory hierarchy and different permutations of communication is becoming increasingly difficult to ignore. As a result, application developers are presented with a daunting task of orchestrating data amongst heterogeneous devices and several memory components by employing multiple APIs.

There is a need for a parallel-programming model that provides application developers with a high level of abstraction and presents a simplified view of the system, somewhat akin to that provided by the global memory layer in PGAS. However, there are various challenges involved in applying an existing parallel-programming

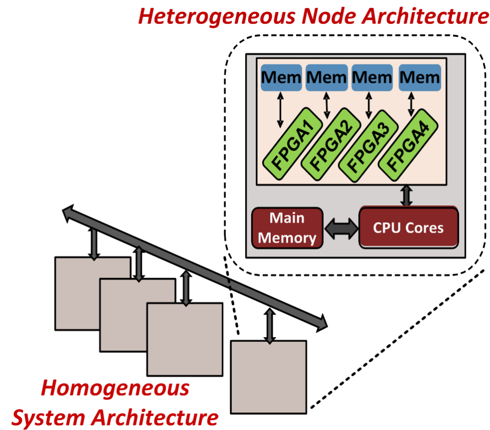

Fig. 2. System architecture of a typical RC machine.

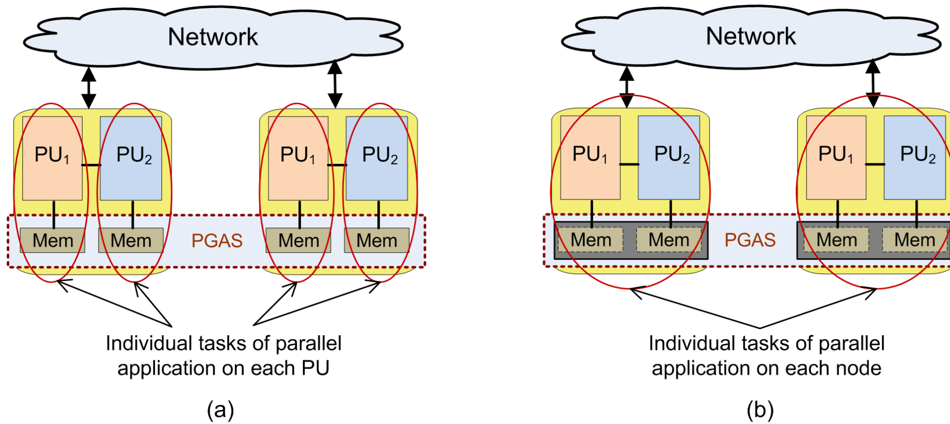

Fig. 3. High-level abstractions for programming heterogeneous systems. (a) An ideal programming abstraction for application developers, (b) A more practical and realizable approach.

model such as PGAS to reconfigurable HPC systems. Some of the concepts and semantics associated with PGAS-based programming model on traditional systems are not directly applicable to such hybrid systems. For example, a majority of parallel programs are described using the SPMD model, where each node in the computational system executes the same program while working on a different part of input data. Heterogeneous systems comprised of devices with different programming paradigms often require an application developer to create separate programs, one for each type of device in the system, and necessitate a re-definition of SPMD for such systems. Similarly, the multi-tier memory hierarchy that exists in reconfigurable HPC systems warrants a re-examination of the distribution of the virtual, global memory layer of PGAS programming model over the physical memory resources of a system.

From application developers' point of view, an ideal programming model should

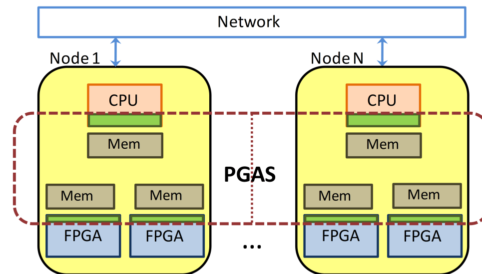

Fig. 4. Distribution of memory and system resources in multilevel PGAS.

provide an abstraction where the heterogeneous devices present in the system are treated as logically equivalent PUs (Figure 3a). Using such a model, each PU will execute a program instance of a SPMD application, obtained by translating the source code into logically equivalent operations in different programming paradigms. The PGAS interface on each PU would be responsible for presenting a logically homogeneous system view to the application developers. While it may provide a simplified view of the system, such an abstraction would be difficult to implement and may lead to inefficient utilization of system resources. For example, FPGA devices yield exceptional performance for computations which have a high degree of parallelism, but can lead to inefficiencies when implementing complete functionality of a SPMD program.

A more practical solution would raise the level of abstraction for application developers while making efficient use of the specialized resources present in a system. Figure 3b shows such an approach, where each individual task of a SPMD application is further partitioned across, and collectively executed by all the PUs on a node. Such a solution can also extend the concept of partitioned, global address space to a multilevel abstraction, which integrates a hierarchy of multiple memory components into a single, virtual memory layer. We call this model multilevel PGAS.

Figure 4 shows the physical distribution of memory components that form the global address space in multilevel PGAS. Memory blocks associated with all PUs in the system, irrespective of their physical location and hierarchy in the system architecture, can form a part of the virtual memory layer and have globally unique memory addresses in the system. Both CPUs and FPGAs provide interfaces required for the global memory abstraction for their corresponding memory blocks. Note that, all physical memory blocks do not have to be a part of the PGAS. The memory blocks that do not form a part of the virtual, global memory layer can be used by their PUs for storing local variables. It should be noted that memory blocks shown in Figure 4 correspond only to off-chip memory resources for the focus of our work. On-chip memory structures of an FPGA such as block RAMs and register files are treated as local storage and not exposed as a part of PGAS. Such modeling of local storage is similar to that of microprocessor cache and registers, which are hidden from the PGAS layer in traditional HPC systems. Such resources were not included in the global address space in our design because the memory consistency required by parallel applications may preclude the usage of BRAMs as

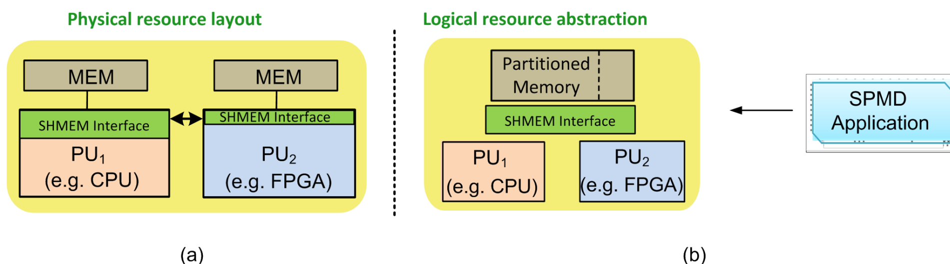

Fig. 5. (a) Physical resource layout of a typical RC system, (b) Logical abstraction provided by multilevel-PGAS model.

shared resources in most cases. However, our framework does not prevent the usage of such resources in the global address space if it can be supported by the target FPGA platform.

Figure 5 depicts a detailed view of the physical distribution of resources within each node and its equivalent logical abstraction provided by the multilevel-PGAS model (used by SHMEM+). Although the figure depicts two processing units per node, one CPU and one FPGA, it can be generalized to include any number and variety. As shown in Figure 5a, the global address space, partitioned across multiple nodes in the system, is composed of memory blocks which are physically distributed across different processing units within a node. However, the logical abstraction presented to a designer (shown in Figure 5b) is a flattened view of the node's shared memory. Thus, application designers do not have to understand the distribution of data over the physical memory resources when accessing a remote node.

