Science of Computer Programming 80 (2014) 65–90

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

An object-oriented parallel programming language for distributed-memory parallel computing platforms

Eduardo Gurgel Pinho, Francisco Heron de Carvalho Junior *

Mestrado e Doutorado em Ciência da Computação, Universidade Federal do Ceará, Brazil

HIGHLIGHTS

- The PObC++ language implements the concept of Object-Oriented Parallel Programming (OOPP).
- OOPP reconciles distributed-memory parallel programming with OO programming principles.
- OOPP separates concerns about inter-object and inter-process communication.
- OOPP makes it possible the encapsulation of distributed-memory parallel computations in objects.
- Performance of PObC++ programs is almost similar to the performance of C++/MPI programs.

ARTICLE INFO

Article history:
Received 6 March 2012
Received in revised form 12 November 2012
Accepted 23 March 2013
Available online 2 April 2013

Keywords:
Object-oriented programming languages
Parallel programming languages
Parallel programming techniques
High performance computing

ABSTRACT

In object-oriented programming (OOP) languages, the ability to encapsulate software concerns of the dominant decomposition in objects is the key to reaching high modularity and loss of complexity in large scale designs. However, distributed-memory parallelism tends to break modularity, encapsulation, and the functional independence of objects, since parallel computations cannot be encapsulated in individual objects, which reside in a single address space. For reconciling object-orientation and distributed-memory parallelism, this paper introduces OOPP (Object-Oriented Parallel Programming), a style of OOP where objects are distributed by default. As an extension of C++, a widespread language in HPC, the PObC++ language has been designed and prototyped, incorporating the ideas of OOPP.

1. Introduction

The better cost-benefit of parallel computing platforms for High Performance Computing (HPC),[1] due to the success of off-the-shelf distributed-memory parallel computing platforms, such as Clusters [1] and Grids [2], has motivated the emergence of new classes of applications from computational sciences and engineering. Besides high performance requirements, these applications introduce stronger requirements of modularity, abstraction, safety and productivity for the existing parallel programming tools [3]. Unfortunately, parallel programming is still hard to incorporate into the usual large scale software development platforms that may be developed to deal with such kinds of requirements [4]. Also, automatic parallelization is useful only in restricted contexts, such as in scientific computing libraries [5]. Skeletal programming [6], a promising alternative for high-level parallel programming, has not achieved the acceptance expected [7]. These days, libraries of message-passing subroutines that conform to the MPI (Message Passing Interface) standard [8] are

* Corresponding author. Tel.: +55 8530880021; fax: +55 8533669837.
E-mail addresses: edgurgel@lia.ufc.br (E.G. Pinho), carvalho.heron@gmail.com, heron@lia.ufc.br (F.H. de Carvalho Junior).

[1] HPC is here defined as a domain of applications with stronger requirements of computation performance, to achieve a result in a minimal space of time, and/or memory, which surpasses the capacity of usual individual computers. Traditionally, they come from engineering and computational sciences, but several examples have emerged from corporative domains.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.03.014

widely adopted by parallel programmers, offering expressiveness, portability and efficiency across a wide range of parallel computing platforms. However, they still present a low level of abstraction and modularity in dealing with the requirements of the emerging large scale applications in HPC domains.

In the context of corporative applications, object-oriented programming (OOP) has been consolidated as the main programming paradigm to promote development productivity and software quality. Object-orientation is the result of two decades of research in programming tools and techniques motivated by the need to deal with the increasing levels of software complexity since the software crisis context of the 1960s [9]. Many programming languages have been designed to support OOP, such as: C++, Java, C#, SmallTalk, Ruby, Objective-C, and so on. Despite their success in the software industry, object-oriented languages are not popular in HPC, dominated by Fortran and C, as a consequence of the high level of abstraction and modularity offered by these languages. When parallelism comes onto the scene, the situation is worse, due to the lack of safe ways to incorporate explicit message-passing parallelism to these languages without breaking important principles, such as the functional independence of objects and their encapsulation.

This paper presents PObC++ (Parallel Object C++), a new parallel extension to C++ which implements the ideas behind OOPP (Object Oriented Parallel Programming), a style of parallel programming where objects are intrinsically parallel, so deployed in a set of nodes of a distributed-memory parallel computer, and communication is distinguished in two layers: *intra-object* communication, for common process interaction by message-passing, and *inter-object* communication, for usual object coordination by method invocations. In OOPP, objects are called *p-objects* (parallel objects). The decision to support C++ comes from the wide acceptance of C++ in HPC. However, OOPP might be supported by other OO languages, such as Java and C#. The main premise that guides the design of PObC++ is the preservation of basic object-orientation principles while introducing a style of programming based on message-passing, inheriting the well-known programming practices using MPI (Message Passing Interface) [8].

Section 2 discusses the current context regarding message-passing parallel programming and object-oriented programming, as well as their integration. Section 3 presents the main premises and concepts behind OOPP, showing how it is supported by PObC++. This section ends off by presenting the overall architecture of the current prototype of PObC++ compiler. Section 4 presents three case studies of PObC++ programming, aimed at giving evidence of the expressiveness, programming productivity, and the performance of OOPP. Finally, Section 5 will present our conclusions, describe ongoing research, and plant ideas for further research initiatives.

2. Context and contributions

This work attempts to bring together two widely accepted programming techniques in a coherent way:

- *Message-Passing* (MP), intended for HPC applications, which have stronger *performance* requirements as the main driving force, generally found in scientific and engineering domains;
- *Object-Orientation* (OO), intended for large-scale applications, which have stronger *productivity* requirements for development and maintenance, generally found in corporative domains.

The following sections review concepts of the two above programming techniques that are important in the context of this work, also providing a discussion about the strategies that have been applied for their integration (related works).

2.1. Parallel programming and message passing with MPI

MPI is a standard specification for libraries of subroutines for message-passing parallel programming that are portable across distributed-memory parallel computing platforms [8]. MPI was developed in the mid 1990s by a consortium integrating representatives from academia and industry, interested in a message-passing interface that could be implemented efficiently in virtually any distributed parallel computer architecture, replacing the myriad of proprietary interfaces developed at that time by supercomputer vendors for the specific features and needs of their machines. It was observed that such diversity results in higher costs for users of high-end parallel computers, due to the poor portability of their applications between architectures from distinct vendors. Also, the lack of standard practices breaks the technical evolution and dissemination of computer architectures and programming techniques for parallel computing. MPI was initially proposed as a kind of parallel programming "assembly", on top of which specific purpose, higher-level parallel programming interfaces could be developed, including parallel versions of successful libraries of subroutines for scientific computing and engineering. However, MPI is now mostly used to develop final applications. The MPI specification is now maintained by the MPI Forum (http://www.mpi-forum-org).

