A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era

Article *in* IEEE Transactions on Parallel and Distributed Systems · June 2012

DOI: 10.1109/TPDS.2011.308

3 authors:

Javier Diaz-Montes
Rutgers, The State University of New Jersey
65 PUBLICATIONS **470** CITATIONS

SEE PROFILE

Camelia Muñoz-Caro
University of Castilla-La Mancha
107 PUBLICATIONS **1,017** CITATIONS

SEE PROFILE

Alfonso Niño
University of Castilla-La Mancha
111 PUBLICATIONS **1,037** CITATIONS

SEE PROFILE

A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era

Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño

Abstract—In this work, we present a survey of the different parallel programming models and tools available today with special consideration to their suitability for high-performance computing. Thus, we review the shared and distributed memory approaches, as well as the current heterogeneous parallel programming model. In addition, we analyze how the partitioned global address space (PGAS) and hybrid parallel programming models are used to combine the advantages of shared and distributed memory systems. The work is completed by considering languages with specific parallel support and the distributed programming paradigm. In all cases, we present characteristics, strengths, and weaknesses. The study shows that the availability of multi-core CPUs has given new impulse to the shared memory parallel programming approach. In addition, we find that hybrid parallel programming is the current way of harnessing the capabilities of computer clusters with multi-core nodes. On the other hand, heterogeneous programming is found to be an increasingly popular paradigm, as a consequence of the availability of multi-core CPUs+GPUs systems. The use of open industry standards like OpenMP, MPI, or OpenCL, as opposed to proprietary solutions, seems to be the way to uniformize and extend the use of parallel programming models.

Index Terms—Parallelism and concurrency, distributed programming, heterogeneous (hybrid) systems.

1 Introduction

Microprocessors based on a single processing unit (CPU) drove performance increases and cost reductions in computer applications for more than two decades. However, this process reached a limit around 2003 due to heat dissipation and energy consumption issues [1]. These problems have limited the increase of CPU clock frequencies and the number of tasks that can be performed within each clock period. The solution adopted by processor developers was to switch to a model where the microprocessor has multiple processing units known as cores [2]. Nowadays, we can speak of two approaches [2]. The first, multi-core approach, integrates a few cores (currently between two and ten) into a single microprocessor, seeking to keep the execution speed of sequential programs. Actual laptops and desktops incorporate this kind of processor. The second, many-core approach uses a large number of cores (currently as many as several hundred) and is specially oriented to the execution throughput of parallel programs. This approach is exemplified by the Graphical Processing Units (GPUs) available today. Thus, parallel computers are not longer expensive and elitist devices, but commodity machines we find everywhere. Clearly, this change of paradigm has had (and will have) a huge impact on the software developing community [3].

- *J. Diaz is with the Pervasive Technology Institute, Indiana University, 2719 East Tenth Street, Bloomington, IN 47408. E-mail: javidiaz@indiana.edu.*
- *C. Muñoz-Caro and A. Niño are with the Grupo SciCom, Departamento de Tecnologias y Sistemas de Informacion, Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13004 Ciudad Real, Spain. E-mail: {camelia.munoz, alfonso.nino}@uclm.es.*

Manuscript received 22 Apr. 2011; revised 7 Dec. 2011; accepted 8 Dec. 2011; published online 28 Dec. 2011.
Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-04-0242.
Digital Object Identifier no. 10.1109/TPDS.2011.308.

Most of the software applications are developed following the sequential execution model, which is naturally implemented on traditional single-core microprocessors. Therefore, each new, more efficient, generation of single-core processors translates into a performance increase of the available sequential applications. However, the current stalling of clock frequencies prevents further performance improvements. In this sense, it has been said that "sequential programming is dead" [4], [5]. Thus, in the present scenario we cannot rely on more efficient cores to improve performance but in the appropriate coordinate use of several cores, i.e., in concurrency. So, the applications that can benefit from performance increases with each generation of new multi-core and many-core processors are the parallel ones. This new interest in parallel program development has been called the "concurrency revolution" [3]. Therefore, parallel programming, once almost relegated to the High Performance Computing community (HPC), is taken a new star role on the stage.

Parallel computing can increase the applications performance by executing them on multiple processors. Unfortunately, the scaling of application performance has not matched the scaling of peak speed, and the programming burden continues to be important. This is particularly problematic because the vision of seamless scalability needs the applications to scale automatically with the number of processors. However, for this to happen, the applications have to be programmed to exploit parallelism in the most efficient way. Thus, the responsibility for achieving the vision of scalable parallelism falls on the applications developer [6].

In this sense, there are two main approaches to parallelize applications: autoparallelization and parallel programming [7]. They differ in the achievable application performance and ease of parallelization. In the first case, the sequential programs are automatically parallelized using

ILP (instruction level parallelism) or parallel compilers. Thus, the main advantage is that existing applications just need to be recompiled with a parallel compiler, without modifications. However, due to the complexity of automatically transforming sequential algorithms into parallel ones, the amount of parallelism reached using this approach is low. On the other hand, in the parallel programming approach, the applications are specifically developed to exploit parallelism. Therefore, developing a parallel application involves the partitioning of the workload into tasks, and the mapping of the tasks into workers (i.e., the computers where the tasks will be processed). In general, parallel programming obtains a higher performance than autoparallelization but at the expense of more parallelization efforts. Fortunately, there are some typical kinds of parallelism in computer programs such as task, data, recursive, and pipelined parallelism [8], [9], [10]. In addition, much literature is available about the suitability of algorithms for parallel execution [11], [12] and about the design of parallel programs [10], [13], [14], [15]. From the design point of view, different patterns for exploiting parallelism have been proposed [8], [10]. A pattern is a strategy for solving recurring problems in a given field. In addition, the patterns can be organized as part of a pattern language, allowing the user to use the patterns to build complex systems. This approach applied to parallel programming is presented in [10]. Here, the pattern language is organized in four design spaces or phases: finding concurrency, algorithm structure, supporting structures, and implementation mechanisms. A total of 19 design patterns are recognized and organized around the first three phases. In particular, four patterns corresponding to the supporting structures phase can be related to the different parallel programming models [10]. These are: Single Program Multiple data (SPMD, where the same program is executed several times with different data), Master/Worker (where a master process sets up a pool of worker processes and a bag of tasks), loop parallelism (where different iterations of one or more loops are executed concurrently), and fork/join (where a main process forks off several other processes that execute concurrently until they finally join in a single process again).

Parallel systems, or architectures, fall into two broad categories: shared memory and distributed memory [8]. In shared memory architectures we have a single memory address space accessible to all the processors. Shared memory machines have existed for a long time in the servers and high-end workstations segment. However, at present, common desktop machines fall into this category since in multi-core processors all the cores share the main memory. On the other hand, in distributed memory architectures there is not global address space. Each processor owns its own memory. This is a popular architectural model encountered in networked or distributed environments such as clusters or Grids of computers. Of course, hybrid shared-distributed memory systems can be built.

The conventional parallel programming practice involves a pure shared memory model [8], usually using the OpenMP API [16], in shared memory architectures, or a pure message passing model [8], using the MPI API [17], on distributed memory systems. The largest and fastest computers today employ both shared and distributed memory architectures. This provides flexibility when tuning the parallelism in the programs to generate maximum efficiency and an appropriate balance of the computational and communication loads. In addition, the availability of General Purpose computation on GPUs (GPGPUs) in actual multi-core systems has lead to the Heterogeneous Parallel Programming (HPP) model. HPP seeks to harness the capabilities of multi-core CPUs and many-core GPUs. Accordingly to all theses hybrid architectures, different parallel programming models can be mixed in what is called hybrid parallel programming. A wise implementation of hybrid parallel programs can generate massive speedups in the otherwise pure MPI or pure OpenMP implementations [18]. The same can be applied to hybrid programming involving GPUs and distributed architectures [19], [20].

In this paper, we review the parallel programming models with especial consideration of their suitability for High Performance Computing applications. In addition, we consider the associated programming tools. Thus, in Section 2 we present a classification of parallel programming models in use today. Sections 3 to 8 review the different models presented in Section 2. Finally, in Section 9 we collect the conclusions of the work.

2 Classification of Parallel Programming Models

Strictly speaking, a parallel programming model is an abstraction of the computer system architecture [10]. Therefore, it is not tied to any specific machine type. However, there are many possible models for parallel computing because of the different ways several processors can be put together to build a parallel system. In addition, separating the model from its actual implementation is often difficult. Parallel programming models and its associated implementations, i.e., the parallel programming environments defined by Mattson et al. [10], are overwhelming. However, in the late 1990s two approaches become predominant in the HPC parallel programming landscape: OpenMP for shared memory and MPI for distributed memory [10]. This allows us to define the classical or *pure parallel models*. In addition, the new processor architectures, multi-core CPUs and many-core GPUs, have produced *heterogeneous parallel programming models*. Also, the simulation of a global memory space in a distributed environment leads to the *Partitioned Global Address Space (PGAS) model*. Finally, the architectures available today allow definition of *hybrid, shared-distributed memory + GPU, models*. The parallel computing landscape would not be complete without considering the *languages with parallel support* and the *distributed programming model*. All these topics are presented in the next sections.

3 Pure Parallel Programming Models

Here, we consider parallel programming models using a pure shared or distributed memory approach. As such, we consider the threads, shared memory OpenMP, and

TABLE 1
Pure Parallel Programming Models Implementations

Implementation:	Pthreads	OpenMP	MPI
Programming Model	Threads	Shared Memory	Message Passing
System Architecture	Shared memory	Shared memory	Distributed and Shared memory
Communication Model	Shared Address	Shared Address	Message Passing or Shared Address
Granularity	Course or Fine	Fine	Course or Fine
Synchronization	Explicit	Implicit	Implicit or Explicit
Implementation	Library	Compiler	Library
WebPage	a)	b)	c)

distributed memory message passing models. Table 1 collects the characteristics of the usual implementations of these models.

3.1 POSIX Threads

In this model, we have several concurrent execution paths (the threads) that can be controlled independently. A thread is a lightweight process having its own program counter and execution stack [9]. The model is very flexible, but low level, and is usually associated with shared memory and operating systems. In 1995 a standard was released [21]: the POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995), or as it is usually called Pthreads.

The Pthreads, or Portable Operating System Interface (POSIX) Threads, is a set of C programming language types and procedure calls [7], [22], [23]. Pthreads is implemented as a header (pthread.h) and a library for creating and manipulating each thread. The Pthreads library provides functions for creating and destroying threads and for coordinating thread activities via constructs designed to ensure exclusive access to selected memory locations (locks and condition variables). This model is especially appropriate for the fork/join parallel programming pattern [10].

In the POSIX model, the dynamically allocated heap memory, and obviously the global variables, is shared by the threads. This can cause programming difficulties. Often, one needs a variable that is global to the routines called within a thread but that is not shared between threads. A set of Pthreads functions is used to manipulate thread local storage to address these requirements. Moreover, when multiple threads access the shared data, programmers have to be aware of race conditions and deadlocks. To protect critical section, i.e., the portion of code where only one thread must reach shared data, Pthreads provides mutex (mutual exclusion) and semaphores [24]. Mutex permits only one thread to enter a critical section at a time, whereas semaphores allow several threads to enter a critical section.

In general, Pthreads is not recommended as a general-purpose parallel program development technology. While it has its place in specialized situations, and in the hands of expert programmers, the unstructured nature of Pthreads constructs makes difficult the development of correct and maintainable programs. In addition, recall that the number of threads is not related to the number of processors available. These characteristics make Pthreads programs not easily scalable to a large number of processors [6]. For all these reasons, the explicitly-managed threads model is not well suited for the development of HPC applications.

3.2 Shared Memory OpenMP

Strictly speaking, this is also a multithreaded model, as the previous one. However, here we refer to a shared memory parallel programming model that is task oriented and works at a higher abstraction level than threads. This model is in practice inseparable from its practical implementation: OpenMP.