The PGAS interface on each node is responsible for providing application designers with an abstraction of a single, integrated memory block. Similar to the case for memory resources, the logical view of the PGAS interface presented to the developer is different from its physical implementation. The physical implementation of the interface itself is system-dependent and can be realized in different ways by system architects. While each node provides the entire functionality required by the PGAS interface, each PU within a node may implement only a subset of this functionality. The distribution of these responsibilities amongst the PUs within a node is dictated by their capabilities in the system. For example, in our current design, the CPUs provide a majority of the SHMEM functionality and the FPGAs only provide assistance for transfers to and from the FPGA's memory using the vendor-specific memory controllers. As future work, we intend to investigate the feasibility of FPGA-initiated transfers, which will require more extensive support from FPGAs and may lead to some resource utilization on the FPGAs by SHMEM+, unlike our current design. The multilevel-PGAS model supports two additional features which help in attaining high performance for applications. First, it allows application developers to specify affinity of various application data to specific memory components within a node during memory allocation. Therefore, the data can be placed in a memory block closer to the processing unit that operates on it most frequently. Second, multilevel-PGAS model requires explicit transfers between local memory components. In systems equipped with multiple non-coherent memory blocks within a node, DMA operations are often employed for data transfer between

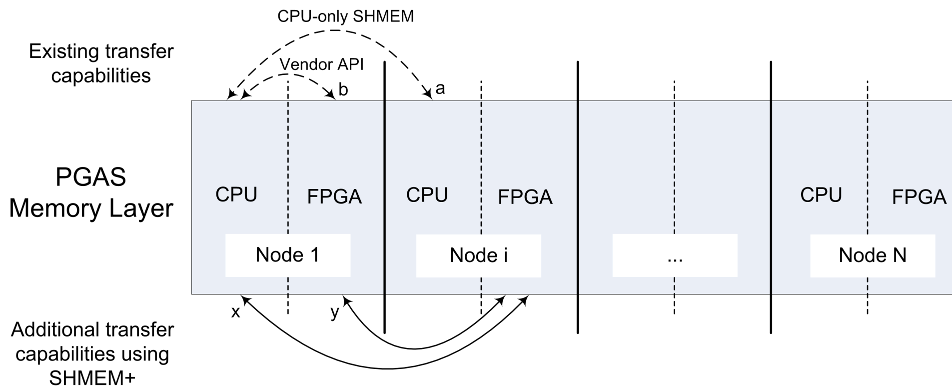

Fig. 6. Data transfer choices available to the developer with SHMEM+.

different memory blocks, which are expensive operations and can significantly hamper application performance. Having explicit calls for data transfers within a local node eliminates the possibility of inefficiencies caused by transparent but expensive transfers which are implicitly embedded in the application code.

4. OVERVIEW OF SHMEM+

Using the multilevel-PGAS programming model, we extend conventional SHMEM to become what we call SHMEM+, a communication library which enables additional communication capabilities between heterogeneous devices. Using SHMEM+, designers can create highly scalable applications that execute over a mix of microprocessors and FPGAs. Previous implementations of the SHMEM API have targeted specific systems [Cray T3ETM Fortran Optimization Guide - 004-2518-002] and often lacked portability. SHMEM+ is built over services provided by Global Address Space NETworking (GASNet from UC Berkeley)[Bonachea and Jeong 2002] which is a language-independent, communications middleware that provides network-independent, high-performance primitives tailored for implementing parallel GAS languages. As a result, SHMEM+ can be easily ported to other systems that are supported by GASNet by simply modifying the FPGA interfaces that employ vendor-specific APIs.

SHMEM+ provides developers with a high-productivity environment for establishing communication in an RC application, by providing developers with several choices for data transfers between devices of a heterogeneous RC system, some of which did not exist in conventional SHMEM library. Figure 6 illustrates the different options for data transfers provided by SHMEM+ in a system with a CPU and an FPGA on each node. The existing transfer capabilities are marked by labels ‘a’ (CPU-only SHMEM) and ‘b’ (platform-specific APIs) in the figure, and the ones introduced by SHMEM+ are labeled as ‘x’ and ‘y’. While these additional data-transfer options simplify the process of developing parallel applications and improve productivity, the developers should understand the tradeoffs associated with such transfers. For example, direct transfer between two remote FPGAs eliminates the need for a developer to carefully orchestrate the data through the local

Table I. Baseline functions currently supported in SHMEM+ library

Function	SHMEM+ Call	Type	Purpose
Initialization	shmem_init	Setup	Initializes SHMEM library and other resources
Comm. Id	my_pe	Setup	Provides a unique ID for each process
Comm. size	num_pes	Setup	Provides number of PEs in the system
Finalize	shmem_finalize	Setup	De-allocates resources and gracefully terminates
Malloc	shmalloc	Setup	Allocates memory for shared variables
Get	shmem_int_g	P2P	Reads single element from a remote node
Put	shmem_int_p	P2P	Writes single element to a remote node
Get	shmem_getmem	P2P	Bulk read from a remote node
Put	shmem_putmem	P2P	Bulk write to a remote node
Quiet	shmem_quiet	Synch.	Waits for completion of outstanding puts
Barrier Sync.	shmem_barrier_all	Synch.	Synchronizes all the nodes

CPUs on the source and destination nodes. However, it might occasionally reduce the opportunities of overlapping intermediate steps of communication that exist in the application. SHMEM+ does not force developers to work at a particular level of abstraction. Instead it provides transfer functions which improve productivity along with functions which allow more detailed control over data transfers, for achieving higher performance. The choice of the data-transfers employed will often depend on the characteristics and structure of target application.

4.1 SHMEM+ Interface

Our design of SHMEM+ as described in this paper focuses on a subset of baseline functions selected from the entire API function set of SHMEM. In this paper, we discuss 11 baseline functions shown in Table I, which include five setup functions, four point-to-point messaging calls, and two synchronization routines. Some of these functions can be easily extended to support other SHMEM functions; such is the case for single-element and contiguous data-transfer routines. In this version of SHMEM+, we focus primarily on blocking communication. However, we also provide limited support for non-blocking communication as in the case for transfers between a CPU and its local FPGAs. More extensive support for non-blocking transfers is the focus of our ongoing research and future work. In addition, all of the various transfers are currently initiated by CPU devices which invoke SHMEM+ functions to transfer data between any two locations.

It is our objective to keep the interface of SHMEM+ consistent with previous SHMEM implementations. However, the functionality provided by SHMEM+ has been extended in various ways to incorporate support for FPGAs and provide multilevel-PGAS abstraction. Thus, SHMEM+ functions perform these extra tasks in addition to the ones performed by traditional SHMEM routines. For example, the *shmem_init* routine performs FPGA initialization (i.e. configuration of FPGA with the required hardware design) and FPGA memory-management operations, concomitant to initialization and management of CPU memory segments as performed by the traditional *shmem_init* function. The routines for data transfer (variations of *shmem_get* and *shmem_put*) perform exchanges between any two devices, such as two CPUs or between a CPU and an FPGA, etc. Based on the target memory address specified in the function, SHMEM+ identifies whether the requested

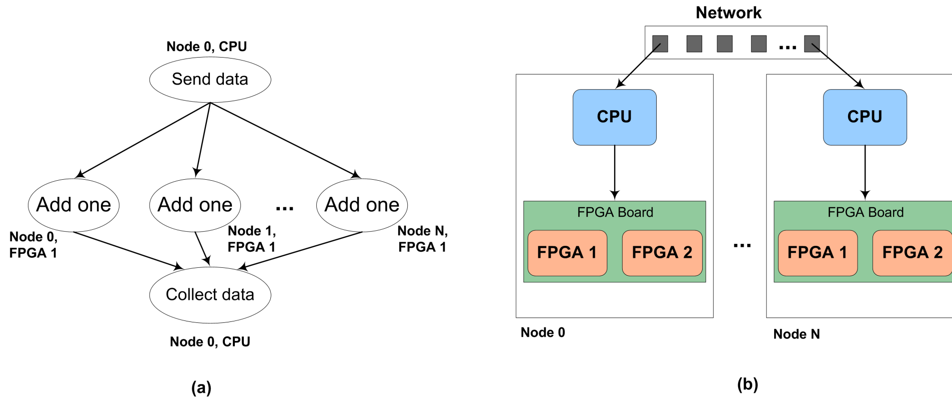

Fig. 7. (a) Task-graph of a multi-FPGA "add-one" application along with the desired mapping of tasks onto devices, (b) An abstract representation of the architecture of the target RC system.

data resides in CPU or FPGA memory and employs appropriate means of transferring the data. In addition, transfers to both remote and local FPGAs can be performed using the same interface, eliminating the need for multiple APIs. Without SHMEM+, application developers must decompose the algorithm in multiple stages, using the conventional SHMEM library for system-level decomposition and lower-level vendor APIs for distributing the functions across various PUs within a node, and then carefully orchestrate the communication through multiple libraries based on the location and type of the target device. The memory allocation routine (*shmalloc*), which allocates memory for data variables from the shared address space, has been modified to allow users to specify the affinity of any data to a particular memory block in the system. For example, a set of data that is operated upon by an FPGA can be specified to be allocated on FPGA memory which, as explained in Section 3, can be beneficial for application performance. The application developer conveys this information by specifying the "type" parameter (type = 0 for CPU memory, 1 for FPGA memory) in the *shmalloc* function call.