MPI is now the main representative of message-passing parallel programming. Perhaps it is the only parallel programming interface, both portable and general purpose, to efficiently exploit the performance of high-end distributed parallel computing platforms. Since the end of the 1990s, any new installed cluster or MPP[2] has supported some implementation of MPI. In fact, most vendors of parallel computers adopt MPI as their main programming interface, offering

[2] Massive parallel processor.

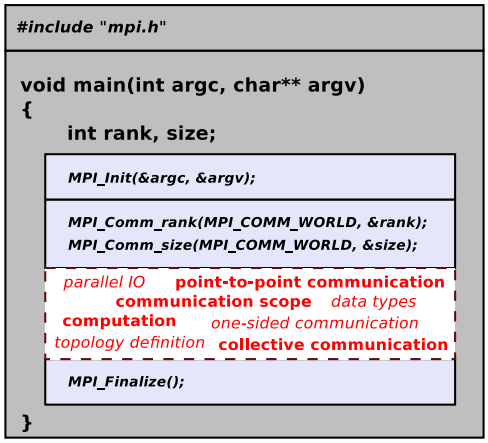

Fig. 1. MPI program.

highly optimized implementations for their architectures. MPI is also considered to be one of the main reasons for the increase in popularity of cluster computing, due to the availability of efficient open-source implementations for Linux-based platforms. MPI has become popular even in shared memory parallel computers, since its wide acceptance among parallel programmers is seen as a way of reducing the learning curve of parallel programming for these architectures.

Many free implementations of MPI, most of them open-source, have been developed, supporting a wide range of computer platforms, such as MPICH, OpenMPI, LAM-MPI, and MSMPI. Also, unofficial versions of the MPI specification in languages not supported by the official specification have also been implemented, such as Boost.MPI (C++), MPI.NET (C#), and JavaMPI (Java).

Two versions of the MPI specification have been proposed by the MPI forum, officially specified in Fortran and C: MPI-1 and MPI-2. MPI-2 extends MPI-1 with many innovations proposed by the community of MPI users. Hundreds of subroutines are supported, with various purposes, enumerated below:

- point-to-point communication (MPI-1);
- collective communication (MPI-1);
- communication scopes (MPI-1);
- process topologies (MPI-1);
- data types (MPI-1);
- one-sided communication (MPI-2);
- dynamic process creation (MPI-2);
- parallel input and output (MPI-2).

For the purposes of this paper, it is relevant to provide only an overview of the MPI programming model. A complete description of their subroutines can be obtained in the official specification document, available at the MPI Forum website. There are also many tutorials publicly available on the web.

2.1.1. MPI programming model

In the original specification, an MPI program is a single program that is executed in each processing node of the parallel computer. This is known as SPMD (Single Program Multiple Data) programming, where processes execute the same computation either over a subset of a distributed data structure (*data parallelism*) or applied to different values of input parameters (*parameter sweep*). Each process is identified by a distinct *rank*, an integer varying between 0 and $size - 1$, where *size* denotes the number of processes.

MPI processes can execute distinct computations over different data structures, using ranks to distinguish processes with different roles. Therefore, MPI implementations also support MPMD (Multiple Program Multiple Data), where the MPI program comprises many programs representing different kinds of processes.

The overall structure of an MPI source code is illustrated in Fig. 1. The header file "mpi.h" contains the prototypes of the MPI subroutines, as well as constant definitions and data type declarations. The calls to MPI_Init and MPI_Finalize must enclose any MPI subroutine call. Respectively, they initialize the MPI environment and free any resource used by the MPI environment during execution before finish. The calls to MPI_Comm_rank and MPI_Comm_size make possible a process to request its identification (*rank*) and for the number of running processes (*size*), respectively. In parallel programs where processes need to refer to each other, e.g. processes that call communication subroutines, the values of *rank* and *size* are essential. MPI_COMM_WORLD is a constant of type MPI_Communicator, representing the global *communicator* (communication scope), which involves all the running processes.

The code enclosed in the dashed rectangle defines the parallel computation, which involves a sequence of computations and calls to MPI subroutines. The subroutines that are relevant for the purposes of this paper are highlighted: point-to-

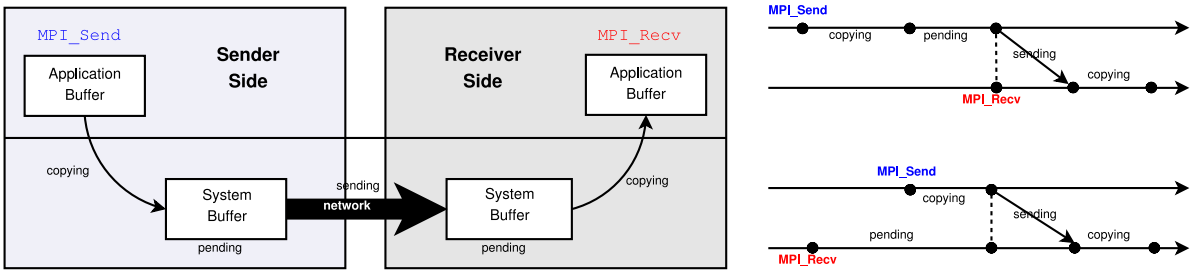

Fig. 2. MPI point-to-point communication semantics.

point/collective communication and communication scopes. In the following sections, we give a brief overview of these subsets of subroutines.

2.1.2. Point-to-point communication

The basic point-to-point communication subroutines are MPI_Recv and MPI_Send, through which a sender process sends a data buffer, including a number of items of a given MPI data type, to a receiver process. Together with MPI_Init, MPI_Finalize, MPI_Comm_size, and MPI_Comm_rank, they form the basic subset of subroutines of MPI, with expressive power to develop any parallel program. MPI also offers a comprehensive set of point-to-point communication routines, with various synchronization semantics.

There are two main communication modes for point-to-point communication subroutines in MPI: blocking and non-blocking. To understand the semantics of these subroutines, it is important to know some details about how MPI communication takes place. This is illustrated in Fig. 2. On the left-hand side, the *application buffer* and the *system buffer* are depicted. The application buffer is passed by the programmer to the sending or receiving subroutine (MPI_Send/MPI_Recv), where the data to be sent or received will be accessible by the program in a contiguous chunk of memory. In fact, it is accessible by a regular variable of the program, which is convenient for programs that perform computations over multidimensional arrays (e.g. in scientific computing). The system buffer is an internal buffer where the data from the application buffer is copied by the receiving operation (sender side) or by the sending operation (receiver side). Thus, it is not accessible by the program. Inconsistencies may occur if the programmer accesses the application buffer before it is copied into the system buffer or directly sent to the receiver. On the right-hand side of Fig. 2, the operations performed in a communication are described, involving the application and system buffers. In the first scenario (top), the send operation is executed before the receive operation. In the second scenario (bottom), it is the receiver that calls the receiving subroutine before the sender. The semantics of the various point-to-point communication subroutines of MPI depends on how they make use of the buffers.