OpenMP [25] is a shared memory application programming interface (API) whose aim is to ease shared memory parallel programming. The OpenMP multithreading interface [16] is specifically designed to support HPC programs. It is also portable across shared memory architectures. OpenMP differs from Pthreads in several significant ways. While Pthreads is purely implemented as a library, OpenMP is implemented as a combination of a set of compiler directives, pragmas, and a runtime providing both management of the thread pool and a set of library routines. These directives instruct the compiler to create threads, perform synchronization operations, and manage shared memory. Therefore, OpenMP does require specialized compiler support to understand and process these directives. At present, an increasing number of OpenMP versions for Fortran, C, and C++ are available in free and proprietary compilers, see Appendix 1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.308.

In OpenMP the use of threads is highly structured because it was designed specifically for parallel applications. In particular, the switch between sequential and parallel sections of code follows the fork/join model [9]. This is a block-structured approach for introducing concurrency. A single thread of control splits into some number of independent threads (the fork). When all the threads have completed the execution of their specified tasks, they resume the sequential execution (the join). A fork/join block corresponds to a parallel region, which is defined using the PARALLEL and END PARALLEL directives.

The parallel region enables a single task to be replicated across a set of threads. However, in parallel programs is very common the distribution of different tasks across a set of threads, such as parallel iterations over the index set of a loop. Thus, there is a set of directives enabling each thread to execute a different task. This procedure is called worksharing. Therefore, OpenMP is specially suited for the loop parallel program structure pattern, although the SPMD and fork/join patterns also benefit from this programming environment [10].

OpenMP provides application-oriented synchronization primitives, which make easier to write parallel programs. By including these primitives as basic OpenMP operations, it is possible to generate efficient code more easily than, for instance, using Pthreads and working in terms of mutex and condition variables.

In May 2008 the OpenMP 3.0 version was released [26]. The major change in this version was the support for explicit tasks. Explicit tasks ease the parallelization of applications where units of work are generated dynamically, as in recursive structures or in while loops. This new characteristic is very powerful. By supporting while loops and other iterative control structures, it is possible to handle graph algorithms and dynamic data structures, for instance.

The characteristics of OpenMP allow for a high abstraction level, making it well suited for developing HPC applications in shared memory systems. The pragma directives make easy to obtain concurrent code from serial code. In addition, the existence of specific directives eases to parallelize loop-based code. However, the high cost of traditional multiprocessor machines prevented the widespread use of OpenMP. Nevertheless, the ubiquitous availability of multi-core processors has renewed the interest for this parallel programming model.

3.3 Message Passing

Message Passing is a parallel programming model where communication between processes is done by interchanging messages. This is a natural model for a distributed memory system, where communication cannot be achieved by sharing variables. There are more or less pure realizations of this model such as Aggregate Remote Memory Copy Interface (ARMCI), which allows a programming approach between message passing and shared memory. ARMCI is detailed later in Section 5.2.1. However, over time, a standard has evolved and dominated for this model: the Message Passing Interface (MPI).

MPI is a specification for message passing operations [6], [27], [28], [29]. MPI is a library, not a language. It specifies the names, calling sequences, and results of the subroutines or functions to be called from Fortran, C or C++ programs. Thus, the programs can be compiled with ordinary compilers but must be linked with the MPI library. MPI is currently the *de facto* standard for HPC applications on distributed architectures. By its nature it favors the SPMD and, to a lesser extent, the Master/Worker program structure patterns [10]. Appendix 2, available in the online supplemental material, collects some well-known MPI implementations. It is interesting to note that MPICH-G2 and GridMPI are MPI implementations for computational Grid environments. Thus, MPI applications can be run on different nodes of computational Grids implementing well-established middlewares such as Globus (the *de facto* standard, see Section 8.1 later) [30].

MPI addresses the message-passing model [6], [27], [28]. In this model, the processes executed in parallel have separate memory address spaces. Communication occurs when part of the address space of one process is copied into the address space of another process. This operation is cooperative and occurs only when the first process executes a send operation and the second process executes a receive operation. In MPI, the workload partitioning and task mapping have to be done by the programmer, similarly to Pthreads. Programmers have to manage what tasks are to be computed by each process. Communication models in MPI comprise point-to-point, collective, one-sided, and parallel I/O operations. Point-to-point operations such as the "MPI_Send"/"MPI_Recv" pair facilitate communications between processes. Collective operations such as "MPI_Bcast" ease communications involving more than two processes. Regular MPI send/receive communication uses a two-sided model. This means that matching operations by sender and receiver are required. Therefore, some amount of synchronization is needed to manage the matching of sends and receives, and the associated buffer space, of messages. However, starting from MPI-2 [31], one-sided communications are possible. Here, no sender-receiver matching is needed. Thus, one-sided communication decouples data transfer from synchronization. One-sided communication allows remote memory access. Three communication calls are provided: "MPI_Put" (remote write), "MPI_Get" (remote read), and "MPI_Accumulate" (remote update). Finally, parallel I/O is a major component of MPI-2, providing access to external devices exploiting data types and communicators [28].

On the other hand, with Symmetric Multiprocessing (SMP) machines being commonly available, and multi-core processors becoming the norm, a programming model to be considered is a mixture of message passing and multithreading. In this model, user programs consist of one or more MPI processes on each SMP node or multi-core processor, with each MPI process itself comprising multiple threads. The MPI-2 Standard [31] has clearly defined the interaction between MPI and user created threads in an MPI program. This specification was written with the goal of allowing users to write multithreaded MPI programs easily. Thus, MPI supports four "levels" of thread safety that a user must explicitly select:

- MPI THREAD SINGLE. A process has only one thread of execution.
- MPI THREAD FUNNELED. A process may be multithreaded, but only the thread that initialized MPI can make MPI calls.
- MPI THREAD SERIALIZED. A process may be multithreaded, but only one thread at a time can make MPI calls.
- MPI THREAD MULTIPLE. A process may be multithreaded and multiple threads can call MPI functions simultaneously.

Further details about thread safety is provided in [31]. In addition, in [32] the authors analyze and discuss critical issues of thread-safe MPI implementations.

In summary, MPI is well suited for applications where portability, both in space (across different systems existing now) and in time (across generations of computers), is important. MPI is also an excellent choice for task-parallel computations and for applications where the data structures are dynamic, such as unstructured mesh computations.

Over the last two decades (the computer cluster era), message passing, and specifically MPI, has become the HPC standard approach. Thus, most of the current scientific code allows for parallel execution under the message passing model. Examples are: the molecular electronic structure codes NWChem [33] and Gamess [34], or mpiBLAST [35] the parallel version of the Basic Local Alignment Search Tool (BLAST) used to find regions of local similarity between nucleotide or protein sequences. Message passing (colloquially understood as MPI) is so tied to HPC and scientific computing that, at present, in many scientific fields HPC is synonymous of MPI programming.

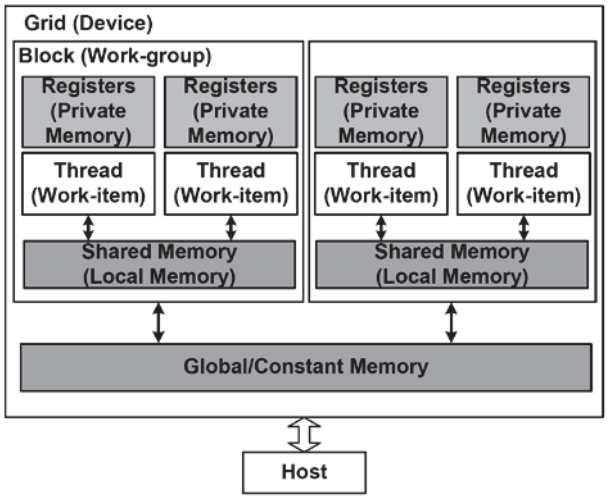

Fig. 1. CUDA (OpenCL) architecture and memory model.

4 Heterogeneous Parallel Programming Models

In the beginning of 2001 NVIDIA introduced the first programmable GPU: GeForce3. Later, in 2003 the Siggraph/Eurographics Graphics Hardware workshop, held in San Diego, showed a shift from graphics to nongraphics applications of the GPUs [36]. Thus, the GPGPU concept was born. Today, it is possible to have, in a single system, one or more host CPUs and one or more GPUs. In this sense, we can speak of heterogeneous systems. Therefore, a programming model oriented toward these systems has appeared. The heterogeneous model is foreseeable to become a mainstream approach due to the microprocessors industry interest in the development of Accelerated Processing Units (APUs). An APU integrates the CPU (multi-core) and a GPU on the same die. This design allows for a better data transfer rate and lower power consumption. AMD Fusion [37] and Intel Sandy Bridge [38] APUs are examples of this tendency.

In the first CPU+GPU systems, languages as Brook [39] or Cg [40] were used. However, NVIDIA has popularized (Compute Unified Device Architecture) CUDA [41] as the primary model and language to program their GPUs. More recently, the industry has worked together on the Open Computing Language (OpenCL) standard [42] as a common model for heterogeneous programming. In addition, different proprietary solutions, such as Microsoft's DirectCompute [43] or Intel's Array Building Blocks (ArBB) [44], are available. This section reviews these approaches.

4.1 CUDA

CUDA is a parallel programming model developed by NVIDIA [41]. The CUDA project started at 2006 with the first CUDA SDK released in early 2007. The CUDA model is designed to develop applications scaling transparently with the increasing number of processor cores provided by the GPUs [1], [45]. CUDA provides a software environment that allows developers to use C as high-level programming language. In addition, other languages bindings or application programming interfaces are supported; see Appendix 3, available in the online supplemental material.

For CUDA, a parallel system consists of a host (i.e., CPU) and a computation resource or device (i.e., GPU). The computation of tasks is done in the GPU by a set of threads running in parallel. The GPU threads architecture consists in a two-level hierarchy, namely the block and the grid, see Fig. 1.

The block is a set of tightly coupled threads, each identified by a thread ID. On the other hand, the grid is a set of loosely coupled blocks with similar size and dimension. There is no synchronization at all between the blocks, and an entire grid is handled by a single GPU. The GPU is organized as a collection of multiprocessors, with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple multiprocessors. Threads within a block can cooperate by sharing data through some shared memory, and by synchronizing their execution to coordinate memory accesses. More detailed information can be found in [41], [46]. Moreover, there is a best practices guide that can be useful to programmers [47]. CUDA is well suited for implementing the SPMD parallel design pattern [10].

Worker management in CUDA is done implicitly. That is, programmers do not manage thread creations and destructions. They just need to specify the dimension of the grid and block required to process a certain task. Workload partitioning and worker mapping in CUDA is done explicitly. Programmers have to define the workload to be run in parallel by using the function "Global Function" and specifying the dimension and size of the grid and of each block.

The CUDA memory model is shown in Fig. 1. At the bottom of the figure, we see the global and constant memories. These are the memories that the host code can write to and read from. Constant memory allows read-only access by the device. Inside a block, we have the shared memory and the registers or local memory. The shared memory can be accessed by all threads in a block. The registers are independent for each thread.

Finally, we would like to mention a recent initiative by Intel. This initiative is called Knights Ferry [48], [49], and is

being developed under the Intel Many Integrated Core (MIC) architecture. Knight Ferry is implemented on a PCI card with 32 x86 cores. The MIC supports a more classical coherent shared memory parallel programming paradigm than CUDA. Moreover, it will be programmed using native C/C++ compilers from Intel.

4.2 OpenCL

OpenCL [42], [50], [51] is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors. The first specification of OpenCL, OpenCL 1.0, was finished in late 2008 by the Khronos Group [42]. Essentially, OpenCL distinguishes between the devices (usually GPUs or CPUs) and the host (CPU). The idea behind OpenCL is to write kernels (functions that execute on OpenCL devices) and APIs for creating and managing these kernels. The kernels are compiled for the targeted device in runtime, during the application execution. This enables the host application to take advantage of all the computing devices in the system.