4.2 SHMEM+ Application Example

In order to develop a better understanding and appreciation for SHMEM+, we highlight the differences introduced by SHMEM+ in an application through an example. Figure 7a shows the task-graph of a multi-FPGA "add-one" application along with the desired mapping of each task onto a device (labeled alongside in the task graph). The task "Send data"which is mapped on the CPU of Node 0 sends input data to the "Add one" tasks (mapped on the first FPGA of each node) and collects the output data on completion of processing. The architecture of the target RC system for the application is presented in Figure 7b. Figure 8 lists the code snippets of the add-one application designed using (a) SHMEM+ and (b) a combination of CPU-only SHMEM and vendor-specific APIs. With SHMEM+ an application developer has the capability of transferring the input data to various FPGAs (both local and remote) using the same interface of SHMEM+. Traditionally, this transfer would have been achieved by distributing the data from CPU on

Send input data

Collect ouput data

```
int main(int argc, char* argv[])
{
...
...
shmem_init(&argc, &argv);

num_pe = num_pes(); //# of total tasks
me = my_pe(); // my task id

src_data = shmalloc(DATA_LEN*num_pe*sizeof(int),CPU);
res_data = shmalloc(DATA_LEN*num_pe*sizeof(int),CPU);
fpga_inp_data = shmalloc(DATA_LEN*sizeof(int),FPGA1);
fpga_out_data = shmalloc(DATA_LEN*sizeof(int),FPGA1);

//source data initialization

shmem_barrier_all(); //all nodes synchronize

//distribute the input data set to all the FPGAs
if(me==0)
  for(int i=0;i<num_pe;i++)
    shmem_putmem( fpga_inp_data, src_data+(i*DATA_LEN),
                  DATA_LEN*sizeof(int), i);

wait_for_fpga_done();//Wait for processing to complete

//Send the results to node 0
shmem_putmem( res_data+(me*DATA_LEN), fpga_out_data,
              DATA_LEN*sizeof(int), 0);

shmem_barrier_all();
print_result();

shmem_finalize();
return 0;
}
```

(a)

```
int main(int argc, char* argv[])
{
...
...
shmem_init(&argc, &argv);

num_pe = num_pes(); //# of total tasks
me = my_pe(); // my task id

src_data = (int *)shmalloc(DATA_LEN*num_pe*sizeof(int));
res_data = (int *)shmalloc(DATA_LEN*num_pe*sizeof(int));
fpga_inp_data = (int *)shmalloc(DATA_LEN*sizeof(int));
fpga_out_data = (int *)shmalloc(DATA_LEN*sizeof(int));

//source data initialization

shmem_barrier_all(); //all nodes synchronize

//distribute the input data set to all the FPGAs
if(me==0)
  for(int i=0;i<num_pe;i++)
    shmem_putmem( fpga_inp_data, src_data+(i*DATA_LEN),
                  DATA_LEN*sizeof(int), i);

shmem_barrier_all(); //Wait for data from node 0
fpga_write(fpga_inp_data); //send input data to
                           //local FPGA using Vendor API

wait_for_fpga_done();//Wait for processing to complete

//Send the results to node 0
fpga_read(fpga_out_data); //read output data from
                          //local FPGA using Vendor API
shmem_putmem( res_data+(me*DATA_LEN), fpga_out_data,
              DATA_LEN*sizeof(int), 0);

shmem_barrier_all();
print_result();

shmem_finalize();
return 0;
}
```

(b)

Fig. 8. Code snippets for a multi-FPGA add-one application designed using (a) SHMEM+, (b) SHMEM.

Node 0 to the CPUs on other nodes. All the nodes would then have to perform a synchronization operation to ensure the receipt of data before proceeding with a transfer to their local FPGAs. Although Figure 8b summarizes the transfer to the local FPGA using a single function call, the process is often less than trivial and non-uniform across different FPGA platforms. Even for this simplified example, it is easy to see the benefits of employing SHMEM+ over traditional methods of application design. More complex applications with more intricate communication patterns will benefit from larger reduction in program complexity and developer effort.

SHMEM+ provides application developers with a parallel-programming model that enables productive and portable design of scalable RC applications. There are a variety of factors that contribute towards improvement in developer productivity that are listed in Table II. Applications developed without using SHMEM+ exhibit higher conceptual complexity. Application developers are often forced to employ multiple libraries with varying APIs to incorporate communication amongst a cluster of host CPUs and to facilitate coordination between a host CPU and its local FPGAs. In addition, any communication with an FPGA on a remote node will have to be explicitly routed by the developer, through the host CPU on that node. The processing on the host CPU of the remote node will have to be interrupted to service this communication request, which further increases the complexity of developing the parallel program. With SHMEM+, a developer is oblivious to such details and exposed to a higher level of abstraction. Similarly, high-level SHMEM+ functions eliminate the need to explicitly perform various intermediate steps of communication, leading to a reduction in code size. Portability and scalability increase the

Table II. Major factors that contribute towards increased developer productivity when using SHMEM+

Factor	Description
Program complexity	High-level abstraction provided by SHMEM+ functions shields the application developer from various underlying details
Learning curve	Familiar APIs and programming model lead to reduced learning period when migrating to a new system
Source lines of code	Each SHMEM+ function can perform several intermediate steps of communication which eliminates the need for extra code and function calls
Application portability	Applications have longer life cycle as they can be executed on a variety of platforms
Scalability	Reduction in recurring developer effort to execute an application on systems of different sizes

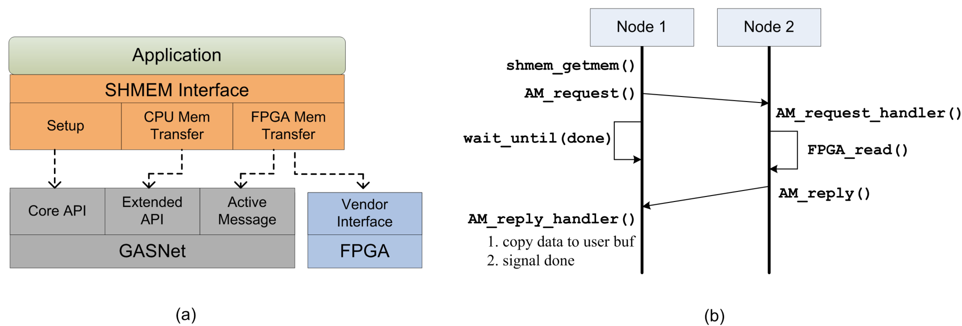

Fig. 9. (a) Software architecture of SHMEM+, (b) Data transfer example using Active Messages.

application lifespan and reduce the recurring cost that would have been involved without the use of a library like SHMEM+. A combination of the factors shown in Table II (and more) have a collective influence on the development time for an application. We will present a brief discussion about the total development hours spent in application development by our team for our case studies in Section 5.2.

4.3 Design of SHMEM+

Figure 9a illustrates the software architecture of SHMEM+. It makes use of GASNet's Core API, Extended API, and Active Message (AM) services. The setup functions, which perform memory allocation and other initialization tasks, employ the "Core API" services of GASNet. The data transfers to/from the CPU memory were built using the "Extended API," which provides direct support for high-level operations such as remote memory access. As a result, SHMEM+ functions that perform transfers between two CPUs can be implemented by simply providing wrappers around the underlying GASNet functions. Since transfers to/from FPGA memory are not directly supported by underlying GASNet functions, they were developed using the AM service in conjunction with FPGA interfaces that we created for our FPGA-platform (more details about our platform are provided in Section 5).