In a blocking communication subroutine, the sender or the receiver returns the control to the caller whenever the access to the application buffer by the program is safe. In blocking receiving, such a status is reached when the received data has been copied to the application buffer. In blocking sending, it is reached after sending data has been safely copied to the system buffer or directly sent to the receiver. Blocking sending has three modes: *synchronous* (MPI_Ssend), *buffered* (MPI_Bsend), and *ready* (MPI_Rsend). In synchronous blocking sending, there is no system buffer. The data is sent to the receiver directly from the application buffer. Thus, MPI_Ssend only completes after a matching MPI_Recv has been executed at the receiver side. In buffered blocking sending, there is a system buffer allocated by the programmer using the MPI_Attach_buffer subroutine, which must be large enough for storing pending calls to MPI_Bsend. If the buffer is full in a call to MPI_Bsend, i.e. there are many pending calls, the call behaves like a synchronous blocking sending. In a ready blocking sending, there is neither a system buffer neither at the receiver side nor at the sender side. Thus, there must be a pending matching call to MPI_Recv when a MPI_Rsend is called.

In the MPI implementations, MPI_Send is a buffered blocking sending, with a small buffer pre-allocated by the MPI environment. For this reason, it behaves like synchronous blocking sending for large messages. The official MPI document does not specify the semantics of MPI_Send to be followed by implementers.

For each blocking subroutine, there is a corresponding non-blocking one, the name of which differs by the prefix "I" (e.g. MPI_Irecv, MPI_Isend, MPI_Issend, MPI_Ibsend, MPI_Irsend). Non-blocking subroutines do not wait until the access to the application buffer is safe before returning the control to the caller. The control is returned to the caller with a request handle, a value of type MPI_Request. The communication occurs in the background, making it possible to overlap useful computation with communication, minimizing synchronization overheads and avoiding certain potential deadlocked states. The programmer is responsible for taking care of preventing corrupting accesses to the application buffer until the communication completes. A set of subroutines exists for testing and waiting for the completion of one or more requests (MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome, MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome). For this, the programmer must use the request handles returned by the non-blocking communication subroutines to refer to the pending operations it is interested in synchronizing.

2.1.3. Collective communication

MPI supports subroutines that encapsulate communication operations involving many processes, often found in parallel algorithms. There are operations for disseminating data from one process to the other ones (MPI_Bcast and MPI_Scatter) and others for collecting data from all the processes to one process (MPI_Gather), possibly executing a reduction operation on the sets of data received (MPI_Reduce). Also, there are operations where all the involved processes disseminate data to each other and, as a consequence, must collect (reduce) the data received (MPI_Alltoall, MPI_Allgather, MPI_Allreduce). Some operations are personalized (MPI_Scatterv, MPI_Alltoallv, MPI_Allgatherv, MPI_Allreducev), which means that the sender processes have a different piece of data addressed to each other's process. Prefix operations are also supported (MPI_Scan, MPI_Reducescan).

2.1.4. Groups and communicators

All the above communication operations execute in the context of a communicator. The most important one is MPI_COMM_WORLD, which involves all the running processes of the parallel program. By using the abstraction of *process groups*, an MPI programmer may create communicators involving arbitrary subsets of processes. Communication scopes has been originally proposed for avoiding interference between communication operations performed by different subroutines of parallel scientific libraries, possibly distinct, in message-passing parallel programs.

2.2. Principles of object-oriented languages

Object-orientation is an influential data abstraction mechanism whose basis was introduced in the mid 1960s, with the Simula'67 programming language [10,11]. Following Simula'67, the most prominent object-oriented language was Smalltalk [12], developed at Xerox PARC in the 1970s. The designers of Smalltalk adopted the pervasive use of objects as a computation basis for the language, being the first to coin the term object-oriented programming (OOP). During the 1990s, OOP became the mainstream in programming, mostly influenced by the rising in popularity of graphical user interfaces (GUI), where OOP techniques were extensively applied. However, the interest in OOP rapidly surpassed the use in GUI's, as the software engineers and programmers recognized the power of OOP principles in dealing with the increasing complexity and scale of software. Today, the most used OOP languages are C++, Java, and C#.

Modern object-oriented languages are powerful programming artifacts. Often, their rich syntax, complex semantics, and comprehensive set of libraries hide the essential principles of object-orientation. In this section, we present the essential characteristics of object-oriented imperative languages, in its pure sense, focusing on the ones that are relevant for the purposes of this paper.

2.2.1. Objects

In an imperative programming setting, a pure *object* is a runtime software entity consisting of the following parts:

- a *state* defined by a set of internal objects called *attributes*;
- a set of subroutines called *methods*, which define the set of valid *messages* the object may accept.

The methods of an object define the object's valid state transformations, which define the computational meaning of the object. The signatures of the methods of an object form the object's *interface*.

An object-oriented program comprises a set of objects that coordinate their tasks by exchanging messages in the form of method invocations. From the software architecture point of view, each object addresses a concern in the dominant decomposition of the application. Thus, coordination of objects results in the implementation of the overall application concern.

Concerns in software engineering. The different programming paradigms created in the last decades primarily tried to break the complexity of a problem by recursively dividing it into a set of smaller subproblems that can be easier to be understood and solved. This is software modularization. From this perspective, a software is recursively broken into a set of software modules, whose relation and interaction are specified. Each module must address a *software concern*, which is defined as a conceptual part of a solution such that the composition of concerns may define the solution needed by the software [13]. The modularization process based on concerns is called separation of concerns (SoC). The concrete notion of module in a programming environment depends on the programming paradigm. For example, object-oriented programming uses objects to describe concerns, whereas functional programming uses functions in a mathematical sense.

2.2.2. Encapsulation

The most primitive principle behind object-oriented programming (OOP) is *encapsulation*, also called *information hiding*, which states that an object which knows the interface of another object does not need to make assumptions about its internal details to use its functionality. It only needs to concentrate on the interface of the objects they depend on. In fact, an OOP language statically prevents an object from accessing the internal state of another object, by exposing only its interface.

Encapsulation prevents programmers from concentrating on irrelevant details about the internal structure of a particular implementation of an object. In fact, the implementation details and attributes of an object may be completely modified

without affecting the parts of the software that depend on the object, provided its interface, as well as its behavior, is preserved. In this sense, encapsulation is an important property of OOP in dealing with software complexity and scale. More importantly, encapsulation brings to programmers the possibility of working at higher levels of *safety* and *security*, by allowing only essential and valid accesses to be performed on critical subsets of the program state.