The OpenCL operational model can be described as four interrelated models: the platform, execution, memory, and programming models. The platform model is viewed from a hierarchical and abstract perspective. In this model, a host coordinates execution, transferring data to and from an array of Compute Devices. Each Compute Device is composed of an array of Compute Units. Each Compute Unit is composed of an array of Processing Elements.

Execution of an OpenCL program involves simultaneous execution of multiple instances of a kernel on one or more OpenCL devices. A kernel is the basic executable code unit. It is called work-item and has a unique ID. Work-items can be organized into work-groups for synchronization and communication purposes. The different executions of a program are queued and controlled by the host application. This last, sets up the context in which the kernels run, including memory allocation, data transfer among memory objects, and creation of command queues used to control the sequence in which commands are run. However, the programmer is responsible for synchronizing any necessary execution order.

OpenCL defines a multilevel memory model similarly to CUDA, see Fig. 1. First, we have the Private memory that can only be used by a single work-item. In an upper level, we found the Local memory that can be used by all work-items in a work-group. Next, the constant memory is reserved for read-only access by work-items of any work-group in a single compute device. This memory can be written and read by the host application, but remains constant during the execution of a kernel. Finally, we have the global memory, which is available for reading and writing by all work-items in all work-groups on the device. Like CUDA, OpenCL is well suited for implementing the SPMD parallel design pattern [10].

OpenCL has been designed to be used not only in GPUs but also in other platforms like multi-core CPUs. Thus, it can support both data parallel [52], and task parallel [53] programming patterns [10], which are well suited for GPUs and CPUs architectures, respectively.

OpenCL provides APIs for high-level languages. The main supported APIs are for C [50] and C++ [54]. However, there are other language bindings or application programming interfaces supported, see Appendix 4 available in the online supplemental material.

4.3 DirectCompute

DirectCompute is Microsoft's approach to GPU programming. DirectCompute is a part of the Microsoft DirectX APIs collection [43]. In fact, it is also known as DirectX11 Compute Shader. It was initially released with the DirectX 11 API, but runs on both DirectX 10 and DirectX 11 graphics processing units. In particular, it was introduced thanks to the new Shader Model 5 [55] provided in DirectX 11, which allows computation independently of the graphic pipeline, therefore suitable for GPGPUs. The main drawback of DirectCompute is that it only works on Windows platforms.

4.4 Array Building Blocks

Intel's Array Building Blocks (ArBB) provides a generalized vector-parallel-programming solution for data-intensive mathematical computation [44], [56], [57]. Users express computations as operations on arrays and vectors. ArBB comprises a standard C++ library interface and a powerful runtime. A just-in-time (JIT) compiler supplied with the library translates the operations into target-dependent code, where a target could be the host CPU or an attached GPU. As runtime, ArBB uses Intel's Threading Building Blocks [58], which contributes to abstract platform details and threading mechanisms for scalability and performance. Intel's ArBB can run data-parallel vector computations on a possibly heterogeneous system. By design, Intel ArBB prevents parallel programming bugs such as data races and deadlocks.

Thanks to the GPUs, heterogeneous programming is becoming a valuable tool in the HPC arena. As a few examples, we have applications in linear algebra [59], molecular dynamics [60], medical imaging [61] or bioinformatics [62]. For excellent surveys of GPUs capabilities and applications see [63], [64].

5 Partitioned Global Address Space

Shared memory parallel programs are considered easier to develop than message passing programs. However, message passing usually achieves better scalability and portability. This is because shared memory parallel programming models do not exploit cache data locality effectively. Distributed Shared Memory (DSM) models try to combine the advantages of both approaches, supporting the notion of shared memory in a distributed architecture. DSM approaches are in the arena since the late 1980's [65]. In DSM models, each processor sees its own memory operations in the order specified by its program. This does not automatically protect processors from seeing each other's operations (and data) out of order. This results in a memory consistency problem that was a key issue in the development of early DSM systems [65]. The lack of locality awareness enhances the problem.

The PGAS memory model is a DSM approach that implements a locality-aware paradigm [66]. PGAS provides a global address space, along with an explicitly SPMD control model. PGAS implementations typically make the

distinction between local and remote memory references. This permits to exploit data locality, which leads to a performance increase on distributed memory hardware [67]. In the PGAS model, SPMD threads (or processes) share a part of their address space. In addition, a part of the shared space is local to each thread or process. Data structures can be allocated either globally or privately. Global data structures are distributed across address spaces, typically under the control of the programmer. Remote global data are accessible to any thread as simple assignment or dereference operations. The compiler and runtime are responsible for converting such operations into messages between processes on a distributed memory machine. Thus, programs using the PGAS model can exploit locality by making each thread or process to work principally with its local data [66], [67].

While programs may require nothing else but communication through global data structures, most PGAS languages provide APIs for bulk communication and synchronization. In this sense, several libraries are being used as PGAS runtimes. Apart from the runtime libraries, there are languages specifically designed to support the PGAS memory model. In this context, languages developed under DARPA's High Productivity Computing Systems (HPCS) project [68] deserve special consideration. Runtimes, PGAS and HPCS languages are considered in the following sections.

5.1 PGAS Runtimes

Here, we present the most common runtimes available.

5.1.1 Global-Address Space Networking (GASNet)

GASNet is a language-independent, low-level networking layer that provides network-independent primitives and high-performance one-sided communication [69], [70]. GASNet is intended to be used as a compilation target and as a tool for runtime libraries development. The design is partitioned into two layers to maximize portability without sacrificing performance. The lower layer is a general interface called the GASNet core API. This is based on Active Messages [71], and is implemented directly on top of each individual network architecture. The upper layer is a wider and more expressive interface called the GASNet extended API, which provides high-level operations such as remote memory access and various collective operations. In the context of PGAS languages, Unified Parallel C (UPC), Coarray Fortran (CAF), Titanium, and Chapel all use GASNet. These languages are considered later in the present section.

5.1.2 Aggregate Remote Memory Copy Interface

ARMCI [72], [73], [74] is a library offering remote memory copy functionality. In addition, ARMCI includes a set of atomic and mutual exclusion operations. ARMCI development is driven by the need to support the global-address space communication model in contexts of regular or irregular distributed data structures, communication libraries, and compilers. One-sided put/get operations are allowed. ARMCI provides compatibility with message-passing libraries (primarily MPI), which is necessary for applications that frequently use hybrid shared-distributed memory programming models. Both blocking and a nonblocking APIs are needed. The nonblocking API can be used by some applications to overlap computations and communications.

5.1.3 Kernel Lattice Parallelism (KeLP)

KeLP [75] is a C++ class library built on the standard Message Passing Interface, MPI. Thus, it acts as a middleware between the application and the low-level communication substrate. KeLP interoperates with MPI, which eases low-level performance tuning. KeLP supports a small set of geometric programming abstractions to represent data structure and data motion. KeLP's data orchestration model separates the description of communication patterns from the interpretation of these patterns. The programmer uses intuitive geometric constructs to express dependence patterns among dynamic collections of arrays [76].

5.2 Languages Supporting the PGAS Model

Here, we consider the most significant ones.

5.2.1 Unified Parallel C

Unified Parallel C is an extension of the C programming language designed for high-performance computing on large-scale parallel machines [77], [78], [79]. The main goals are to provide an efficient access to the underlying machine and to establish a common syntax and semantics for parallel programming in C. UPC combines the programmability advantages of the shared memory paradigm, and the control over data layout and performance of message passing. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses the SPMD computation pattern [10]. The amount of parallelism is fixed at the program startup, typically with a single thread of execution per processor [78], [79].

5.2.2 Coarray Fortran/Fortran 2008

CAF appeared as a small extension of Fortran 95 for parallel processing [80], [81]. However, the most recent Fortran standard, Fortran 2008 [82], approved in September 2010, incorporates coarrays as part of the language definition.

The coarray extension addresses the two fundamental issues of any parallel programming model: work distribution and data distribution [83]. With respect to work distribution, the coarray extension adopts the SPMD programming pattern. Thus, a single program is replicated a fixed number of times. Each replication, called image, has its own set of data objects. The number of images may be equal to, greater or less than the number of physical processors. The images execute mostly asynchronously. Synchronization can be requested by the programmer through specific statements. On the other hand, with respect to data distribution, coarrays allow the programmer to specify the relationship among memory images. This is simple, since coarrays are like ordinary variables but have an index in square brackets for access between images. In this way, references without square brackets correspond to local data, while a square bracketed reference involves a communication between images.

Due to its novelty and lack of compiler support, Fortran 2008 has not yet made its entrance in the HPC world.

5.2.3 Titanium

Titanium is a language and system for high-performance parallel scientific computing. Titanium uses Java as its base, but it is not a strict extension of it. The main additions to Java are immutable classes, multidimensional arrays, an explicitly parallel SPMD pattern of computation with a global address space, and zone-based memory management [84], [85]. An important feature is that Titanium programs can run unmodified on uniprocessors, shared memory machines, and distributed memory machines. Nonetheless, performance tuning may be necessary to arrange an application's data structures for distributed memory.

The development team is working on the design of program analysis techniques and optimizing transformations for Titanium programs. A compiler and a runtime system exploiting these techniques are also being considered [84]. The compiler optimizes the code and translates Titanium into C. Then, it is compiled to native binaries by a C compiler and linked to the Titanium runtime libraries (there is no Java Virtual Machine (JVM)).

5.3 High Productivity Computing Systems Languages

5.3.1 X10

X10 is a Java-derived, type-safe, parallel object-oriented language developed in the IBM PERCS project [86] as part of the DARPA program on HPCS [68]. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end computers. X10 introduces a flexible treatment of concurrency, distribution, and locality, within an integrated type system. It extends the PGAS model to the globally asynchronous, locally synchronous (GALS) model, originally developed in hardware and in embedded software research [87]. Locality is managed explicitly using places, computational units with local shared memory. A program runs over a set of places. Each place can host data or run activities. An activity is a lightweight thread that can run on its place, or (explicitly or implicitly) asynchronously update memory, in other places [86], [88]. For synchronization, X10 uses "clocks," which are a generalization of barriers. Clocks permit activities to synchronize repeatedly. They provide a structured, distributed, and determinate form of coordination.

5.3.2 Chapel

Chapel [89] is a parallel programming language developed by Cray, Inc., as part of DARPA's HPCS program [68]. Chapel is a portable language designed to improve the programmability of large-scale parallel computers, while matching or beating the performance and portability of current programming models like MPI. Chapel supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel also includes locality awareness, which provides distribution of shared data structures without requiring a fragmentation of control structure. It also supports code reuse and rapid prototyping via object-oriented design and features for generic programming [88], [90].

5.3.3 Fortress

Fortress [91] is a new programming language designed by SUN Microsystems for HPC with high programmability. This language is also a part of DARPA's HPCS program [68]. At present, Fortress is an open-source project. The name "Fortress" is derived from the intent to produce a "secure Fortran," i.e., a language for HPC providing abstraction and type safety according to modern programming language principles. However, this is a completely new language in which all aspects of the design have been rethought from the ground up. As a result, it supports features such as transactions, specification of locality, and implicit parallel computation, as integral features built into the core of the language. It has a novel type system to integrate functional and object-oriented programming better. Thus, it supports mathematical notation and static checking of properties, such as physical units and dimensions, static type checking of multidimensional arrays and matrices, and definitions of domain-specific language syntax in libraries. Moreover, Fortress has been designed with the intent to be a "growable" language, gracefully supporting the addition of future language features [88], [92].