Figure 9b shows the sequence of steps involved in a transfer using Active Messages when a CPU requests data from a remote FPGA. The CPU on Node 1 initiates the transfer by calling the *shmem_getmem* function, which sends an AM request

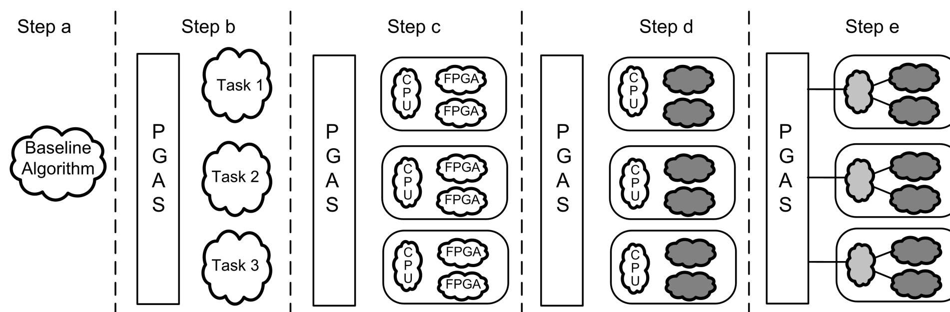

Fig. 10. Design methodology for application development using multilevel PGAS and SHMEM+.

to the CPU on remote node (Node 2). Upon receiving the AM request message, Node 2 invokes an AM request handler which reads the requested data from the local FPGA in a temporary buffer and sends an AM reply message containing the requested data to the initiating node. When Node 1 receives the reply message, it invokes a corresponding reply handler to copy the incoming data into the user-specified location. The message handlers shown in the figure employ *FPGA_read* and *FPGA_write* functions, which we developed using the FPGA-board vendor's API to communicate with FPGA memory. Due to overhead incurred by AM services and data access to/from the FPGA board, communication with an FPGA can result in slightly higher latency and lower bandwidth when compared to CPU transfers.

4.4 Design Methodology With SHMEM+

One of the important goals of multilevel PGAS and SHMEM+ is to provide developers with a framework for building large-scale RC applications using familiar techniques of parallel programming. The design methodology for building applications using SHMEM+ is described in Figure 10. A developer begins with a baseline algorithm of the application (step a). A parallel algorithm is obtained by system-level decomposition (step b) of the baseline into multiple tasks, each of which is assigned to a node in the target RC system. Various conventional techniques of decomposition can be employed during this stage, such as SPMD, pipelining, etc. Each task of the parallel algorithm is further decomposed into constituent functions which are distributed amongst the processing units in each node (step c). In the following step (step d), the developer describes the functions mapped on FPGAs as hardware engines using a hardware description language (HDL) or HLLs. Finally, the remainder of the functions are described in software to be mapped on CPUs (step e). The software also provides the FPGA with control signals required by the hardware engines developed in the previous step. The functions that are mapped on the CPUs employ SHMEM+ routines to access the PGAS in the system and perform synchronization operations. Although the current version of SHMEM+ only allows CPU-initiated transfers, the capability of FPGA-initiated transfers in future can provide numerous opportunities for innovative application design. The design flow described here is further exemplified through multiple case studies in Section 5.

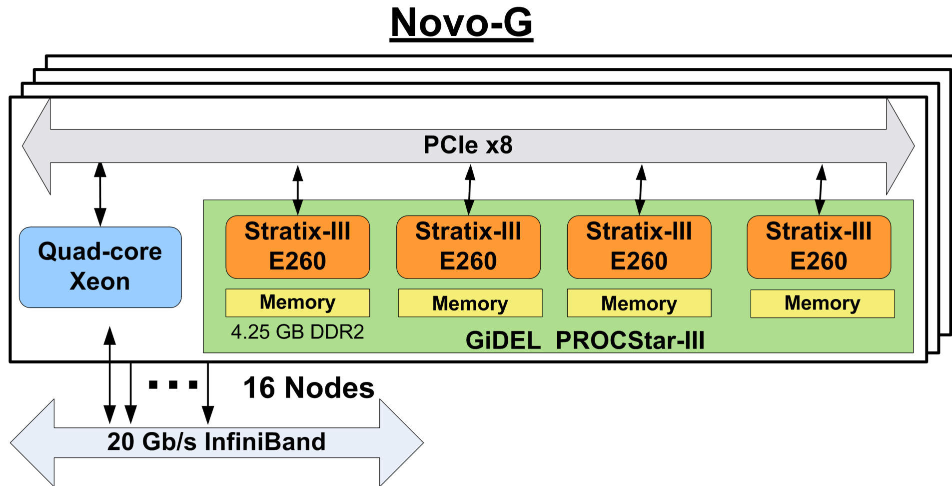

Fig. 11. System architecture of the Novo-G RC supercomputer

5. EXPERIMENTAL RESULTS

In this section, we present the performance obtained for various memory transfers with SHMEM+ and compare it against the performance obtained with the vendor-proprietary, CPU-only version of SHMEM provided by Quadrics for QsNet systems. We then present two case studies to illustrate the design methodology, and evaluate various advantages of application-design using SHMEM+.

To evaluate portability and scalability of applications designed using SHMEM+, we conducted our experiments on two different systems. Our first system (Mu cluster) consists of four Linux servers connected via QsNetII from Quadrics. Each server is comprised of an AMD 2GHz Opteron 246 processor and a tightly coupled set of four FPGA accelerators on a PROCStar-III PCIe board from GiDEL. The FPGA board features four Altera Stratix-III EP3SE260 FPGAs, each with two external DDR2 memory banks of 2GB and one bank of 256MB. The second system is our Novo-G RC supercomputer, which is comprised of 24 computer servers (of which only 16 were operational at the time of this research), each equipped with a Nehalem quad-core Xeon processor and a PROCStar-III board from GiDEL. The servers are connected via DDR InfiniBand. An architecture overview of Novo-G is provided in Figure 11. The SHMEM+ library was initially developed on the Mu cluster and later ported on Novo-G. Because Novo-G and Mu cluster use the same FPGA board, porting the SHMEM+ library to the Novo-G system was a straightforward process, just requiring an installation of GASNet on the new system. Ideally, any system that is supported by GASNet can be supported by SHMEM+ with ease, as in the case of Novo-G. However, in general, the process of porting SHMEM+ onto a new platform requires solving: (a) system-level issues and (b) FPGA platform-level issues. For supporting a new FPGA platform, the system architects are responsible for creating functions to interact with the FPGA board. Although the time to port SHMEM+ is largely dependent on the skills of a system developer and the complexity of the vendor APIs for the FPGA board,

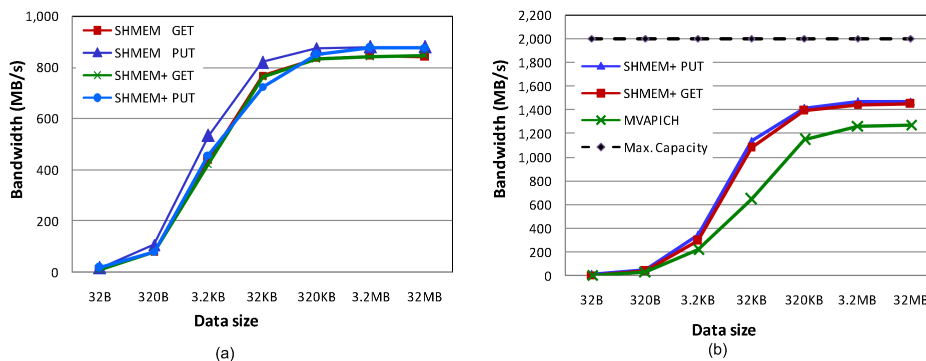

Fig. 12. Bandwidth of point-to-point routines using SHMEM+ and Quadrics SHMEM for transfers between two CPUs on (a) Mu cluster, (b) Novo-G. Note that in the performance results for Mu cluster, the line graphs of SHMEM GET and SHMEM+ GET overlap each other.

it should be in the order of a couple of weeks for an experienced system developer. Our current design of SHMEM+ library allows each of the four FPGAs on each node to support up to 2GB of shared memory which forms a part of the PGAS layer. The remainder of the memory is available to the FPGAs for storing local data.