2.2.3. Classes

A *class* is defined as a set of similar objects, presenting a set of similar *attributes* and *methods*. Classes may also be introduced as the programming-time counterparts of objects, often called prototypes or templates, specifying the attributes and methods that objects instantiated from them must carry at run time.

Let A be a class with a set of attributes α and a set of methods μ. A parallel programmer may derive a new class from A, called A', with a set of attributes α' and a set of methods μ', such that $\alpha \subseteq \alpha'$ and $\mu \subseteq \mu'$. This is called *inheritance* [14]. A is a *superclass* (generalization) of A', whereas A' is a *subclass* (specialization) of A. By the *substitution principle*, an object of class A' can be used in a context where an object of class A is required. Thus, in a good design, all the valid internal states and state transformations of A are also valid in A'. Such safety requirement cannot be enforced by the usual OOP languages.

Inheritance of classes can be single or multiple. In *single inheritance*, a derived class has exactly one superclass, whereas in *multiple inheritance* a class may be derived from a set of superclasses. Modern OOP languages, such as Java and C#, have abolished multiple inheritance, still supported by C++, by adopting the single inheritance mechanism once supported by Smalltalk. To deal with use cases of multiple inheritance, Java introduced the notion of *interface*. An interface declares a set of methods that must be supported by objects that implement it. Interfaces define a notion of *type* for objects and classes.

2.2.4. Abstraction

Classes and inheritance bring four important abstraction mechanisms to OOP [14]:

- *Classification/instantiation* constitutes the essence of the use of classes. As already defined, classes group objects with similar structure (methods and attributes). Objects represent *instances* of classes.
- *Aggregation/decomposition* comes from the ability to have objects as attributes of other objects. Thus, a concept represented by an object may be described by their constituent parts, also defined as objects, forming a recursive hierarchy of objects that represent the structure behind the concept.
- *Generalization/specialization* comes from inheritance, making it possible to recognize commonalities between different classes of objects by creating superclasses from them. Such an ability makes possible a kind of *polymorphism* that is typical in modern OO languages, where an object reference, or variable, that is typed with a class may refer to an object of any of its subclasses.
- *Grouping/individualization* is supported due to the existence of collection classes, which allows for the grouping together of objects with common interests according to the application needs. With polymorphism, collections of objects of related classes, by inheritance relations, may be valid.

2.2.5. Modularity

Modularity is a way of managing complexity in software, by promoting the division of large scale and complex systems into collections of simple and manageable parts. There are some accepted criteria in classifying the level of modularity achieved by a programming method: *decomposability*, *composability*, *understandability*, *continuity*, and *protection* [15].

OOP promotes the organization of the software in *classes* from which the objects that perform the application will be instantiated at run time. In fact, classes will be the building blocks of OOP software. In a good design, classes capture simple and well-defined concepts in the application domain, orchestrating them to perform the application in the form of objects (*decomposability*). Classes promote the reuse of software parts, since the concept captured by a class of objects may be present in several applications (*composability*). Indeed, abstraction mechanisms makes it possible to reuse only those class parts that are common between objects in distinct applications. Encapsulation and a high functional independence degree promote independence between classes, making it possible to understand the meaning of a class without examining the code of other classes it depends on (*understandability*). Also, they avoid the propagation of modifications in the requirements of a given class implementation to other classes (*continuity*). Finally, exception mechanisms makes it possible to restrict the scope of the effect of an error condition at runtime (*protection*).

2.2.6. Functional independence

Functional independence is an important property of objects to be achieved in the design of their classes. It is a measure of the independence among the objects that constitute the application. It is particularly important for the purposes of this paper. Functional independence is calculated by two means: *cohesion* and *coupling*. The cohesion of a class measures the degree to which the tasks its objects perform define a meaningful unit. Thus, a highly cohesive class addresses a single and well-defined concern. The coupling of a class measures its degree of dependency in relation to other classes. Low coupling means that modifications in a class tend to cause minor effects in other classes they depend on. Also, low coupling minimizes propagation of errors from defective classes to the classes they depend on. From the discussion above, we may conclude that functional independence is better as high cohesion and low coupling are achieved.

2.3. Parallelism support in object-oriented languages

As a result of the wide acceptance of MPI among parallel programmers in HPC domains, implementations of MPI for object-oriented languages have been developed. Some of these implementations are simple wrappers for native MPI implementations [16–18], whereas others adopt an object-oriented style [19–21]. Both approaches present two drawbacks. First of all, they go against the original MPI designer's intention to serve as just a portability layer for message-passing in parallel implementations of specific purpose scientific libraries. Secondly, MPI promotes the decomposition of a program in the dimension of processes, causing the breaking of concerns in a set of cooperating objects, or classes, increasing their coupling and, as a consequence, sacrificing functional independence.

Charm++ [22] is a library for message-passing on top of C++, portable across distributed and shared-memory parallel architectures. A program is built from a set of parallel processes called *chares*, which can be dynamically created by other *chares*. They can communicate via explicit message-passing and share data through special kinds of objects. The placement of *chares* is defined by a dynamic load balancing strategy implemented in the run-time system. Chares are special objects, bringing the drawback of using objects to encapsulate processes instead of concerns.

A common approach, supported by JavaParty [23], ParoC++ [24], POP-C++ [25], relies on deploying objects, representing processes, across the nodes of the parallel computer, where they can interact through method invocations instead of message-passing. Indeed, such an approach may be supported by any OO language with some form of remote method invocation. Despite avoiding the backdoor communication promoted by raw message-passing, method invocations promote client–server relations between the objects that act as processes, whereas most of the parallel algorithms assume peer-to-peer relations among processes. For this reason, ParoC++ proposed forms of asynchronous method invocations in order to improve the programmability of common process interaction patterns. POP-C++ extended ParoC++ for grid computing.

With the advent of virtual execution machines, another common approach is to implement parallel virtual machines, where parallelism is managed implicitly by the execution environment [26]. However, we argue that such implicit approaches will never reach the level of performance supported by explicit message-passing in the general case, since efficient and scalable parallel execution depends on specific features of the architectures and applications, such as the strategy of distributing the data across nodes of a cluster in order to promote data locality and minimize the amount of communication.

Another parallel programming model that has been proposed for object-oriented languages is PGAS (Partitioned Global Address Space), supported by languages like X10 [27], Chapel [28], Fortress [29], and Titanium [30]. Most of these languages have been developed under the HPCS (High Productivity Computer Systems) program of DARPA [31], since 2002. The HPCS program has two goals: to boost the performance of parallel computers and increment their usability. In PGAS, the programmer can work with shared and local variables without explicitly sending and receiving messages. Thus, there is neither a notion of parallel object nor a notion of message-passing interaction between objects, like in the other approaches and the approach proposed in this paper. Objects interact through the partitioned address space, despite being placed in distinct nodes of the parallel computer. Such an approach makes parallel programming easier, but may incur in performance penalties since memory movements are controlled by the system. Each language has its own way of expressing task and data parallelism, through different forms, such as: asynchronous method invocation, explicit process spawn, dynamic parallelism to handle "for loops" and partitioned arrays.