Despite its undeniable advantages, PGAS is not a model suitable for general environments. In fact, two drawbacks make its adoption outside the HPC niche difficult. First, the PGAS model implicitly assumes that all processes run on similar hardware. Second, the PGAS model does not support dynamically spawning multiple activities. This makes difficult to handle many non-HPC/non-data-parallel applications, like those that require runtime dynamic load balancing. For that reason, an extension called the Asynchronous Partitioned Global Address Space (APGAS) has been proposed [67].

6 Hybrid Programming

Combining the shared memory and distributed memory programming models is an old idea [93]. The goal is to exploit the strengths of both models: the efficiency, memory savings and ease of programming of the shared memory model with the scalability of the distributed memory one. In fact, this is the ultimate target of the PGAS model. However, rather than developing new runtimes or languages, we can rely in mixing the already available programming models and tools. This approach is known as hybrid (parallel) programming. This programming model is a modern software trend for the current hybrid hardware architectures. The basic idea is to use message passing (usually MPI) across the distributed nodes and shared memory (usually OpenMP or even Pthreads) within a node. Hybrid programming can also involve the use of GPUs as source of computing power. Typically, CUDA is used here, although other GPU programming approaches, such as OpenCL, could be used. However, since OpenCL directly supports multi-GPU and GPU+CPU programming its use is not specially extended in the hybrid programming field.

Along this section, we will present the different hybrid approaches.

6.1 Combining Pthreads and MPI

Combining Pthreads and MPI we could take full advantage of the shared memory available on individual multi-core cluster nodes. The idea is to use Pthreads to extend MPI, improving the speed and efficiency of the program. In this way, Pthreads is used to generate concurrent tasks to be executed on a single processor with the results gathered using shared memory. In addition, MPI permits the communication between nodes, allowing working with the SMP cluster as a whole.

We have found several examples of this hybrid programming model. In [94], the authors used it to develop a parallel file compression program. Other example is found in [95], where this model is used for discovering bounded prime numbers. A last example is found in [96], where this programming model is used to develop a parallel version of the RAxML code for phylogenetic studies. However, the use of this model is not widely extended due to the drawbacks that involve programming with Pthreads, see Section 3.1.

6.2 Combining MPI and OpenMP

The rationale of hybrid MPI/OpenMP programming is to take advantage of the features of both programming models. Thus, it mixes the explicit decomposition and task placement of MPI with the simple and fine-grain parallelization of OpenMP. This model likely represents the most widespread use of mixed programming on SMP clusters. The reasons are its portability and the fact that MPI and OpenMP are industry standards. However, it is not clear that this programming model will always be the most effective mechanism. So, it cannot be regarded as ideal for all codes. In practice, serious consideration must be given to the nature of the codes before embarking on a mixed mode implementation. Considerable work has gone into studying this hybrid model [97], [98], [99]. Here, we have collected some reasons justifying to combine MPI and OpenMP:

- The programming model matches the current hardware trend (multi-core and multiprocessor machines).
- Some applications clearly expose two levels of parallelism: coarse-grained (suitable for MPI), and fine-grained (best suited for OpenMP).
- There are situations in which the application requirements or system restrictions may limit the number of MPI processes (scalability problems). Thus, OpenMP can offer an additional amount of parallelism.
- Some applications show an unbalanced workload at the MPI level. OpenMP can be used to address this issue by assigning a different number of threads to each MPI process.
- OpenMP avoids the extra communication overhead within computer nodes induced by MPI. Thus, the memory latency and data movement within a node is reduced because it is possible to synchronize on memory instead of using synchronization barriers.

Nevertheless, there are also reasons that make the use of this programming model inefficient:

- Introducing OpenMP into an existing MPI code also means introducing the drawbacks of OpenMP such as:
  - Limitations when controlling work distribution and synchronization.
  - Overhead introduced by threads creation and synchronization.
  - Dependence on the quality of the compiler and the runtime support for OpenMP.
- Shared memory issues (for instance in ccNUMA architectures).
- The interaction of MPI and OpenMP runtime libraries may have negative side effects on the program's performance.
- Some applications naturally expose only one level of parallelism, and there may be no benefit in introducing a hierarchical parallelism pattern.

Most of the hybrid MPI/OpenMP code is based on a hierarchical model, which makes possible to exploit large and medium-grain parallelism at the MPI level, and fine-grain parallelism at the OpenMP level. Thus, at high level, the program is explicitly structured as several MPI tasks, whose sequential code is enriched with OpenMP directives to add multithreading features taking advantage of the presence of shared memory. This programming model can be implemented in different ways depending on the overlap between communication and computation [98]. In particular, there are two main categories: no overlapping communication/computation and overlapping communication/computation. In the first one, there are no MPI calls overlapping with other application code in other threads. This category can be implemented in two ways [98]:

1. MPI is called only outside parallel regions and on the master thread. The advantage of this method is that there is no message passing inside SMP nodes. Thus, we have no problem with the topology since the master thread controls communications between nodes. On the other hand, the drawback could be the efficiency, since all other threads are sleeping while the master thread communicates. In addition, the MPI library must support at least the MPI_THREAD_FUNNELED level of thread safety.
2. MPI is called outside the parallel regions of the application code, but the MPI communication is done itself by several CPUs. The thread parallelization of the MPI communication can be done automatically by the MPI library routines, or explicitly by the application, using a full thread-safe MPI library.

In this first category, the noncommunicating threads are sleeping (or executing another application, if nondedicated nodes are used). This problem of idling CPUs is solved in the next category of MPI/OpenMP methods.

The second category corresponds to overlapping communication and computation. Here, while the communication is done by the master thread (or a few threads), all other noncommunicating threads are executing application code. As in the previous case, we can find two approaches:

1. Only the master thread calls MPI routines, i.e., all communications are funneled to the master thread.

2. Each thread handles its own communication needs, or the communication is funneled to more than one thread.

The main problem of this second category is that the application must be splitted into code that can run without access to the nonlocal data, and code that needs such access, which is very complicated. In addition, using this method, we lose the major OpenMP advantage because the communication/computation is done via threads rank (the thread ID in OpenMP). Therefore, we cannot use worksharing directives. Finally, the communication load of the threads is inherently unbalanced. To tackle this last problem, we can use load balancing strategies like fixed reservation or adaptive balancing [98].

Once the different strategies to develop a hybrid application are known, the ideal situation is to build it, using one of the previous methods, from scratch. Nevertheless, this is not always possible. Sometimes, it is necessary to build the application from previous MPI or OpenMP code. Here, in the "retrofit" process leading from the initial to the hybrid code, several issues can be taken into account [99]:

- Retrofit MPI applications with OpenMP. This is the easiest "retrofit" option because the program state synchronization is already explicitly handled. The benefits depend on the amount of work that can be parallelized with OpenMP (usually loop-level parallelization). In addition, this kind of "retrofit" is beneficial for communication bound applications, since it reduces the number of MPI processes needing to communicate. However, CPU processor use on each node becomes an issue to study during the "retrofit" process.
- Retrofit OpenMP applications with MPI. This case is not as straightforward as the previous one since the program state must be explicitly handled with MPI. This approach requires careful consideration of how each process will communicate with the others. Sometimes, it may require a complete redesign of the parallelization. Nevertheless, a successful "retrofit" usually yields greater improvements in performance and scaling.

An advantage of the MPI+OpenMP approach is that MPI one-sided communications decouples data transfer from synchronization, whereas multithreading relaxes the SPMD design pattern usually applied. We have found many applications in which this hybrid programming model provides clear performance improvement. For example, Bova et al. [100] have developed mixed mode versions of five separate codes: the general-purpose wave prediction application CGWAVE, the molecular electronic structure package GAMESS, a Linear algebra application, a thin-layer Navier-Stokes solver (TLNS3D), and the seismic processing benchmark SPECseis96. Bush et al. [101] have developed mixed MPI/OpenMP versions of some kernel algorithms and larger applications. Successful examples of applications can also be found for coastal wave analysis [102], atmospheric modeling [103], iterative solvers for finite-element methods [104], and performance simulations of an MPI/OpenMP code for the N-body problem under the HeSSE environment [105].

Nevertheless, as commented before, this model is not always the most efficient alternative. For example, Cappello and Etiemble [106], Duthie et al. [107], Smith [108], Henty [109], and Chow and Hysom [110] all show in several examples that the pure MPI codes outperform their mixed counterparts despite the underlying architecture.

6.3 Combining CUDA and Pthreads

This is an easy way to support multi-GPU parallelism. Thus, using this model, one CPU thread is assigned to each GPU. Therefore, each device has its own context on the host. An example of this programming model can be found in [111] applied to the Navier-Stokes solver. The main problem is that the programmer has to split the code to provide the same amount of work to each GPU.

Other way to apply this model is taking advantage of multi-core CPUs. For example, in [112] Pthreads are used to preprocess some data that later are sent to the GPU to be fully processed. In this way, CPU and GPU computations are overlapped.

6.4 Combining CUDA and OpenMP

As in the previous case, OpenMP can be used to optimize the generation of input data and their transference toward a GPU. The problem is that CUDA cannot share the CPU and GPU memories. That means that the GPU needs to receive input data from the CPU to implement operations. Thus, to take advantage of the efficiency of the GPU, it is essential to minimize data transmission between the CPU and the GPU. To tackle this problem, the authors in [113] have used OpenMP, which allows the CPU to generate as much data as possible (data preprocessing). Other example is found in [114], where this model is used to implement a parallel cloth simulation, which offers higher animation frame rates. On the other hand, this model can be implemented using the CPU to postprocess the results obtained in the GPU. This has been applied in [115] to scene recognition issues.

Finally, as in the Pthreads case, this model could be used to support multi-GPU parallelism. However, we have not found any published work implementing this approach.

6.5 Combining CUDA and MPI

This model is useful for parallelizing programs in GPU clusters. Here, the programming environment and the hardware structure of cluster nodes are very different from traditional ones, because of the heterogeneous architecture model based on the CPU and the GPU. Unlike traditional cluster systems, this model separates process control tasks from data computing tasks. In particular, MPI is used to control the application, the communication between nodes, the data schedule, and the interaction with the CPU. Meanwhile, CUDA is used to compute the tasks in the GPU [116], [117].

In the same way, this model can be applied to GPU clusters where each node has several GPUs. The key concept is similar: one MPI process is started per GPU. Since we must ensure that each process has assigned a unique GPU identifier, an initial mapping of hosts to GPUs is performed. A master process gathers all the host names, assigns GPU identifiers to each host, such that no process on the same host has the same identifier, and scatters the result back [118],

[119]. As an example, in [119] the authors optimize its incompressible flow solver code using this model.

This programming model is not the ideal approach for all parallel applications. There are instances where it delivers poor performance. In [120] Karunadasa and Ranasinghe applied this model to Strassen [121] and Conjugate Gradient [122] algorithms with different results. In the first algorithm the approach has shown to work fine, whereas in the second it was less effective. The authors provide two considerations to make before trying to improve the performance of an application using CUDA+MPI [120]:

- First: Does the algorithm have two levels of parallelism? Only if the algorithm has a second level of parallelism the second consideration (see next point) should be taken into account.
- Second: Does the second level of parallelism have some compute intensive part that can be processed in a GPU? If it has not a considerable amount of work that can be easily handled by the GPU, then there will not be many performance increases by executing that second level parallel component with CUDA.

From the previous sections a clear conclusion arises: there is no general solution for hybrid parallelization even when we consider the same architecture. In practice, the best solution depends on the characteristics of each application. Despite that, the hybrid models offer an attractive way to harness the possibilities provided by current architectures.

7 Languages with Parallel Support

The previous sections have considered parallel programming models. However, in the parallel programming field special interest is due to languages with direct parallel support. These languages are usually, but not always, HPC oriented. From the parallel programming standpoint these languages fall under the umbrella of the shared or distributed memory models. From the plethora of proposed languages, we consider here some representative examples of approaches not yet considered in the previous sections.