5.1 Benchmarking Performance of Communication Routines

Figure 12 depicts performance of point-to-point communication in SHMEM+ for transfers between two CPUs on our two systems and compares it with the performance obtained by the SHMEM library from Quadrics on the Mu cluster. Bulk communication routines such as *shmem_getmem* and *shmem_putmem* attain a peak throughput of about 850MB/s on the Mu cluster. The bandwidth obtained with SHMEM+ calls, for transfers between two CPUs, is comparable to the proprietary version of SHMEM available from Quadrics for our Mu cluster. The SHMEM+ routines for these transfers benefit from direct support provided by GASNet and thus incur minimal overheads. Novo-G offers higher peak bandwidth (over 1400 MB/s, approx. 75% of the max. capacity of the network) for point-to-point transfers between two CPUs when compared to the Mu cluster due to the faster interconnect. Although the peak bandwidth obtained on Novo-G is higher than Mu cluster, the performance of GASNet for smaller message sizes is better on Mu cluster. Since there was no known implementation of SHMEM available for InfiniBand, we compared the performance of SHMEM+ with the performance of synchronous data-transfer functions present in MVAPICH [Network-Based Computing Laboratory] (an implementation of MPI over InfiniBand) which is a commonly used communication library. The graph indicates that SHMEM+ outperforms MVAPICH for large data transfers and offers a higher peak bandwidth.

Figure 13a shows performance of data transfers between a CPU and an FPGA using SHMEM+ routines on Novo-G. The “Local PUT” and “Local GET” labels represent the bandwidth of data transfers between a host CPU and its local FPGA on the same node. The bandwidth of such local transfers is specific to the particular

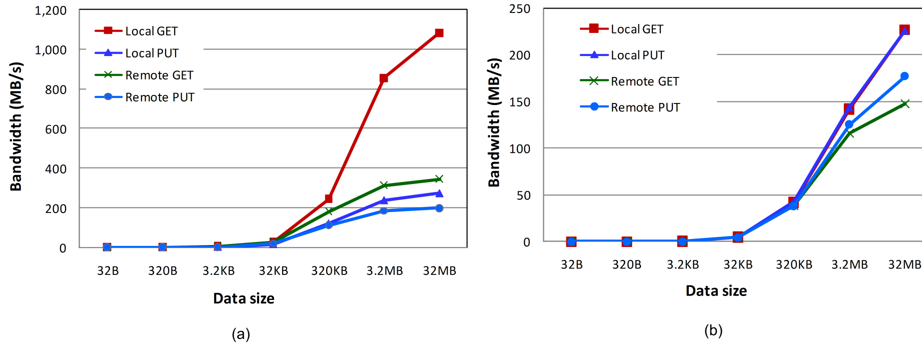

Fig. 13. Bandwidth of point-to-point routines on Novo-G using SHMEM+ for transfers between (a) CPU and FPGA, (b) Two FPGAs. The label 'Local' in the graphs represents transfers between devices which are on the same node, whereas the label 'Remote' represents the transfers between devices on separate nodes. Note that for transfers between two FPGAS, the line graphs of Local GET and Local PUT overlap each other.

Table III. End-to-end latency of transfers between various combinations of devices for traditional SHMEM on Mu cluster, and SHMEM+ on Mu cluster and Novo-G. The times are reported in microseconds for data transfer of 32 bytes.

Transfers between:	**SHMEM (Mu)**	**SHMEM+ (Mu)**	**SHMEM+ (Novo-G)**
Two remote CPUs	3.87 μs	3.76 μs	7.96 μs
CPU & local FPGA	205.49 μs	213.60 μs	644.92 μs
CPU & remote FPGA	211.67 μs	219.49 μs	664.70 μs
Two remote FPGAs	427.09 μs	425.27 μs	1292.22 μs

FPGA board and depends upon a variety of factors associated with interconnect(s) between CPU and FPGA, efficiency of the communication controller on the board, etc. Many RC systems offer a higher bandwidth for read operation from an FPGA (FPGA to CPU) when compared to write operation (CPU to FPGA). Similarly, our system yields a peak bandwidth of approximately 275MB/s for local put operations (CPU to FPGA) and approximately 1000MB/s for local get operations (FPGA to CPU). The "Remote PUT" and "Remote GET" labels represent the bandwidth of data transfers between a CPU and an FPGA on a different node. As expected, the bandwidth for such transfers is observed to be lower than the bandwidth attained for local transfers. Figure 13b shows performance of transfers between two FPGAs. The labels "Local" indicate the two devices are on the same node whereas "Remote" represent transfers between devices on different nodes. Since transfers between any two FPGAs require two transfers over the PCIe bus internally (a read operation and a write operation to FPGA), bandwidth obtained for such transfers is lower than the bandwidth of transfers between a CPU and an FPGA and is limited by the performance of the write operation. As a result the peak bandwidth obtained for direct transfers between two FPGAs is approximately 230MB/s.

Table III reports the end-to-end latency (EEL) observed for transfers between various combinations of devices. The smallest data size for transfers in our experiments was restricted to 32 bytes by the requirements of the FPGA board. The second and third column in the table compare the latencies observed for traditional

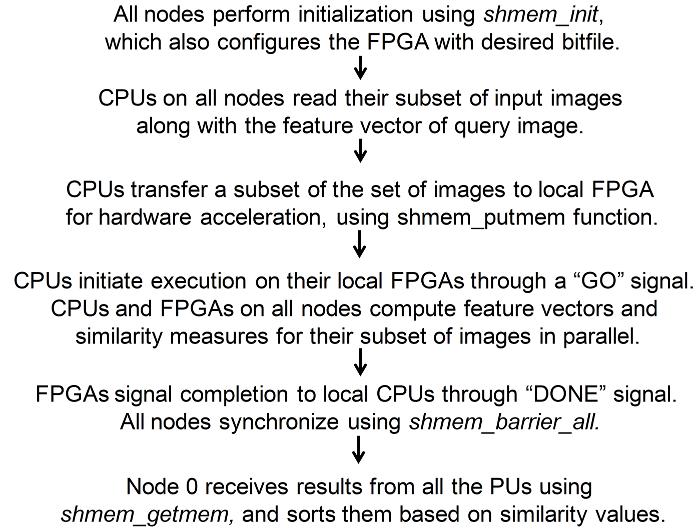

Fig. 14. Processing steps involved in parallel algorithm for CBIR using SHMEM+.

SHMEM with those observed for SHMEM+ on Mu cluster. The differences between the two are less than 5% for all the cases. The table also lists the EEL observed for transfers on Novo-G. The latency for data transfers using GASNet on Novo-G is higher than on Mu cluster, which concurs with the performance graphs presented earlier. From results presented in this section, it can be observed that performance of SHMEM+ compares well with conventional SHMEM for transfers between two CPUs, and SHMEM+ performs reasonably well for communication with an FPGA.

5.2 Case Study 1: Content-based Image Retrieval

Content-Based Image Retrieval (CBIR) is a common application in computer vision and consists of searching a large database of digital images for the ones that are visually similar to a given query image, where the search is based on contents of the image. The content in this context can be one of the several features present in the image, such as colors, shapes, textures, or any other information that can be derived from the image. CBIR has been widely adopted in many domains such as biomedicine, military, commerce, education, and Web image classification and searching. Each image in a CBIR system is represented by a feature vector, which is based on characteristics of the image as cited above. Similarity between a query image and the set of images in the database is determined by measuring similarity between their feature vectors. The processes of determining the feature vector and analyzing images for similarities are often the most computationally intensive stages in any CBIR system [Guyon et al. 2006]. There are various forms of parallelism available in the application that can be exploited by RC systems to accelerate the search process [Skarpathiotis and Dimond 2004].

Our implementation presented in this paper employs a technique based on auto-correlogram of color components [Huang et al. 1997], where the feature vector is based on color information in the image. A correlogram of an image corresponds to

a table where the rows are indexed by color pairs (c_i, c_j) such that the d-th column in row (c_i, c_j) stores the probability of finding a pixel of color c_j at a distance d from a pixel of color c_i in the image. For the case of auto-correlogram, the table only consists of rows where $c_i = c_j$. In this paper, we use a modified version of auto-correlogram, which stores an absolute count of the occurrences of a pixel of color c_i instead of the probability of such an event. Similarity between two images is determined by calculating the sum of absolute differences between their feature vectors.