2.4. Contributions

From the above context, the authors argue that attempts to reconcile distributed-memory parallel programming and object-oriented languages break important object-orientation principles and/or do not reach the level of flexibility, generality and high performance of parallel programming using the MPI standard. Concerning these problems, this paper includes the following contributions:

- An alternative perspective of object-oriented programming where objects are parallel by default, called OOPP (Object Oriented Parallel Programming);
- The design of a language based on OOPP, called PObC++ (Parallel Object C++), demonstrating the viability of OOPP as a practical model;
- A prototype of PObC++, which may be used to validate OOPP usage and performance;
- A comparison between the performance of a PObC++ program and the performance of its C++/MPI (non-OO) counterpart, which evidences that object-orientation does not add significant overheads;
- Discussions about programming techniques behind OOPP, using a set of selected case studies.

3. Object-Oriented Parallel Programming (OOPP) and PObC++

We found that the main reason for the difficulties in reconciling object-orientation and distributed-memory parallel programming lies in the usual practice of mixing concerns and processes in the same dimension of software decomposition [32]. In distributed-memory parallel programming, concerns that must be implemented by a team of processes (*parallel concerns*) are common. Consequently, an individual object, which addresses some application concern, must be distributed, which means that they must be located at a set of nodes of the parallel computing platform. In the usual practice, an object is always located in the address space of a single node, and teams of objects are necessary to address parallel concerns.

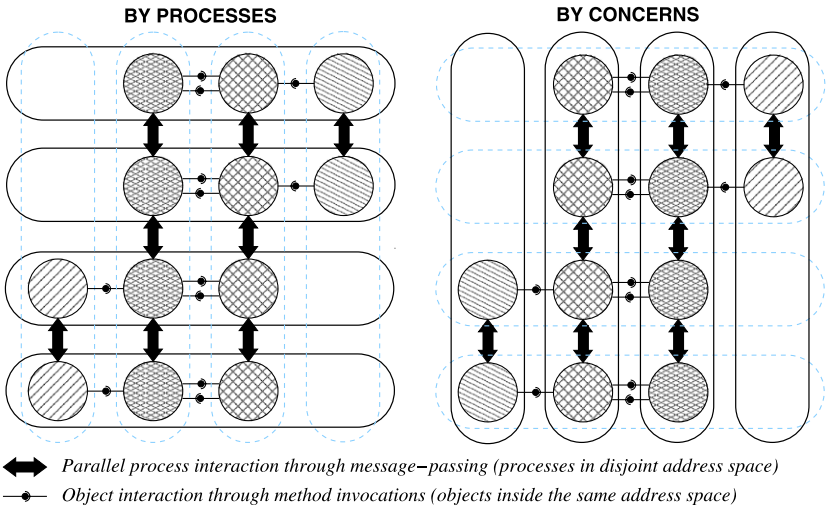

Fig. 3. Process vs. object perspectives.

On the left-hand side of Fig. 3 ("BY PROCESSES"), the common practice of parallel programming in OOP languages is illustrated, where individual objects execute in a single address space and parallel concerns are implemented by teams of objects that communicate through either message-passing or remote method invocations. In the latter approach, there is no clear distinction between messages for parallel interaction and object coordination. Moreover, these kinds of client–server relations are not appropriate for communication among parallel interacting peers. In the former approach, parallel interaction is a clandestine form of object coordination, by using some low-level communication library, such as MPI or Sockets, possibly breaking the encapsulation of objects and reducing their functional independence.

On the right-hand side of Fig. 3 ("BY CONCERNS"), the practice that we argue to be the best suited one for distributed-memory parallel programming with OOP languages is illustrated. It is the base of the Object-Oriented Parallel Programming (OOPP), the approach we are proposing. Objects that cooperate to implement a parallel concern now constitute a *parallel object*, here referred to as *p-object*. Each individual object is a *unit* of the *p-object*. Application concerns are now encapsulated in a *p-object*, where parallel interactions are no longer clandestine. In fact, parallel interaction and object coordination are distinguished at different hierarchical levels, leading to the concepts of *intra-object* and *inter-object* communication. Intra-object communication may be performed using message-passing, which is better suited for parallel interaction between peer units, whereas inter-object communication may use local method invocations.

From the above considerations, we argue that a fully **concern-centric** decomposition approach improves the *functional independence* of objects, now parallel objects, by eliminating the additional *coupling* of objects and classes which result from a **process-centric** decomposition approach. We propose a language for OOPP, called PObC++, a parallel extension to C++, implemented on top of MPI for enabling process creation, communication, and synchronization. C++ is adopted because it is widely accepted and disseminated among parallel programmers due to its high performance, mainly in computational sciences and engineering. However, the parallel concepts introduced in C++ may be easily introduced to Java or C#, the two mainstream programming languages in general application domains.

PObC++ supports a parallel programming style inspired by the MPI standard, which has been widely accepted among HPC programmers since the mid 1990s. It may be distinguished from other object-oriented parallel programming alternatives in the following aspects:

- objects keep atomicity of concerns, since each unit of a *p-object* can address the role of a process with respect to a concern that is implemented in parallel;
- objects send messages to other objects only by usual method invocations, avoiding clandestine communication through low-level message passing as in existing approaches;
- fully explicit parallelism is supported by means of an explicit notion of process and intra-object message-passing communication, providing full control over typical parallel programming responsibilities (load balancing, data locality, and so on).

The following subsections introduce the main concepts and abstractions behind OOPP, using simple examples in PObC++. PObC++ attempts to reconcile full explicit message-passing parallel programming with object-orientation, by introducing the smallest possible set of new concepts and abstractions. For this reason, pragmatic decisions have been made for supporting MPI without breaking the principles behind OOPP. We think that such an approach may lead to a better learning curve for new users of PObC++. Further works will study how to improve OOPP, making it more attractive to parallel programmers.