7.1 Java

Java is one of the most popular languages for common applications, but it is also a parallel programming language. In fact, Java was designed to be multithreaded [123]. This design allows the JVM to take advantage of multi-core systems for critical tasks such as JIT compilation or Garbage Collection. In [124], the authors study the performance of Java using different benchmarks. They get an overall good performance for basic arithmetic operations. With respect to intensive computations, their implementation of the Java NAS Parallel Benchmark [125] shows a similar performance that the Fortran/MPI one. However, the scalability is an issue when performing intensive communications, but this problem can be tackled using optimized network layers [126].

Java includes type safety and integrated support for parallel and distributed programming. Java provides Remote Method Invocation (RMI) for transparent communication between JVMs.

With respect to HPC, Java has the appeal of high-level constructs, memory management, and portability of applications. In addition, current Java implementations, thanks to Just-In-Time compilers, are only about a 30 percent slower than native C or Fortran applications [127], [128]. This is a great improvement over the initial Java performance, when pure interpretation of bytecode was used. This performance was estimated to be worse than that of Fortran by a factor of four [129]. Therefore, over the years, Java is becoming more and more interesting for HPC applications. Thus, is not a surprise that the message passing model was proposed for use under Java. To uniformize criteria, in 1998 the Message-Passing Working Group of the Java Grande Forum [130] was formed. This working group came up with an initial draft for a Java message passing API specification. The first implementation based on these guidelines was mpiJava [131]. mpiJava was a set of Java Native Interface (JNI) wrappers to native MPI packages. mpiJava paved the way for other implementations such as the open source MPJ Express [132] library. MPJ Express is a message passing library for Java suitable for use on computer clusters and multi-core CPUs [133]. Shafi et al. [132] present an interesting comparison of Java and MPJ Express versus C and MPI by considering the Gadget-2 massively parallel structure formation code used in cosmology. The results show that the Java version is comparable to the C one.

7.2 High Performance Fortran (HPF)

High Performance Fortran [134] is an extension of Fortran 90 with constructs supporting parallel computing. HPF is maintained by the HPFF [134]. The first version was published in 1993. The language was designed to support the data parallel programming pattern. In this pattern, a single program controls the distribution of data across all processors. It also controls the operations on these data. A primary design goal of HPF was that it would enable top performance on parallel computer architectures without sacrificing portability [135].

HPF represents a simple way to work in distributed memory environments. The language syntax presents a global memory space to the programmer, eliminating the necessity of understanding message passing issues [136]. Examples of application to typical problems such as heat distribution, the N-Body problems, or the LU decomposition can be found in [136].

7.3 Cilk

Cilk is a multithreaded language [137]. Thus, it implements a shared memory approach suitable for current multi-core CPUs. Cilk is C based and was developed at MIT in 1994. The philosophy behind Cilk is that the programmer should concentrate on structuring the program to expose parallelism and exploit locality. Thus, the Cilk runtime system [137] takes care of details like load balancing, paging, and communication protocols. The Cilk scheduler uses a "work stealing" policy to divide procedure execution efficiently among multiple processors. Thus, idle processors can perform work in benefit of busy ones. Unlike other multithreaded languages, Cilk is algorithmic. This means that the runtime system guarantees an efficient and

predictable performance [138]. A Cilk proprietary extension to C++ is distributed by Intel as Cilk Plus [139].

7.4 Z-level Programming Language (ZPL)

ZPL, originally called Orca C, was developed between 1993 and 1995 in the University of Washington, and was released to the public in 1997 [140]. ZPL is an implicitly parallel programming language [141]. This means that all the instructions to implement and manage parallelism are inserted by the compiler. ZPL is a parallel array language designed for high-performance scientific and engineering computations. ZPL uses a machine model, the Candidate Type Architecture (CTA), which abstracts Multiple Instruction Multiple Data (MIMD) parallel computers [141]. One of the goals of ZPL is performance portability, understood as consistent high performance across MIMD parallel platforms [141]. The cornerstone for ZPL's performance portability is the introduction of an explicit performance model [141].

ZPL is an extremely interesting approach. There are undeniable advantages in a language designed from the ground up to provide an abstract MIMD machine model and a portable performance model. Thus, for instance, performance is independent of the compiler and of the machine. However, the last ZPL available version is an alpha version dating back to 2000 [142]. In addition, although there are in the literature interesting examples of its use [143], it seems that the scientific and engineering community has not paid enough attention to ZPL. ZPL has had a great influence in the development of Chapel since part of the development team also participates in Chapel's developing.

7.5 Erlang

Erlang is a general-purpose, message passing concurrent functional programming language and runtime environment. Erlang started life in the telecommunications industry at Ericsson, where its developers sought to create a language for building telecommunications switches. In 1998, Erlang became open source software [144]. Erlang has built-in support for concurrency, distribution, and fault tolerance. Erlang's concurrency primitives provide more than just a fast way to create threads. They also enable parts of an application to monitor other parts, even if they are running on separate hosts across the network, and restart those other parts if they fail. Erlang's libraries and frameworks take advantage of these capabilities to let developers build systems with extreme availability and reliability [145].

Due to its design orientation, Erlang is very efficient for developing distributed, reliable, soft real-time concurrent systems such as Telecommunication systems or Servers for Internet applications. However, it is not well suited for applications where performance is a prime requirement, such as HPC systems.

7.6 Glasgow Parallel Haskell (GpH)

Glasgow parallel Haskell is a concurrent version of the functional Haskell language developed in the Glasgow University [146]. GpH applies a semiexplicit deterministic parallel programming model [147]. Here, the semantics of the program remains completely deterministic, and the programmers are not required to identify threads, communication, or synchronization. They merely annotate subcomputations that might be evaluated in parallel, leaving the choice of whether to actually do so to the runtime system. These so-called sparks are created and scheduled dynamically, and their grain size varies widely.

GpH supports execution on clusters of computers. At present, the Glasgow Haskell compiler supports GpH language constructs on shared memory machines.

8 Distributed Programming

No survey of the parallel programming landscape can be complete without considering distributed programming. Here, we have individual computing units interconnected by some network. A detailed presentation of the distributed systems topic can be found in [148]. According to the classification proposed in [148] what we can call "Distributed Computing Systems" are those specifically oriented to HPC. Examples are computer clusters and Grids of computers. The parallel programming models useful in these systems have been considered in the previous sections of this review. Here, we will consider Grid computing. In addition, the present section focuses in the colloquial meaning given to Distributed Systems. That is, the Information Distributed Systems considered in [148]. These systems are mainly, but not exclusively, enterprise-oriented, with an emphasis on interoperability among networked applications. Different programming models, architectures, and tools coexist here. This leads to a somewhat confusing landscape. In this work, we will use a chronological order to present the most representative components of this fauna.

8.1 Grid Computing

The Grid concept was first presented by Foster and Kesselman in 1998 [149]. A computational Grid is defined as a hardware and software infrastructure providing dependable, consistent, and pervasive access to resources among different administrative domains [149]. Therefore, grid computing is a distributed computing model that permits the integration of arbitrary computational resources over the Internet. An updated and detailed presentation of Grid computing can be found in [150]. Grid computing relies on a middleware layer to achieve its goals. Different Grid middlewares do exist, like gLite [151] produced in the Enabling Grids for E-sciencE (EGEE) European project [152]. However, the *de facto* standard is the Globus toolkit [30], now in its 4 version. Like any Grid middleware, Globus addresses four key elements: Grid security, data management, resource allocation, and information services [150], [30].

The perspective of harnessing the computational power of different machines makes the Grid an interesting option for HPC. However, the heterogeneous nature of the resources, and the bandwidths and latencies involved in the Internet, prevents the treatment of concurrent problems when communication among processes is important. An ideal case to be run on the Grid is that of parameter sweep problems, where several communication independent processes are run simultaneously [153]. These embarrassingly parallel problems can be tackled under a master/worker parallel programming pattern [10]. The parameter sweep model finds increasing applications in different

scientific areas, for example: Bioinformatics [154], High-Energy Physics [155], or Molecular Science [153].

8.2 CORBA

In the beginning it was CORBA, we could say. CORBA arises in the early 1990s as an attempt to permit interoperability among different applications running on distributed systems. CORBA is a standard defined by the Object Management Group [156]. The first really operative version was the 2.0, released in 1997. CORBA 2.0 provided a standardized protocol and a C++ mapping. The last CORBA version is 3.1, released in 2008.

The CORBA specification provides a stable model for distributed object-oriented systems [148]. At the core of this model lays the concept of Remote Procedure Call (RPC) introduced in 1984 by Birrel and Nelson [157]. RPC is as simple as allowing programs to call procedures in another machine. CORBA applications are composed of objects-components with well-defined interfaces that describe the services they provide to other objects in the system. Applications complying with the CORBA standard are abstracted from underlying languages, operating systems, networking protocols, and transports. Instead, they rely on object request brokers to provide a fast and flexible communication and object activation [158]. CORBA uses an interface definition language (IDL) to specify the public interfaces that the objects present to the outside world. CORBA then specifies a "mapping" from the IDL to a specific implementation language like C++ or Java. Standard mappings exist for Ada, C, C++, Lisp, Smalltalk, Java, COBOL, PL/I, and Python.

CORBA has played a major role in the distributed programming field. However, it has been somewhat relegated due to the complexity of its API, and to the lack of security, of versioning features, and of threads support. For a more detailed discussion, see [159]. CORBA is used at present to tie together components within corporate networks and for developing real-time and embedded systems [159], [160], [161]. In this last field CORBA is in widespread use.

8.3 Distributed Component Object Model (DCOM)

The DCOM is the distributed version of Microsoft's Component Object Model [162]. DCOM was intended as a Microsoft alternative to CORBA. As CORBA, DCOM is object based and relies in the use of RPC calls. DCOM was originally a proprietary development for Windows. For these reasons it received little acceptance. Now, there is an open systems implementation of DCOM, extended to UNIX, called COMsource [163]. Microsoft finally dropped DCOM for the .NET framework. For a comparison of DCOM and CORBA see [164].

8.4 Web Services

While CORBA was being introduced in the late 1990s, the web was growing and the first e-commerce applications were developed. The first approach for these systems was a client-server model where the client side was a web browser that offers an interface to the user [148], [165]. However, the need for applications that offer services to other applications became soon apparent. In this context, the term web service was coined [148], [165]. Web services are distributed software components that provide information to applications, rather than to humans, using an application-oriented interface. Web services are a distributed middleware technology using a simple XML-based protocol to allow applications to exchange data across the web. Services are described in terms of the messages accepted and generated. Users of such services do not need to know anything about the details of the implementation [165]. They only need to be able to send and receive messages. The information is structured using XML. Thus, it can be parsed and processed easily by automatic tools [165]. Key to web services are the Simple Object Access Protocol (SOAP), the web services description language (WSDL), and the Universal Description, Discovery, and Integration standard (UDDI).

SOAP is a lightweight protocol based on XML for web services to exchange messages over the web [166]. Web services support SOAP as communication protocol over either HTTP or HTTPS. The SOAP specification is maintained by the XML Protocol Working Group of the World Wide Web Consortium (W3C) [167].

WSDL is a specification (maintained by the W3C [168]) defining how to describe web services using a common XML grammar [169]. Therefore, WSDL represents a cornerstone of the web service architecture, since it provides a common language for describing services and a platform for automatically integrating those services [169].

UDDI, on the other hand, establishes the layout of a database containing service descriptions that will allow web service clients to browse for relevant services. UDDI is an open industry initiative, sponsored by the Organization for the Advancement of Structured Information Standards (OASIS) [170]. UDDI provides a standardized service registry format. This allows having a directory service storing service descriptions. This directory permits web service clients to browse for relevant services automatically.