The processing steps involved in the parallel algorithm employed in our experiments are shown in Figure 14. We employed the design-flow described in Section 4 to derive this algorithm as follows:

Step a. The serial algorithm iterates over the set of images in the database to calculate their feature vector and determine their similarity with the query image. Once all the images in the database have been processed, the results are sorted in decreasing order of their similarity.

Step b. Parallel algorithm is obtained by distributing the set of images in the database over the processing nodes in the system.

Step c. The set of images is further partitioned amongst the processing units within each node. The number of images to be processed on each FPGA and CPU was determined based on their processing capability. By exploiting fine-grain parallelism available in algorithm, FPGAs are able to process images at a faster rate than the CPUs and are assigned a larger subset of images.

Step d. Using VHDL, we developed a hardware engine for FPGAs, which iterates over its assigned set of input images to compute their feature vectors and evaluate their similarity to the specified query image.

Step e. We developed software code for processing a subset of images on the CPU, providing each FPGA with control signals to initiate processing and wait for completion, and transferring results from all processing units in the system to root node (node 0) at completion.

In addition to software parallelism described in the algorithm above, our hardware design for each FPGA instantiates multiple computational kernels that operate on five images in parallel. Figure 15 compares the execution time and speedup versus a serial software baseline for different implementations of a CBIR algorithm on the Mu cluster. Our experiments were conducted for an image size of 128×128, with the search database consisting of approximately 2800 images. Advantages of using FPGA devices are evident through faster execution times for RC-based implementations over software-only solutions. The FPGAs were able to process images at a much faster rate than the CPUs leading to over $30\times$ speedup with four nodes when employing FPGA devices. Figure 15 also compares the performance of the algorithm implemented using SHMEM+ with solution implemented using a combination of Quadrics SHMEM (CPU-only) library and platform-specific APIs for interaction with FPGAs. It is evident that the application developed using SHMEM+ incurs minimal overhead compared to traditional techniques of development where expert developers have access to vendor APIs.

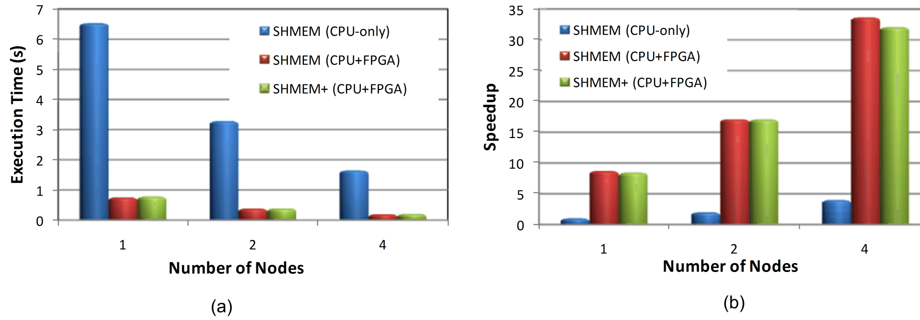

Fig. 15. Performance comparison of different implementations of parallel CBIR application on Mu cluster. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. (a) Execution time of different designs, (b) Speedup obtained by different designs when compared to a serial software baseline running on a single processor. The experiments were conducted for a search database consisting of 2800 images, each of size 128×128.

More importantly, SHMEM+ provides application developers with a parallel-programming model that enables productive and portable design of scalable RC applications. A combination of the factors shown in Table II (and more) have a collective influence on the development time for an application. Although a comprehensive analysis of impact of each of those factors is beyond the scope of this work, to understand the productivity gains of SHMEM+, we present a brief discussion about the total development hours spent in application development by our team.

Table IV compares the development hours spent by our team during various stages of application design employing (a) traditional techniques of implementation and (b) SHMEM+ separately. Although the numbers cited are specific to our team personnel, we believe they are a fair estimate of improvements expected from SHMEM+. For the numbers cited in Table IV, we assume the developer has experience in parallel programming and in creating FPGA designs using VHDL. In addition, we assume the developer is new to the RC system and hence has to undergo a learning process to familiarize with the platform, which is often the case when porting applications to a new system. The rows in Table IV report the time spent (in terms of 8-hour work days) in various phases of application development which required significant amount of the time and effort. The time spent in each activity also includes the hours spent for debugging in that phase, wherever applicable. Since SHMEM+ does not modify the process of developing hardware cores for FPGAs, the time required for FPGA-core development (first two rows of the table) remains unaffected for both techniques, but has been included here for completeness. It should be noted that we employed HDL for developing our hardware-cores and further reductions in effort can be obtained by employing HLLs, if they are supported by the target platform. The time spent in parallel-software development includes the amount of time a developer spends in familiarizing with the platform-specific API, learning the SHMEM API, and finally designing the parallel application using these APIs. When using SHMEM+, a developer employs the SHMEM+ interface for interacting with FPGA memory and hence has to spend

Table IV. Development hours spent in developing CBIR application. Time is reported in terms of 8-hour work days.

Development Phase		Traditional SHMEM	SHMEM+
FPGA-core development	Learning platform-specific wrappers	10 days	10 days
	App-core design	15 days	15 days
Parallel-software development	Platform-specific API learning period	5 days	2 days
	SHMEM API learning period	5 days	5 days
	Parallel-application design	10 days	7 days
	Total software development time	**20 days**	**14 days**

less time understanding only a subset of platform-specific APIs which are required for sending (or receiving) control signals (third row in the table). By contrast, the learning period involved for the SHMEM API remains unaffected as both the techniques expose the developer to a similar interface. Due to a higher level of abstraction provided by SHMEM+, a reduction in the time for designing the parallel application was observed as indicated by the fifth row in the table. As shown in the sixth row, an overall reduction in time and effort of about 30% was obtained for the total time spent in various phases of parallel-software development. Such an improvement could translate to significant savings in the development hours and money spent on the design of a complex application. For example, an application that required 10 weeks of development time could now be completed in just 7 weeks. Applications with more complex communication patterns are expected to have higher gains in productivity.

Since SHMEM+ applications do not employ any vendor-specific APIs for interaction with FPGAs, applications developed using SHMEM+ are highly portable. As long as the SHMEM+ library can be supported on a target RC system, any application designed with SHMEM+ can execute on it without requiring changes to the application source code. We evaluated the portability by migrating the CBIR application from our Mu cluster to Novo-G. This process did not require any modification to software and hardware source code. A simple re-compilation of the software was required to obtain an executable for the new system. It should be noted that for a system with a different FPGA board, some modifications will be needed for the hardware designs. Figure 16 shows the performance of the CBIR application scaling up to 16 nodes on Novo-G. We expanded our search database to include approximately 22,000 images on this larger system. The graphs in Figure 16 compare the performance of a design developed using SHMEM+ with a design based on traditional SHMEM. It is evident that designs developed using SHMEM+ continue to offer comparable performance to designs based on traditional SHMEM for larger system sizes. The minor variation in performance of the design created using SHMEM+ when compared to the one created with SHMEM is due to the difference in the communication mechanism used to gather the results on the root node in the last processing step. As with any form of high-level abstraction, a tradeoff exists between productivity and performance. For our application design using SHMEM+, the ability to use direct transfer to the remote FPGAs eliminates the opportunity to overlap the intermediate steps of communication for this design. However, SHMEM+ does not force developers to work at a particular level of abstraction. Instead it provides application developers with multiple options to meet

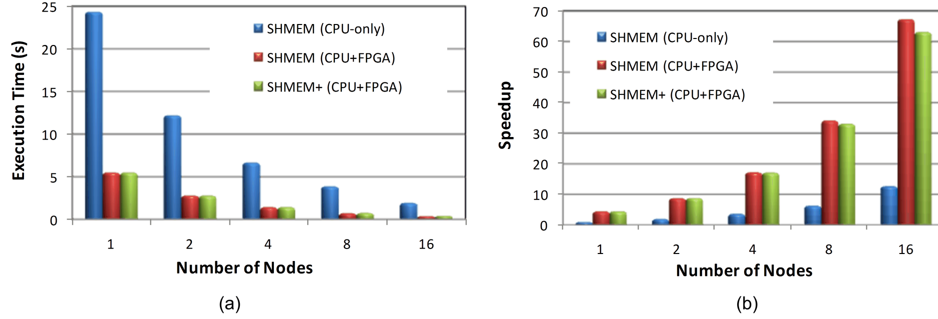

Fig. 16. Performance comparison of different implementations of parallel CBIR application on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. (a) Execution time of different designs, (b) Speedup obtained by different designs when compared to a serial software baseline running on a single processor. The experiments were conducted for a search database consisting of 22,000 images, each of size 128×128.

the demands of the application. Although the performance penalty incurred by the designs created using SHMEM+ is minimal, we made a minor modification to the implementation of the quad-FPGA designs to eliminate this penalty as discussed in the following paragraphs.