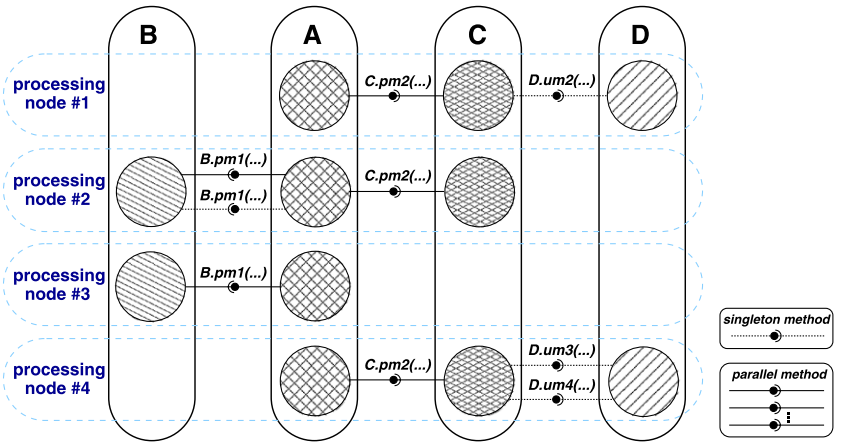

Fig. 4. Parallel and unit method invocations.

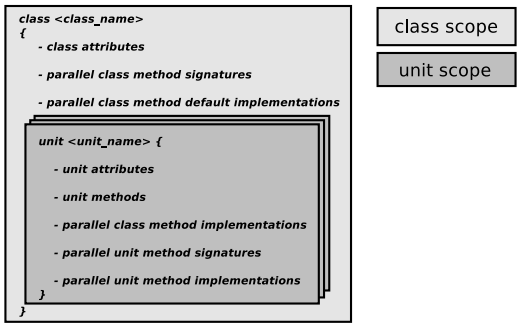

Fig. 5. Parallel class.

3.1. Parallel objects

A Parallel Object (*p-object*) is defined by a set of *units*, each one located at a processing node of a distributed-memory parallel computer. A *p-object* is an object in the pure sense of object-oriented programming, addressing some application concern and communicating with other objects through method invocations. Distinct *p-objects* of an application may be located at distinct subsets of processing nodes of the parallel computer, overlapped or disjoint.

The state of a *p-object* (*global state*) is defined by a set whose elements are the states of each one of its units (*local states*). Local states are defined just as the states of single objects (Section 2.2).

A *p-object* may accept parallel methods and singleton methods. Singleton methods are accepted by individual units of the *p-object*. In turn, a parallel method is accepted by a subset of the units of the *p-object*. Let *A* and *B* be *p-objects*, such that units of *A*, the *caller* units, performs an invocation to a parallel method *m* of *B*, the *callee* units. Each caller unit of *A* must be located at the same processing node of the callee unit of *B*. In a parallel method invocation, message-passing operations may be used for synchronization and communication between the callee units of the *p-object*.

Fig. 4 illustrates parallel method invocations (solid arrows) and singleton method invocations (dashed arrows). The calls to ***um*1**, from **A** to **B**, and ***um*2**, ***um*3**, and ***um*4**, from **C** to **D**, illustrate singleton method invocations. The *p-object* **A** performs calls to the parallel methods ***pm*1** and ***pm*2**, respectively accepted by the *p-objects* **B** and **C**. Notice that both **B** and **C** are located at a subset of the processing nodes where **A** is located, in such a way that the pairs of units involved in a method call (a_2/b_1, a_2/c_2, a_3/b_2, and a_4/c_3) are placed in the same processing node. Therefore, method invocations are always local, involving units of distinct *p-objects* placed inside the same processing node. This is *inter-object communication*, which makes possible coordination between the objects for achieving application concerns. In turn, inter-process communication is always encapsulated inside a *p-object*, by message-passing among its units. This is *intra-object communication*, which aims to implement inter-process communication patterns of parallel programs.

3.2. Classes of parallel objects

A Parallel Class (*p-class*) represents a prototype for a set of *p-objects* with common sets of units, methods and attributes, distinguished only by their execution state. Fig. 5 illustrates the structure of a parallel class in POb C++, introducing the

```
class Hello1 {

 /* parallel class method signature */
 public: void sayHello();

 unit a {
    /* parallel class method implementation */
    void sayHello() {
        cout << "Hello ! I am unit a";
    }
 }

 unit b {
    /* parallel class method implementation */
    void sayHello() {
        cout << "Hello ! I am unit b";
    }
 }

 parallel unit c {
    /* parallel class method implementation */
    void sayHello()[Communicator comm] {
        int rank = comm.getRank();
        cout << "Hello ! I am the "
             << rank << "-th unit c"
    }
 }

 parallel unit d {
     /* parallel class method implementation */
     void sayHello() {
        int rank = comm.getRank();
        cout << "Hello ! I am the "
             << rank << "-th unit d";
     }
     public:
        /* parallel unit method implem. */
        parallel void
          sayBye()[Communicator my\_comm] {
            int rank = my\_comm.getRank();
            cout << "Bye ! I am the "
                 << rank << "-th unit d";
        }
 }

}
```

(a)

```
class Hello2 {

 private: int i; // class attribute

 /* parallel method with default
    implementation outside class */
 public:
    void sayHello();

 unit a {
    public: double n1;  // unit attribute
 }

 unit b {
    private: double n2;  // unit attribute
    public: double n1;   // unit attribute
 }

 unit c {
    public:
        double n1;   // unit attribute
        /* singleton unit method */
        int getMy_i() {
            return i++;
        }
 }
}

/* parallel method default implementation */
void Hello2::sayHello() {
   cout << "I am some unit of p-class Hello";
}

/* parallel method implementations */
void Hello2::b::sayHello() {
   cout << "Hello ! I am unit b";
}

void Hello2::c::sayHello() {
   cout << "Hello ! I am unit c";
}
```

(b)

Fig. 6. Examples of *p-classes*.

unit abstraction. Also, it introduces the possible syntactical scopes in a *p-class* declaration: *class scope* and *unit scope*. Fig. 6 exemplifies the PObC++ syntax for *p-classes*.

Units of a *p-class* may be *singleton units* or *parallel units*. In the instantiation of a *p-class*, only one instance of a singleton unit will be launched in a processing node, whereas an arbitrary number of instances of a parallel unit may be launched, each one in a distinct process node. A reader who is familiar with parallel programming will find that parallel units capture the essence of SPMD programming.

An attribute may be declared in the unit scope or in the class scope. In the former case, it is called a *unit attribute*, whose instance must exist only in the address space of the unit where it is declared. In the latter case, it is called a *class attribute*, which must have an independent instance in the address space of each unit of the *p-object*. In fact, a class attribute is a syntactic sugar for a unit attribute declared in each unit of the *p-object* with the same type and name.

Methods may also be declared in the class scope or in the unit scope. In the former case, they are *parallel class methods*, which are accepted by all the units of a *p-object*. In the latter case, they are either *singleton unit methods*, which are accepted by individual units, or *parallel unit methods*, which are accepted by parallel units. Parallel unit methods are declared in the scope of parallel units using the `parallel` modifier.