When a web service is invoked, the SOAP request is sent over the Internet until it reaches the server holding the service. The server receives and processes the SOAP request by means of a SOAP engine. This is an application running on an appropriate application server. Some of the most popular application servers are GlassFish [171], JBoss [172], Geronimo [173], or Tomcat [174].

Web services expose public interfaces to the network. Within the three-tier systems architecture model [175], the business (logic) layer behind the public interfaces can be implemented through Enterprise JavaBeans (EJB). EJB is the server-side component architecture for the Java Platform Enterprise Edition (Java EE) [176]. EJB technology enables rapid and simplified development of distributed, transactional, secure, and portable applications based on Java technology (called EJBs). In this context, web services can be thought of as wrappers used by EJBs to invoke external services and to allow external services and clients to invoke the EJBs. EJBs are the basic building blocks of software applications on the J2EE platform, which has been the preferred choice for many enterprises for building large-scale, web-accessed applications.

The ease with which web services can be implemented, and the ability to access them from any platform, has led to their rapid adoption as virtualization agents that provide

common manageability interfaces to different resources [165]. However, web services are mainly business oriented as stated in the classic definition by the UDDI consortium, quoted in [165]: "*... self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces*." The use of standards based in XML makes the communication process between services inappropriate for HPC applications. However, web services are useful in e-Science environments for defining workflows [177]. A representative example is the EMBRACE collection of life-science web services [178]. For a detailed review of the web services paradigm see [179].

A key difference between CORBA and web services is that CORBA provides an object-oriented component architecture. On the other hand, web services are message based (SOAP does not really deal with objects). Using CORBA the coupling among the components of a system is usually high. Web services, on the other hand, are loosely coupled entities, which can work independently of each other.

8.5 SOA

Mingled with CORBA and web services, we find the Service-Oriented Architecture (SOA). SOA is an architectural pattern stating that computational units (such as system modules) should be loosely coupled through their service interfaces for delivering the desired functionality. In the well-known Erl's book [180] SOA is defined as: "*... a form of technology architecture that adheres to the principles of service-orientation. When realized through the web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and automation domains of an enterprise.*" The services do not have to be web services, though web services currently represent the *de facto* technology for realizing SOA [181]. The most important aspect of SOA is that it separates the service implementation from its interface. Therefore, the way the service performs the task requested by the user is irrelevant. The only requirement is that the service sends the response back to the user in an agreed-upon format [182].

8.6 REST

The Representational State Transfer (REST), like SOA, is an architectural style. REST was defined by Roy T. Fielding, in its doctoral Thesis, as an architectural style for distributed hypermedia systems [183]. REST specifies several architectural constraints intended to enhance performance, scalability, and resource abstraction within distributed hypermedia systems. The first one is the uniform interface constraint. That means that all resources must present the same interface to the clients. The second is statelessness. Statelessness implies that each request from client to server must contain the information necessary to understand the request. Caching is the third REST architectural constraint. This constraint requires that the data within a response to a request be implicitly or explicitly labeled as cacheable or noncacheable. If the response is cacheable, then a client cache can reuse that response data for later, equivalent, requests. Caching improves performance and scalability. For a detailed presentation of the implications of these constrains, see [184].

There is now an active REST versus SOA debate. Both approaches have their strengths and weaknesses. The main difference between SOA is that REST is resource oriented rather than service oriented. For a detailed comparison of REST and SOA, see [185].

8.7 Ice

In the chronological order used here, the newest arrival in the distributed programming arena is the Internet Communications Engine (Ice). Ice, first released in 2003, is an object oriented middleware for the development of distributed applications. ICE, developed by ZeroC [186], is dual-licensed under the GNU GPL and a commercial license. Ice provides tools, APIs, and library support for building object-oriented client-server applications. Ice has support for C++, .NET, Java, Python, Objective-C, Ruby, and PHP [186]. For a detailed presentation of the Ice middleware see [187]. Ice represents a modern alternative to CORBA since it provides an object model simpler and more powerful. This is achieved by getting rid of inefficiencies, and by providing new features such as user datagram protocol (UDP) support, asynchronous method dispatch, built-in security, automatic object persistence, and interface aggregation [188].

The distributed programming model presents serious drawbacks for HPC applications. These are the existence of limited bandwidths and large latencies among the computers in the network. The amount of communication in HPC applications makes these factors critical. Therefore, HPC applications yield poor performance and do not scale properly in actual heterogeneous and geographically disperse environments. However, as seen before, embarrassingly parallel problems can be efficiently tackled in distributed environments. This is due to the lack of communication among the individual processes involved in these problems. In fact, the key factor is the computing to communication ratio.

9 Conclusions

This work reviews the current parallel programming landscape. From the study a fact is clear: since the switch, around 2003, of the microprocessor industry to the multicore model, and the introduction of GPGPU, parallelism has become a key player in the software arena. It would be desirable that this trend would be clearly reflected in computer science curricula.

The increasing relevance of parallelism in the computational field can be illustrated by considering its effect on the computing literature over the last decade. Thus, Fig. 2 represents the number of hits in the Scopus database [189] for some significant keywords: MPI, OpenMP, CUDA, and OpenCL. We observe that MPI, i.e., the distributed memory model, takes the lion's share, with a huge increase in relevance in the last decade. This can be attributed to the affordability of computer clusters as HPC systems, and to the renewed interest in parallelism after the concurrent revolution. In fact, the distributed memory parallel programming approach is the most popular one. Therefore, the MPI library has been the *de facto* standard in the HPC field for the last two decades. Fig. 2 also shows that OpenMP

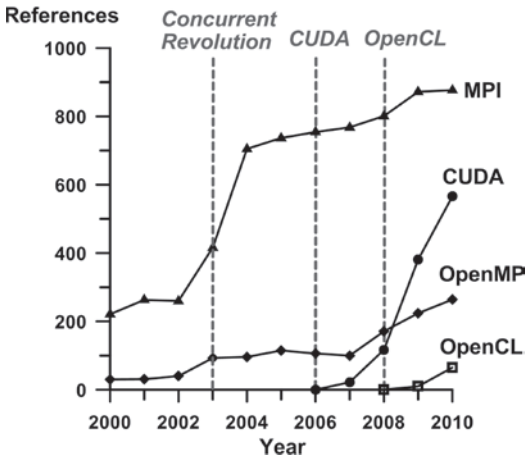

Fig. 2. Parallelism relevance trend in the computational field.

exhibits a slow but continuous increasing trend since 2003. Thus, a consequence of the current widespread use of multi-core systems is a revival of the once almost forgotten shared memory parallel programming model. However, the most significant fact observed in Fig. 2 is the exponential growth of CUDA's relevance since its introduction in 2006. The data trend suggests that GPU-based systems could, at least, stand on an equal footing with computer clusters in the foreseeable future. Finally, we also observe a clear growing tendency in OpenCL. OpenCL was introduced in 2008, but it seems that this approach could have an exponential increase as CUDA has.

The present study also shows that with current multi-core CPUs, computer clusters can mix distributed memory programming, among the cluster nodes, with shared memory parallel programming, within the cores of each node. Thus, the hybrid parallel programming approach is becoming increasingly popular. Hybrid parallel programming examples involving GPUs can also be found in the literature. Therefore, the hybrid approach offers a way to harness the possibilities of current, and legacy, architectures and systems.

In addition, the easy availability of GPUs on multi-core systems is providing momentum to a new parallel programming model: heterogeneous programming. This supersedes pure GPU programming, allowing several multi-core CPUs and several GPUs to collaborate.

Among all the different approaches existing for profiting from parallel systems those based on open industry standards are especially interesting. Open standards allow the different actors involved in the parallel world to have a voice, and contribute to their development. In words of Tim Mattson from Intel [190]: "... the *core to solving the parallel programming challenge is standards*." From this standpoint we have MPI for distributed memory and OpenMP for shared memory. For the new heterogeneous platforms the hole is filled by OpenCL. The open standards approach seems to be the way to spread and uniformize the use of parallel programming models.

REFERENCES

[1] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.
[2] W. Hwu, K. Keutzer, and T.G. Mattson, "The concurrency challenge," *IEEE Design and Test of Computers,* vol. 25, no. 4, pp. 312-320, July 2008.
[3] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *ACM Queue,* vol. 3, no. 7, pp. 54-62, 2005.
[4] W-C. Feng and P. Balaji, "Tools and Environments for Multicore and Many-Core Architectures," *Computer,* vol. 42, no. 12, pp. 26-27, Dec. 2009.
[5] R.R. Loka, W-C. Feng, and P. Balaji, "Serial Computing Is Not Dead," *Computer,* vol. 43, no. 9, pp. 6-9, Mar. 2010.
[6] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *The Sourcebook of Parallel Computing.* Morgan Kaufmann Publishers, 2003.
[7] H. Kasim, V. March, R. Zhang, and S. See, "Survey on Parallel Programming Model," *Proc. IFIP Int'l Conf. Network and Parallel Computing,* vol. 5245, pp. 266-275, Oct. 2008.
[8] M.J. Sottile, T.G. Mattson, and C.E. Rasmussen, *Introduction to Concurrency in Programming Languages.* CRC Press, 2010.
[9] G.R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 1999.
[10] T.G. Mattson, B.A. Sanders, and B. Massingill, *Patterns for Parallel Programming.* Addison-Wesley Professional, 2005.
[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings Publishing Company, 1994.
[12] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors,* vol. 1. Prentice Hall, 1988.
[13] M.J. Quinn, *Parallel Computing: Theory and Practice.* McGraw-Hill, 1994.
[14] P.B. Hansen, *Studies in Computational Science: Parallel Programming Paradigms.* Prentice-Hall, 1995.
[15] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
[16] OpenMP, "API Specification for Parallel Programming," http://openmp.org/wp/openmp-specifications, Oct. 2011.
[17] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI: The Complete Reference, the MPI-2 Extensions,* vol. 2. The MIT Press, Sept. 1998.
[18] K. Kedia, "Hybrid Programming with OpenMP and MPI," Technical Report 18.337J, Massachusetts Inst. of Technology, May 2009.
[19] D.A. Jacobsen, J.C. Thibaulty, and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *Proc. 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition,* Jan. 2010.
[20] C.-T. Yang, C.-L. Huang, and C.-F. Li, "Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters," *Computer Physics Comm.,* vol. 182, no. 1, 2011.
[21] POSIX 1003.1 FAQ, http://www.opengroup.org/austin/papers/\posix_faq.html, Oct. 2011.
[22] D.R. Butenhof, *Programming with POSIX Threads.* Addison-Wesley, 1997.
[23] IEEE, "IEEE P1003.1c/D10: Draft Standard for Information Technology - Portable Operating Systems Interface (POSIX)," Sept. 1994.
[24] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing,* second ed. Addison-Wesley, 2003.
[25] B. Chapman, G. Jost, and R. van der Pas, *Using, OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.
[26] OpenMP 3.0 Specification, http://www.openmp.org/mp-documents/spec30.pdf, Oct. 2011.
[27] P.S. Pacheco, *Parallel Programming with MPI.* Morgan Kaufmann, 1996.
[28] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* second ed. MIT Press, 1999.
[29] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, 1999.
[30] Globus, http://www.globus.org, Oct. 2011.
[31] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," July 1997.
[32] W. Gropp and R. Thakur, "Thread Safety in an MPI Implementation: Requirements and Analysis," *Parallel Computing,* vol. 33, no. 9, pp. 595-604, Sept. 2007.

[33] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong, "NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations," *Computer Physics Comm.* vol. 181, pp. 1477-1589, http://www.nwchem-sw.org, Oct. 2011.

[34] M.S. Gordon and M.W. Schmidt, "Advances in electronic structure theory: GAMESS a decade later," *Theory and Applications of Computational Chemistry, the First Forty Years*, C.E. Dykstra, G. Frenking, K.S. Kim, G.E. Scuseria, eds., Chapter 41, pp 1167-1189, Elsevier, http://www.msg.chem.iastate.edu/GAMESS/GAMESS.html, Oct. 2011.