Parallel algorithms employing multiple FPGAs on each node exhibit more complex communication patterns and often require increased developer effort to obtain an efficient implementation. SHMEM+ has the ability to support multiple FPGAs on each FPGA board using the same interface, which further simplifies the development process and yields additional improvement in productivity. Figure 17 compares the performance of different designs employing all of the four FPGAs on each node of Novo-G. The algorithm designed using SHMEM+ was modified slightly to optimize the collection of results at the end on root node (Node 0). Instead of the root node using a “GET” routine to receive results from all the FPGAs, each processing node uses a “PUT” function to transfer the results from each of its local FPGA to the root node. The optimization allows the designs created with SHMEM+ to exhibit excellent scaling behavior and minimal overheads when compared to the designs created using a combination of CPU-only SHMEM and vendor APIs.

5.3 Case Study 2: Two-dimensional FFT

As our next case study of parallel application, a two-dimensional FFT was chosen because of its emphasis on a more complex communication pattern and its relevance in a variety of application domains such as medical imaging systems, Synthetic Aperture RADAR (SAR) systems, and image processing [Brigham 1988; Gonzales and Woods 2002; Uzun et al. 2005]. The heavy computation demands of Fourier transform [Cooley and Tukey 1965] poses tremendous pressure on the capabilities of computation platforms in most real-world applications, as a result of which several researchers have explored FPGA implementations for the same [Shirazi et al. 1995; Underwood et al. 2001]. A 2-D FFT operation on an image is performed by decomposing it into a series of 1-D FFT over the rows of the image, followed by a

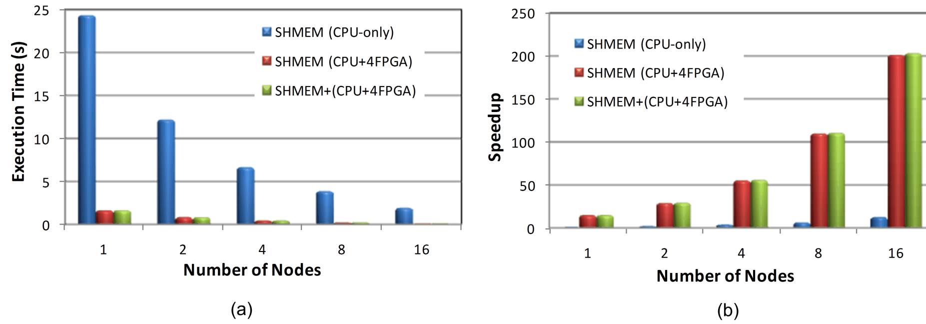

Fig. 17. Performance comparison of different implementations of parallel CBIR application on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and four FPGAs on each node. (a) Execution time of different designs, (b) Speedup obtained by different designs when compared to a serial software baseline running on a single processor. The experiments were conducted for a search database consisting of 22000 images each of size 128×128. Designs based on SHMEM+ continue to offer minimal overheads when compared to tradition techniques of application development

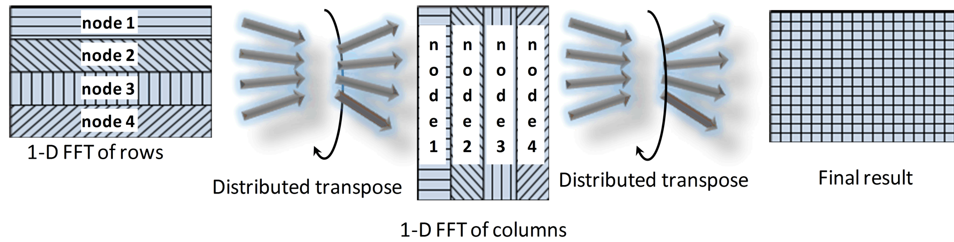

Fig. 18. Abstract representation of the processing steps involved in a parallel two-dimensional FFT algorithm.

series of Fourier transforms over the columns.

Our parallel implementation of 2-D FFT algorithm distributes rows of the input image across the computational nodes which perform a 1-D FFT over their assigned subset of rows as shown in Figure 18. A corner-turn (distributed transpose), which involves all-to-all communication between the processing nodes, is required to re-distribute the data across all the nodes. The nodes then compute 1-D FFT over the columns of the image. Another corner turn is required to re-organize the data and recover the transformed output image. Following the design flow described earlier we derive our implementation as follows:

Step a. The serial algorithm computes the 2-D FFT of the image by performing a series of 1-D FFT over the rows followed by 1-D FFT over the columns of the image.

Step b. Our parallel algorithm is obtained by using block decomposition to distribute a subset of rows and columns to be transformed on each node. An all-to-all communication is required to re-distribute the data between the two stages of 1-D FFTs.

Table V. Development hours spent for developing Two-dimensional FFT application

Development Phase		Traditional SHMEM	SHMEM+
FPGA-core development	Learning platform-specific wrappers	15 days	15 days
	App-core design	15 days	15 days
Parallel-software development	Platform-specific API learning period	5 days	2 days
	Parallel-application design	12 days	10 days
	Total software development time	**17 days**	**12 days**

Step c. The FPGA on each node is able to perform 1-D FFT operations faster than the CPU and is hence assigned to transform the assigned set of rows and columns. Since the CPUs are more efficient in re-organizing the data in their local memory than FPGAs, they are assigned to perform the corner turn.

Step d. Using VHDL, we developed a hardware engine for FPGAs, to perform a series of 1-D FFT over its assigned set of input data. The FPGA waits for control signal from the CPU to begin processing and indicates completion of transforms through another control signal.

Step e. Software code on the CPU is responsible for transferring the input data to the FPGAs and reading the transformed output once FPGA completes processing. The software code also performs an all-to-all communication to complete the corner-turn. In addition, it also provides the control signal required by the FPGAs.

Figure 19 compares the execution time and speedup versus a serial software baseline for different implementations of a parallel 2-D FFT algorithm on Novo-G. Our experiments were conducted for an 8k×8k image. Similar to our first case study, designs for 2-D FFT implemented using SHMEM+ yield performance which is comparable to designs implemented using a combination of traditional, CPU-only SHMEM library and platform-specific APIs. The minor difference in the performance of both of these designs is within reasonable limits of experimental error. A comparison of the hours spent in developing 2-D FFT algorithm using both of these techniques is presented in Table V. For this case study, we assume a parallel application developer is familiar with the SHMEM API and does not have to spend any effort/time learning it. However, since platform-specific learning is a common occurrence each time the application is ported to a new platform, we retain the learning period for the platform-specific API. Our estimates indicate an improvement of approximately 25% in developer productivity. Portability experiments were also conducted for the second case study. Since most of the results and inferences from these experiments were consistent with the first case study, they are not repeated here.