The reader may notice that *parallel class methods* and *parallel unit methods* correspond to the *parallel methods* of *p-objects* discussed in Section 3.1, where communication and synchronization between parallel objects takes place. In turn, their *singleton methods* relates to the *singleton unit methods* of PObC++ classes. To enforce these restrictions in programming, only *parallel methods* have access to *communicators* in their scope. A communicator is a special kind of object that provides an interface for communication and synchronization among units of a *p-object* in the execution of parallel methods.

An implementation of a parallel class method, possibly distinct, must be provided in the scope of each unit of the *p-class*. Alternatively, a default implementation may be provided in the class scope, which may be overridden by specific

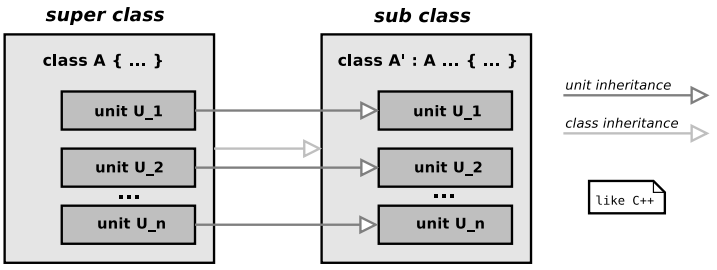

Fig. 7. Inheritance between *p-classes*.

implementations provided in the scope of one or more units. Default implementations of parallel class methods have access only to class attributes, whereas methods declared in the unit scope, singleton or parallel, may access class and unit attributes.

In Fig. 6(a), the *p-class* Hello1 declares four units: *a*, *b*, *c*, and *d*. The first two are singleton units, whereas the last two are parallel ones. The parallel keyword announces a parallel unit. Hello1 has a parallel class method, named sayHello, without a default implementation. Therefore, it is implemented by each unit. There is also an example of parallel unit method, named sayBye, declared by unit *d*. The code of sayBye makes reference to a communicator object received as a special argument using the brackets notation, named `my_comm`. In the code of sayHello, the declaration of the communicator argument is implicit. In such case, it may be referenced using the keywork `comm`. Implicit declarations of communicator objects is a kind of syntactic sugar, since most of parallel methods will use a single communicator object. In a call to a parallel method, the communicator object may also be passed implicitly if there is only one communicator object, implicitly defined, in the scope. Section 3.3 will provide more details about the semantics and use of communicators.

The *p-class* Hello2, in Fig. 6(b), illustrates attributes and methods declared in class and unit scopes. For instance, a copy of the class attribute *i* exists in each unit *a*, *b*, and *c*. They are independent variables, and can be accessed and updated locally, by class and unit methods. Notice that a double precision float-point unit attribute $n1$ is declared in each unit scope. $n2$ is another unit attribute, but accessible only in the scope of unit *b*. The parallel class method sayHello now has a default implementation, defined outside the class as recommended by C++ programming conventions (C++ syntax also allows definitions inside the class). The default implementation of sayHello is overridden by specialized implementations in units *b* and *c*, also defined outside the class declaration. Indeed, only the unit *a* will execute the default implementation of sayHello. Finally, the method getMy_i is a singleton unit method in the scope of *c*. Thus, it has access to the class attribute *i* and to the unit attribute $n1$.

3.2.1. Inheritance

Inheritance between *p-classes* in PObC++ is similar to inheritance between usual C++ classes. The unique peculiarity is the requirement of *unit preservation*, which states that the units of a subclass are all those inherited from the superclass. This is illustrated in Fig. 7.

Adding or removing units of a superclass may violate the *type substitution principle* of type systems that support subtyping, which state that an object of a class may be used in any context where an object of one of its superclasses is required. For instance, let *A* be a *p-class* with *n* distinct units and let A' be another *p-class* inherited from *A* by introducing an additional unit, distinct from the other ones. Thus, A' has $n + 1$ distinct units. Now suppose that a *p-class B*, having *n* distinct units, declares a class attribute v of type *A*. In execution, a *p-object* of *B* instantiates a *p-object* of *A* for the attribute v, by calling the constructor of each unit of *A* in each one of the *n* units of *B*. Now, suppose that one decides to replace the *p-object* of *A* by a *p-object* of A' for the variable v. According to the *type substitution principle*, this would be a safe operation, since A' is a subtype of *A*, a consequence of inheritance. But, this is not possible in the described scenario, since the *p-object* of A' has $n + 1$ units, one more unit than the old *p-object* of *A*. Therefore, it is not possible to determine a distinct processing node inside the *p-object* of *B* to place the additional unit.

3.2.2. Scalable p-objects and parallel units

Scalability is a crucial property to be achieved in the design of efficient parallel programs, with concerns from algorithm design to implementation and tuning for a given parallel computer [33]. The syntax of a parallel programming language supports scalable parallel programs if it makes it possible to write parallel programs that can execute in an arbitrary number of processing nodes without recompilation. In most parallel programming platforms, a scalable parallel program is composed of groups of processes that run the same program. Ranks are used to index processes, making it possible to distinguish between processes in the group. This is the MPMD style supported by MPI libraries (Section 2.1). The number of processes may be defined just before execution. This is the reason why parallel programming artifacts must provide means for the specification of an arbitrary number of processes.

As exemplified before, units may be declared as parallel units using the parallel modifier to make possible scalable *p-objects* in PObC++ possible. During execution of a parallel unit method, the identification of a single unit and the number

of units actually instantiated for the parallel unit, as well as information about the topological organization of such units, may be fetched by invoking the methods of one or more *communicators* that may be provided by the caller, as illustrated in the example introduced in the next section. This is also valid for parallel class methods.

3.2.3. A simple example of parallel class

In Fig. 8(a), the *p-class* MatrixMultiplier declares a unit named manager and a parallel unit named cell, aimed at implementing the well-known systolic matrix multiplication algorithm [34]. In the following discussion, we will refer to these units as the manager and the cells, respectively.

Through an invocation to the distribute parallel class method, the manager distributes the elements of the two input square $n \times n$ matrices, named a and b, among a team of $n \times n$ cells. For that, all the units engaged in the parallel invocation of distribute, i.e. the manager and the cells, perform invocations to the collective communication operation `scatter` of the communicator (`comm`). After this operation, each cell has corresponding elements at position (i, j) of a and b in variables `a` and `b`.

The execution of the matrix multiplication systolic algorithm is performed by an invocation to the compute parallel unit method of the cells. It is worth noticing that the communicator expected in an invocation to the compute parallel method is cartesian, since it is typed by CartesianCommunicator, ensuring that the cell unit instances are organized according to a square mesh topology, as required by the systolic algorithm to work correctly. In compute, the cells first of all exchange their elements to realignment of data, using `isend` and `recv` point-to-point communication operations of `comm`. After that, the cells engage in a loop of n iterations, where, at each iteration, each cell accumulates the product of `a` and `b` in the variable `c`, sends the current values of `a` and `b` to its east and south neighbor and receives new values of `a` and `b` from the west and north neighbors, respectively. At the end of the operation, each cell at position (i, j) has traversed all the elements of the i-th row of matrix a and j-th column of the matrix b, accumulating the sum of the corresponding values in `c`. Thus, `c` contains the resulting element of matrix c.