[35] H. Lin, X. Ma, W. Feng, and N. Samatova, "Coordinating Computation and I/O in Massively Parallel Sequence Search," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 4, pp. 529-543, http://www.mpiblast.org, Oct. 2011.

[36] M. Macedonia, "The GPU Enters Computing's Mainstream," *Computer*, vol. 36, no.10, pp. 106-108, Oct. 2003.

[37] AMD Fusion, http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx, Oct. 2011.

[38] Sandy Bridge, http://software.intel.com/en-us/articles/sandy-bridge/, Oct. 2011.

[39] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *Proc. SIGGRAPH*, 2004.

[40] W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-Like Language," *Proc. SIGGRAPH*, 2003.

[41] CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html, Oct. 2011.

[42] Khronos Group, http://www.khronos.org/opencl, Oct. 2011.

[43] Microsoft DirectX Developer Center,http://msdn.microsoft.com/en-us/directx/default, Oct. 2011.

[44] Sophisticated Library for Vector Parallelism: Intel Array Building Blocks, Intel; http://software.intel.com/en-us/articles/intel-array-building-blocks, 2010.

[45] Nvidia Developer Zone, http://developer.nvidia.com, Oct. 2011.

[46] Nvidia Company. Nvidia CUDA Programming Guide, v3.0, 2010.

[47] Nvidia Company. Nvidia CUDA C Programming Best Practices Guide, Version 3.0, 2010.

[48] Michael Wolfe, "Compilers and More: Knights Ferry Versus Fermi," *HPCwire*, Aug. 2010.

[49] K. Skaugen, "Petascale to Exascale. Extending Intel's HPC Commitment"," *Proc. Int'l Supercomputing Conf. (ISC '10)*, 2010.

[50] OpenCL 1.1 Specification, http://www.khronos.org/registry/cl/specs/\opencl-1.1.pdf, Oct. 2011.

[51] Introduction to OpenCL, http://www.amd.com/us/products/technolo-gies/stream-technology/opencl/pages/opencl-intro.aspx, Oct. 2011.

[52] W.D. Hillis and G.L. Steele, "Data Parallel Algorithms," *Comm. ACM*, vol. 29, pp. 1170-1183, 1986.

[53] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[54] OpenCL 1.1 C++ Bindings Specification, http://www.khronos.org/\ registry/cl/specs/opencl-cplusplus-1.1.pdf, Oct. 2011.

[55] Shader Model 5 (Microsoft MSDN), http://msdn.microsoft.com/en-us/library/ff471356(v = vs.85).aspx, Oct. 2011.

[56] A. Ghuloum et al., "Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture," *Intel Technology J.*, vol. 11, no. 4, pp. 333-347, 2007.

[57] W. Kim, M. Voss, "Multicore Desktop Programming with Intel Threading Building Blocks," *IEEE Software*, vol. 28, no. 1, pp. 23-31, Jan./Feb. 2011.

[58] Intel Threading Building Blocks, http://www.threadingbuildingblocks.org. Oct. 2011.

[59] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *ACM Trans. Graphics*, vol. 22, pp. 908-916, 2003.

[60] D.C. Rapaport, "Enhanced Molecular Dynamics Performance with a Programmable Graphics Processor," *Computer Physics Comm.* vol. 182, pp. 926-934, 2011.

[61] F. Xu and K. Mueller, "Accelerating Popular Tomographic Reconstruction Algorithms on Commodity PC Graphics Hardware," *IEEE Trans. Nuclear Science*, vol. 52, pp. 654-663, June 2005.

[62] S.A. Manavski and G. Valle, "CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment," *BMC Bioinformatics*, vol. 9, no. 2, pp. 1-9, Mar. 2008.

[63] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcel, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80-113, 2007.

[64] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[65] J. Protic, M. Tomasevic, and V. Milotinuvic, "A survey of distributed shared memory systems," *Proc. 28th Hawaii Int'l Conf. System Sciences (HICSS '95)*, pp. 74-84, 1990.

[66] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty, and Y. Yao, "An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C," *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 36-47, 2005.

[67] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Grove, D. Cunningham, O. Tardieu, I. Peshansky, and S. Kodali, "The Asynchronous Partitioned Global Address Space Model," *Proc. First Workshop Advances in Message Passing*, 2010.

[68] DARPA's, http://www.darpa.mil/Our_Work/MTO/Programs/High_ Productivity_Computing_Systems_(HPCS).aspx, Oct. 2011.

[69] D. Bonachea and J. Jeong, "GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages," *CS258 Parallel Computer Architecture Project*, 2002.

[70] GASNet, http://gasnet.cs.berkeley.edu./, Oct. 2011.

[71] A. Mainwaring and D. Culler, "Active Messages: Organization and Applications Programming Interface," technical report, UC Berkeley, 1995.

[72] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," *Proc. Third Workshop Runtime Systems for Parallel Programming (RTSPP) of IPPS/SPDP '99*, 1999.

[73] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. Panda, "High Performance Remote Memory Access Comunications: The ARMCI Approach," *Int'l J. High Performance Computing and Applications*, vol. 20, pp. 233-253, 2006.

[74] Aggregate Remote Memory Copy Interface, http://www.emsl.pnl.gov/docs/parsoft/armci, Oct. 2011.

[75] The KeLP Programming System, http://cseweb.ucsd.edu/groups/hpcl/scg/kelp, Oct. 2011.

[76] S.J. Fink, S.R. Kohn, and S.B. Baden, "Efficient Run-Time Support for Irregular Block-Structured Applications," *J. Parallel and Distributed Computing*, vol. 50, pp. 61-82, 1998.

[77] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[78] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, 2005.

[79] Unified Parallel C, http://upc.gwu.edu, Oct. 2011.

[80] R.W. Numrich and J.K. Reid, "Co-Arrays in the Next Fortran Standard," *ACM SIGPLAN Fortran Forum*, vol. 24, pp. 4-17, 2005.

[81] Co-Array Fortran, http://www.co-array.org. Apr. 2011.

[82] http://www.nag.co.uk/sc22wg5/, Oct. 2011.

[83] J. Reid, "Coarrays in the Next Fortran Standard," *ACM SIGPLAN Fortran Forum*, vol. 29, no. 2, pp. 10-27, 2010.

[84] Titanium, http://titanium.cs.berkeley.edu, Oct. 2011.

[85] K.A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P.N. Hilfinger, S.L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," *Proc. ACM Workshop Java for High-Performance Network Computing*, 1998.

[86] X10 Language, http://x10.codehaus.org, Oct. 2011.

[87] J. Muttersbach, T. Villiger, and W. Fichtner, "Practical Design of Globally-Asynchronous Locally-Synchronous Systems," *Proc. Sixth Int'l Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC '00)*, pp. 52-59, 2000.

[88] M. Weiland, "Chapel, Fortress and x10: Novel Languages for hpc," technical report from the HPCx Consortium, 2007.

[89] Chapel Language, http://chapel.cray.com. Oct. 2011.

[90] D. Callahan, B.L. Chamberlain, and H.P. Zima, "The Cascade High Productivity Language," *Proc. Ninth Int'l Workshop High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pp. 52-60, 2004.

[91] Project Fortress, http://projectfortress.java.net, Oct. 2011.

[92] G. Steele, "Fortress: A New Programming Language for Scientific Computing," *Sun Labs Open House*, 2005.

[93] T. Sterling, P. Messina, and P.H. Smith, *Enabling Technologies for Petaflops Computing*. MIT Press, 1995.

[94] C. Wright, "Hybrid Programming Fun: Making Bzip2 Parallel with MPICH2 & pthreads on the Cray XD1," *Proc. CUG*, 2006.

[95] P. Johnson, "Pthread Performance in an MPI Model for Prime Number Generation," *CSCI 4576 - High-Performance Scientific Computing*, Univ. of Colorado, 2007.

[96] W. Pfeiffer and A. Stamatakis, "Hybrid MPI/Pthreads Parallelization of the RAxML Phylogenetics Code," *Proc. Ninth IEEE Int'l Workshop High Performance Computational Biology*, Apr. 2010.

[97] L. Smith and M. Bulk, "Development of Mixed Mode MPI/OpenMP Applications," *Proc. Workshop OpenMP Applications and Tools (WOMPAT '00)*, July 2000.

[98] R. Rabenseifner, "Hybrid Parallel Programming on HPC Platforms," *Proc. European Workshop OpenMP (EWOMP '03)*, 2003.

[99] B. Estrade, "Hybrid Programming with MPI and OpenMP," *Proc. High Performance Computing Workshop*, 2009.

[100] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa, "Combining Message-Passing and Directives in Parallel Applications," *SIAM News*, vol. 32, no. 9, pp. 10-14, 1999.

[101] I.J. Bush, C.J. Noble, and R.J. Allan, "Mixed OpenMP and MPI for Parallel Fortran Applications," *Proc. Second European Workshop OpenMP*, 2000.

[102] P. Luong, C.P. Breshears, and L.N. Ly, "Costal Ocean Modeling of the U.S. West Coast with Multiblock Grid and Dual-Level Parallelism," *Proc. ACM/IEEE Conf. Supercomputing'01*, 2001.

[103] R.D. Loft, S.J. Thomas, and J.M. Dennis, "Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models," *Proc. ACM/IEEE Conf. Supercomputing'01*, 2001.

[104] K. Nakajima, "Parallel Iterative Solvers for Finite-Element Methods Using an OpenMP/MPI hybrid Programming Model on the Earth Simulator," *Parallel Computing*, vol. 31, pp. 1048-1065, 2005.

[105] R. Aversa, B. Di Martino, M. Rak, S. Venticinque, and U. Villano, "Performance Prediction through Simulation of a Hybrid MPI/OpenMP Application," *Parallel Computing*, vol. 31, pp. 1013-1033, 2005.

[106] F. Cappello and D. Etiemble, "MPI Versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," *Proc. Conf. High Performance Networking and Computing*, 2000.

[107] J. Duthie, M. Bull, A. Trew, and L. Smith, "Mixed Mode Applications on HPCx," Technical Report HPCxTR0403, HPCx Consortium, 2004.

[108] L. Smith, "Mixed mode MPI/OpenMP programming," Technical Report Technology Watch 1, UK High-End Computing, EPCC, United Kingdom, 2000.

[109] D.S. Henty, "Performance of hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling," *Proc. ACM/IEEE Conf. Supercomputing'00*, 2000.

[110] E. Chow and D. Hysom, "Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters," Technical Report UCRL-JC-143957, Lawrence Livermore Nat'l Laboratory 2001.

[111] J.C. Thibault and I. Senocak, "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *Proc. 47th AIAA Aerospace Sciences Meeting*, 2010.

[112] S. Jun Park and D. Shires, "Central Processing Unit/Graphics Processing Unit (CPU/GPU) Hybrid Computing of Synthetic Aperture Radar Algorithm," Technical Report ARL-TR-5074, US Army Research Laboratory, 2010.

[113] H. Jang, A. Park, and K. Jung, "Neural Network Implementation using CUDA and OpenMP," *Proc. Digital Image Computing: Techniques and Applications*, pp. 155-161, 2008.

[114] G. Sims, "Parallel Cloth Simulation Using OpenMP and CUDA," thesis dissertation, Graduate Faculty of the Louisiana State Univ. and Agricultural and Mechanical College, 2009.

[115] Y. Wang, Z. Feng, H. Guo, C. He, and Y. Yang, "Scene Recognition Acceleration using CUDA and OpenMP," *Proc. First Int'l Conf. Information Science and Eng. (ICISE '09)*, 2009.

[116] Q. Chen and J. Zhang, "A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA," *Proc. First Int'l Conf. Information Science and Eng. (ICISE '09)*, 2009.