6. CONCLUSIONS AND FUTURE WORK

The lack of integrated, system-wide, parallel programming models has limited current RC applications to small systems sizes. To realize the full potential of reconfigurable HPC systems, parallel programming models and languages that are suited to such systems are critical yet lacking. In this paper, we presented a parallel-programming model and a communication library for scalable, heterogeneous, reconfigurable systems. The multilevel-PGAS model proposed in this paper is able to capture key characteristics of RC systems, such as different levels of memory hierarchy and differences in the execution model of heterogeneous devices present

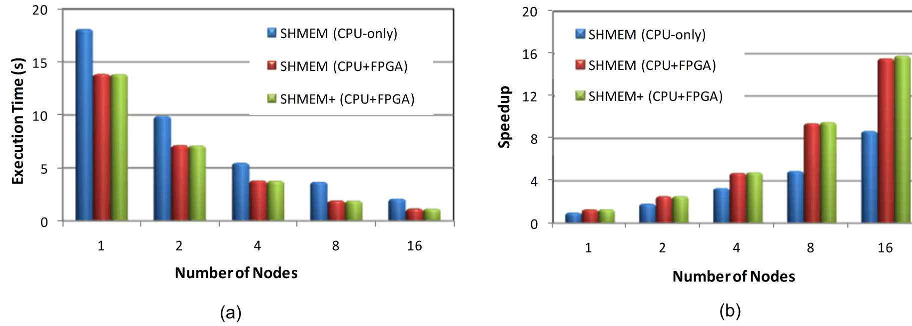

Fig. 19. Performance comparison of different implementations of parallel 2-D FFT algorithm on Novo-G. Software designs involve only CPU devices whereas RC designs involve both CPU and FPGAs on each node. (a) Execution time of different designs, (b) Speedup obtained by different designs when compared to serial software baseline running on a single processor. The experiments were conducted for an $8k \times 8k$ image.

in the system. The existence of such a programming model will enable productive development of scalable, parallel applications for reconfigurable HPC systems.

Using the multilevel-PGAS programming model, we extend the existing SHMEM library to SHMEM+, the first known version of the library that enables designers to create scalable applications that execute over a mix of microprocessors and FPGAs. SHMEM+ offers developers of RC applications a high-level of abstraction that allows them to facilitate complex communication in application between heterogeneous devices, while providing high productivity and performance. Results from our experiments and case studies demonstrate that performance offered by SHMEM+ is comparable to the existing vendor-proprietary version of SHMEM. Our case studies showcase the simplified design process involved with SHMEM+ for developing scalable RC applications, which is very similar to traditional methods for development of parallel applications. More importantly, the higher level of abstraction provided by SHMEM+ leads to significant improvement in productivity without sacrificing performance significantly. Although it is difficult to quantify the productivity gains, our case studies demonstrate an average improvement in productivity of about 30%. In addition, by hiding the details of vendor-specific FPGA communication from developers, SHMEM+ creates highly portable applications.

Directions for future work include various expansions to the SHMEM+ library and its communication capabilities. We plan to enhance the communication model by investigating mechanisms for FPGA-initiated transfers, which are often not supported by FPGA platforms and hence will have to be supported through a virtual abstraction. Nevertheless, the capabilities offered by such transfers provide opportunities for innovative application design. The multilevel PGAS programming model and the SHMEM+ library can be easily extended and applied to other systems besides HPC, such as high-performance embedded systems (HPEC) which employ a variety of embedded processors and accelerators, including GPUs and many-core processors. The application of SHMEM+ to such systems and its potential impact will be explored in future research. In addition, we intend to inves-

tigate, develop, and evaluate tools to support performance analysis for SHMEM+ applications.

ACKNOWLEDGMENTS

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. This work was also supported by the United States Department of Defense and used resources of the Extreme-Scale Systems Center at Oak Ridge National Laboratory. The authors gratefully acknowledge equipment and tools from Altera and GiDEL. We would also like to thank Rafael Garcia, former M.S. student in our lab, for his contributions to this work.

REFERENCES

Aggarwal, V., Garcia, R., Stitt, G., George, A., and Lam, H. 2009. SCF: a device- and language-independent task coordination framework for reconfigurable, heterogeneous systems. In *HPRCTA '09: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, New York, NY, USA, 19–28.

Aggarwal, V., George, A., Yalamanchili, K., Yoon, C., Lam, H., and Stitt, G. 2009. Bridging parallel and reconfigurable computing with multilevel PGAS and SHMEM+. In *HPRCTA '09: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, New York, NY, USA, 47–54.

APGAS 2009. *Workshop on Asynchrony in the PGAS Programming Model*. Available at http://research.ihost.com/apgas09/.

Bonachea, D. and Jeong, J. Spring 2002. GASNet: A portable high-performance communication layer for global address-space languages. CS258 Parallel Computer Architecture Project.

Brigham, E. O. 1988. *The Fast Fourier Transform and its Application*. Prentice Hall.

Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., and Warren, K. 1999. Introduction to UPC and language specification. Tech. rep., University of California-Berkeley, Berkeley, CA, USA.

Cooley, J. W. and Tukey, J. W. 1965. An algorithm for the machine computation of the complex fourier series. *Mathematics of Computation 19*, 297–301.

Cray T3ETM Fortran Optimization Guide - 004-2518-002. SHMEM. http://docs.cray.com/books/004-2518-002/html-004-2518-002/z826920364dep.html.

Darema, F. 2001. The SPMD model: past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, London, UK.

El-Ghazawi, T., Serres, O., Bahra, S., Huang, M., and El-Araby, E. 2008. Parallel programming of high-performance reconfigurable computing systems with Unified Parallel C. In *Proc. of Reconfigurable Systems Summer Institute*. Urbana, Illinois.

El-Ghazawi, T. A., Carlson, W. W., and Draper, J. M. 2001. UPC language specifications v1.0. http://upc.gwu.edu/docs/upc_spec_1.1.1.pdf.

Farreras, M., Marjanovic, V., Ayguade, E., and Labarta, J. 1997. Gaining asynchrony by using hybrid UPC/SMPSs. In *Workshop on Asynchrony in the PGAS Programming Model*. Yorktown Heights, NY, USA.

Gonzales, R. and Woods, R. E. 2002. *Digital Image Processing*. Addison-Wesley.

Guyon, I., Gunn, S., Nikravesh, M., and Zadeh, L. 2006. *Feature Extraction, Foundations and Applications*. Springer.

Huang, J., Kumar, S., Mitra, M., Zhu, W.-J., and Zabih, R. 1997. Image indexing using color correlograms. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*. 762–768.

MPI. MPI standard. http://www.mcs.anl.gov/research/projects/mpi/.

Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu.

Nishtala, R., Hargrove, P. H., Bonachea, D. O., and Yelick, K. A. 2009. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *IEEE International Parallel & Distributed Processing Symposium*. Rome, Italy, 1–12.

Numrich, R. W. and Reid, J. K. 1998. Co-Array Fortran for parallel programming.

OpenMP. The OpenMP API specification for parallel programming. http://openmp.org/wp/.

Saldana, M., Patel, A., Madill, C., Nunes, D., Danyao, W., Styles, H., Putnam, A., Wittig, R., and Chow, P. 2008. MPI as an abstraction for software-hardware interaction for HPRCs. In *HPRCTA '08: Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, New York, NY, USA.

SGI. Introduction to the SHMEM programming model. http://docs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=man&fname=/usr/share/catman/man3/intro_shmem.3.html&srch=intro_shmem.

Shih, K., Balachandran, A., Nagarajan, K., Holland, B., Slatton, C., and George, A. 2008. Fast real-time LIDAR processing on FPGAs. In *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms*. Las Vegas, NV, USA.

Shirazi, N., Athanas, P. M., and Abbott, A. L. 1995. *Field Programmable Logic and Application*. Springer Berlin, Chapter Implementation of a 2-D fast Fourier transform on an FPGA-based custom computing machine, 282–292.

Skarpathiotis, C. and Dimond, K. 2004. *Field Programmable Logic and Application*. Springer Berlin, Chapter A Hardware Implementation of a Content Based Image Retrieval Algorithm, 1165–1167.

Storaasli, O. 2008. Accelerating senome sequencing 100-1000X with FPGAs. In *Many-core and Reconfigurable Supercomputing Conference (MRSC)*. The Queen's University of Belfast, Northern Ireland.

Underwood, K. D., Sass, R. R., and Walter B. Ligon, I. 2001. Acceleration of a 2d-fft on an adaptable computing cluster. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA, 180–189.

Uzun, I., Amira, A., and Bouridane, A. 2005. FPGA implementations of fast Fourier transforms for real-time signal and image processing. *Vision, Image and Signal Processing, IEE Proceedings 152,* 3 (June), 283 – 296.

Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. 1998. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, New York, NY 10036, USA.

Received 20??; November 20??; accepted January 20??