[117] J.C. Phillips, J.E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," *Proc. ACM/IEEE Conf. Supercomputing*, 2008.

[118] H. Schivea, C. Chiena, S. Wonga, Y. Tsaia, and T. Chiueha, "Graphic-Card Cluster for Astrophysics (GraCCA) - Performance Tests," *New Astronomy*, vol. 13, no. 6, pp. 418-435, 2008.

[119] D.A. Jacobsen, J.C. Thibault, and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *Proc. 48th AIAA Aerospace Sciences Meeting*, 2010.

[120] N.P. Karunadasa and D.N. Ranasinghe, "On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters," *Proc. Int'l Conf. High Performance Computing*, 2009.

[121] V. Strassen, "Gaussian Elimination Is Not Optimal," *Numerische Mathematik*, vol. 13, pp. 354-356, 1969.

[122] M.R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *J. Research of the Nat'l Bureau of Standards*, vol. 49, no. 6, pp. 409-436, 1952.

[123] A.E. Walsh, J. Couch, and D.H. Steinberg, *Java 2 Bible*. Wiley Publishing, 2000.

[124] B. Amedro, V. Bodnartchouk, D. Caromel, C. Delbé, F. Huet, and G.L. Taboada, "Current State of Java for HPC," Technical Report RT-0353, INRIA, 2008.

[125] Nas Parallel Benchmarks, http://www.nas.nasa.gov/Resources/\Software/npb.html, Oct. 2011.

[126] R.V. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H.E. Bal, "Ibis: A Flexible and Efficient Java Based Grid Programming Environment," *Concurrency and Computation: Practice and Experience*, vol. 17, pp. 1079-1107, 2005.

[127] G.L. Taboada, J. Touriño, and R. Doallo, "Java for High Performance Computing: Assessment of Current Research and Practice," *Proc. Seventh Int'l Conf. Principles and Practice of Programming in Java (PPPJ '09)*, pp. 30-39, 2009.

[128] A. Shafi, B. Carpenter, M. Baker, and A. Hussain, "A Comparative Study of Java and C Performance in Two Large-Scale Parallel Applications," *Concurrency and Computation: Practice & Experience*, vol. 15, no. 21, pp. 1882-1906, 2010.

[129] B. Blount and S. Chatterjee, "An Evaluation of Java for Numerical Computing," *Scientific Programming*, vol. 7, no. 2, pp. 97-110, 1999.

[130] Java Grande Forum: http://www.javagrande.org/pastglory/index.html, Oct. 2011.

[131] M. Baker, B. Carpenter, S.H. Ko, and X. Li, "mpiJava: A Java Interface to MPI," *Proc. First UK Workshop Java for High Performance Network Computing*, 1998.

[132] A. Shafi, B. Carpenter, and M. Baker, "Nested Parallelism for Multi-Core HPC Systems Using Java," *J. Parallel Distributed Computing*, vol. 69, pp. 532-545, 2009.

[133] G.L. Taboada, S. Ramos, J. Touriño, and R. Doallo, "Design of Efficient Java Message-Passing Collectives on Multi-Core Clusters," *J. Supercomputing*, vol. 55, pp. 126-154, 2011.

[134] High Performance Fortran, http://hpff.rice.edu/index.htm. Oct. 2011.

[135] H. Richardson, "High Performance Fortran: History, Overview and Current Developments," Technical Report TMC-261, Thinking Machines Corporation, 1996.

[136] C.H.Q. Ding, "High Performance Fortran for Practical Scientific Algorithms: An Up-to-Date Evaluation," *Future Generation Computer Systems*, vol. 15, pp. 343-352, 1999.

[137] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *J. Parallel and Distributed Computing*, vol. 37, pp. 55-69, 1996.

[138] Cilk Project, http://supertech.csail.mit.edu/cilk, Oct. 2011.

[139] Intel Cilk Plus, http://software.intel.com/en-us/articles/intel-cilk-plus, Oct. 2011.

[140] B.L. Chamberlain, S.-E. Choi, E.C. Lewis, C. Lin, L. Snyder, and W.D. Weathersby, "ZPL: A Machine Independent Programming Language for Parallel Computers," *IEEE Trans. Software Eng.*, vol. 26, no. 3, pp. 197-211, Mar. 2000.

[141] L. Snyder, "The Design and Development of ZPL," *Proc. Third ACM SIGPLAN History of Programming Languages Conf.*, June 2007.

[142] Zpl Web: http://www.cs.washington.edu/research/zpl/home/index.html, Oct. 2011.

[143] H. Wu, G. Turkiyyahi, and W. Keirouzt, "ZPLCLAW: A Parallel Portable Toolkit for Wave Propagation Problems," *Proc. Am. Soc. of Civil Eng. (ASCE) Structures Congress*, 2000.

[144] Erlang: http://www.erlang.org, Oct. 2011.

[145] S. Vinoski, "Reliability with Erlang," *IEEE Internet Computing*, vol. 11, no. 6, pp. 79-81, Nov./Dec. 2007.

[146] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Jones, "Algorithm+Strategy=Parallelism," *J. Functional Programming,* vol. 8, no. 1, pp. 23-60, 1998.
[147] S. Marlow, S.P. Jones, and S. Singh, "Runtime Support for Multicore Haskell," *ACM SIGPLAN Notices - ICFP '09,* vol. 44, no. 9, pp. 65-78, 2009.
[148] A.S. Tanenbaum and M.V. Steen, *Distributed Systems: Principles and Paradigms,* second ed. Prentice Hall, 2007.
[149] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kauffman, 1998.
[150] B. Wilkinson, *Grid Computing.* Chapman & Hall/CRC, 2010.
[151] gLite, http://glite.cern.ch, Oct. 2011.
[152] "EGEE, http://www.eu-egee.org," Oct. 2011.
[153] S. Reyes, C. Muñoz-Caro, A. Niño, R.M. Badia, and J.M. Cela, "Performance of Computationally Intensive Parameter Sweep Applications on Internet-Based Grids of Computers: the Mapping of Molecular Potential Energy Hypersurfaces," *Concurrency and Computation: Practice and Experience,* vol. 19, pp. 463-481, 2007.
[154] C. Sun, B. Kim, G. Yi, and H. Park, "A Model of Problem Solving Environment for Integrated Bioinformatics Solution on Grid by Using Condor," *Proc. Int'l Conf. Grid and Cooperative Computing (GCC),* pp. 935-938, 2004.
[155] Large Hadron Collider (LHC) Computing Grid Project for High Energy Physics Data Analysis, http://lcg.web.cern.ch/LCG, Oct. 2011.
[156] OMG, http://www.omg.org/, Oct. 2011.
[157] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems* vol. 2, no. 1, pp. 39-59, 1984.
[158] S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments," *IEEE Comm. Magazine,* vol. 35, no. 2, pp. 46-55, Feb. 1997.
[159] M. Henning, "The Rise and Fall of CORBA," *ACM Queue,* vol. 4, pp. 28-34, June 2006.
[160] Y. Gong, "CORBA Application in Real-Time Distributed Embedded Systems," Survey Report, ECE 8990 Real-Time Systems Design, 2003.
[161] CORBA/e, http://www.corba.org/corba-e/index.htm, Oct. 2011.
[162] COM, http://www.microsoft.com/com/default.mspx, Oct. 2011.
[163] ComSource, http://www.opengroup.org/comsource, Oct. 2011.
[164] P. Emerald, C. Yennun, H.S. Yajnik, D. Liang, J.C. Shih, C.Y. Wang, and Y.M. Wang, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer," *C++ Report,* vol. 10, no. 1, pp. 18-29, 1998.
[165] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Springer-Verlag, 2004.
[166] A. Gokhale, B. Kumar, and A. Sahuguet, "Reinventing the Wheel? CORBA vs. Web Services," *Proc. Conf. World Wide Web (WWW '02),* 2002.
[167] SOAP: http://www.w3.org/standards/techs/soap#w3c_all, Apr. 2011.
[168] WSDL, http://www.w3.org/TR/wsdl20/, Oct. 2011.
[169] E. Cerami, "Web Services Essentials. Distributed Applications with XML-RPC, SOAP, UDDI & WSDL, O'Reilly," 2002.
[170] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=\uddi-spec, Oct. 2011.
[171] http://glassfish.java.net, Oct. 2011.
[172] http://www.jboss.org, Oct. 2011.
[173] http://geronimo.apache.org, Oct. 2011.
[174] http://tomcat.apache.org, Oct. 2011.
[175] W.W. Eckerson, "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications," *Open Information Systems,* vol. 10, no. 1, 1995.
[176] www.oracle.com/technetwork/java/javaee/ejb/index.html, Oct. 2011.
[177] *Workflows for e-Science,* I.J. Taylor, E. Deelman, D.B. Gannon, and M. Shields, eds. Springer-Verlag, 2007.
[178] EMBRACE Service Registry: www.embraceregistry.net, Oct. 2011.
[179] A. Sahai, S. Graupner, and W. Kim, "The Unfolding of the Web Services Paradigm," Technical Report HPL-2002-130, Hewlett-Packard, 2002.
[180] T. Earl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice-Hall, 2005.
[181] S. Mulik, S. Ajgaonkar, and K. Sharma, "Where Do You Want to Go in Your SOA Adoption Journey?," *IT Professional,* vol. 10, no. 3, pp. 36-39, May/June 2008.
[182] J. McGovern, S. Tyagi, M. Stevens, and S. Mathew, "Service Oriented Architecture," *Java Web Services Architecture,* Chapter 2, Morgan Kaufmann, 2003.
[183] R.T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD dissertation, Univ. of California, Irvine, 2000.
[184] R.T. Fielding and R.N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Trans. Internet Technology,* vol. 2, no. 2, pp. 115-150, May 2002.
[185] S. Vinoski, "REST Eye for the SOA Guy," *IEEE Internet Computing,* vol. 11, no. 1, pp. 82-84, Jan./Feb., 2007.
[186] ZeroC Ice, www.zeroc.com/ice.html, Oct. 2011.
[187] M. Henning and M. Spruiell Distributed Programming with Ice, ZeroC, 2003, www.zeroc.com/Ice-Manual.pdf, Oct. 2011.
[188] M. Henning, "A New Approach to Object-Oriented Middleware," *IEEE Internet Computing,* vol. 8, no. 1 pp. 66-75, Jan./Feb. 2004.
[189] Scopus, http://www.scopus.com/home.url, Oct. 2011.
[190] "A Call to Arms for Parallel Programming Standards," HPCWire, SC10 Features, Nov. 2010.

Javier Díaz received the PhD degree in computer science from the Castilla-La Mancha University, Spain. He was a member of the QCyCAR-UCLM research group. At present, he is a postdoctoral fellow at the Pervasive Technology Institute of the Indiana University. His research interests are in the areas of cloud computing, computational grids, scheduling algorithms, middleware and virtualization.

Camelia Muñoz-Caro received the PhD degree from the Complutense University in Madrid, Spain. Her work covers Computer Science and Molecular Physics. Formerly, she worked at the National Research Council of Spain (CSIC) and she has been visiting professor at the Brock University (Ontario, Canada). Currently, she is a member of the Department of Information Technologies and Systems at the Castilla-La Mancha University, Spain, leading the SciCom-UCLM research group. Her research interests involve parallel programming models and their application to scientific and engineering problems.

Alfonso Niño received the PhD degree from the Complutense University in Madrid, Spain. His multidisciplinary work involves Computer Science and Molecular Physics. Formerly, he worked at the National Research Council of Spain (CSIC) and he has been visiting professor at the Brock University (Ontario, Canada). Currently, he is member of the Department of Information Technologies and Systems at the Castilla-La Mancha University, Spain and member of the SciCom-UCLM research group. His research interests involve parallel programming models and their application to scientific and engineering problems.