Finally, the collect parallel class method is invoked for accumulating the values of the `c` variables in cells into the manager, through the `gather` communication operation of `comm`.

We will return to this example in further sections, for illustrating other POb C++ constructions.

3.3. Communication and synchronization

In OOPP, the orthogonalization between *concerns*, encapsulated in *p-objects*, and *processes*, results in a clean separation between two types of messages:

- *inter-object communication*: messages exchanged between parallel objects, implementing the orchestration among the set of application concerns, concretely carried out by *p-objects*, in order to implement the overall application concern. In general, such messages are carried out by means of method invocations, defining a client–server relationship between *p-objects*;
- *intra-object communication*: messages exchanged among the units of *p-objects*, usually by means of message-passing, defining peer-to-peer relationships among units of a *p-object*. Such messages define the interactions among application processes, required by most of parallel algorithms.

In the usual parallel approaches of OOP languages, there is no clear distinction between these kinds of messages. As a consequence, one of the following approaches is adopted:

- parallel synchronization and communication are implemented by means of method invocations between objects, which is inappropriate for parallel programming, since method invocations lead to client–server relationships between pairs of processes, or pairs of subsets of processes, whereas most of the parallel algorithms assume peer-to-peer relationships among them; or
- low-level message passing between objects, defining a communication backdoor for clandestine interaction between objects, resulting in low modularity and high coupling among the objects that implement a parallel concern.

The well-known $M \times N$ coupling problem [35] leads to convincing arguments about the inappropriateness of the first approach. Let M and N be two sets of processes residing in disjoint sets of processing nodes, probably with different cardinalities, that want to communicate some data structure whose distribution differ in the two sets of processes. If each process is implemented as an object, following the usual "BY PROCESSES" perspective of Fig. 3, the logic of the communication interaction needed to exchange data between the objects of each set tends to be scattered across all the objects, with some objects making the role of the client side and others playing the role of the server side. Moreover, many methods may be necessary to control the data transfers back and forth between the two sides of the $M \times N$ coupling of processes. Using OOPP, such coupling could be simply implemented by a single *p-object* with $M + N$ units that encapsulate all the coupling logic using peer-to-peer message-passing communication operations.

With respect to the second approach, objects tend to lose their functional independence. Therefore, they cannot be analyzed in isolation, breaking important modularity principles behind object-orientation that were introduced in Section 2.2.

```
class MatrixMultiplier
{
   public: void distribute();
           int* collect();

   unit manager {
      private:
         int *a, *b, *c, n;
      public:
         void set_ab(int n_, int *a_, int *b_)
         { n = n_; a = a_; b = b_; }
   }

   parallel unit cell {
      private: int a, b, c = 0;
               int i, j, n;
               void calculate_ranks_neighbors
                     (CartesianCommunicator,
                      int, int, int*, int*,
                      int*, int*);
      public: parallel int* compute();
   }
 }

class Main
{
  public:
    int main();

  private:
    Communicator comm_data;
    CartesianCommunicator create_comm_compute();

  unit root
  {
     int main()[Communicator world_comm]
     {
        MatrixMultiplier::manager *mm
            = new MatrixMultiplier::manager();

        comm_data = world_comm.clone();
        create_comm_compute();

        mm->distribute()[comm_data];
        c = mm->collect()[comm_data];
     }
  }

  parallel unit peer
  {
     private:
        CartesianCommunicator comm_compute;

     int main()[Communicator world_comm]
     {
        MatrixMultiplier::cell *mm
            = new MatrixMultiplier::cell();

        comm_data = world_comm.clone();
        comm_compute = create_comm_compute();

        mm->distribute()[comm_data];
        mm->compute()[comm_compute];
        mm->collect()[comm_data];
     }
  }
}
```

(a)

```
void MatrixMultiplier::manager::distribute()
          [Communicator comm]
{
   int fool_a, fool_b;
   comm.scatter(a-1,&fool_a,rankof(manager));
   comm.scatter(b-1,&fool_b,rankof(manager));
}

int* MatrixMultiplier::manager::collect()
          [Communicator comm]
{
   comm.gather(-1, c, rankof(manager));
   return c + 1;
}

void MatrixMultiplier::cell::distribute()
          [Communicator comm]
{
   comm.scatter(&a, rankof(manager));
   comm.scatter(&b, rankof(manager));
}

int* MatrixMultiplier::cell::collect()
          [Communicator comm]
{
   comm.gather(&c, 1, rankof(manager));
   return &c;
}

void MatrixMultiplier::cell::compute()
           [CartesianCommunicator comm]
{
   int west_1_rank, east_1_rank,
       north_1_rank, south_1_rank;
   int west_n_rank, east_n_rank,
       north_n_rank, south_n_rank;

   int i = comm.coordinates[0];
   int j = comm.coordinates[1];

   calculate_ranks_neighbors(comm, i, j,
            &west_n_rank, &east_n_rank,
            &north_n_rank, &south_n_rank);

   // initial alignment
   comm.isend<int>(west_n_rank,0,&a,1);
   comm.isend<int>(north_n_rank,0,&b,1);
   comm.recv<int>(east_n_rank,0,&a,1);
   comm.recv<int>(south_n_rank,0,&b,1);

   calculate_ranks_neighbors(comm, 1, 1,
            &west_1_rank, &east_1_rank,
            &north_1_rank, &south_1_rank);

   // start systolic calculation
   c += a * b;
   for (k=0; k < n-1; k++)
   {
     comm.isend<int>(east_1_rank,0,&a,1);
     comm.isend<int>(south_1_rank,0,&b,1);
     comm.recv<int>(west_1_rank,0,&a,1);
     comm.recv<int>(north_1_rank,0,&b,1);
     c += a * b;
   }
}
```

(b)

Fig. 8. Matrix multiplier example.

3.3.1. Intra-object communication: message-passing among units

Intra-object communication between units of a *p-object* is implemented by means of message-passing through *communicators*, as in MPI. In PObC++, a communicator is a primitive *p-object*, whose methods enable communication among the units of a *p-object* in the execution of parallel methods. The idea of treating communicators as objects comes from state-

```
CartesianCommunicator
   Main:root::create_comm_compute()
          [Communicator my_comm]
{
   Group group_all = my_comm.group();
   Group group_peers
     = group_all.exclude(ranksof(root));
   my_comm.create(group_peers);

   /* the returned communicator is a
      null communicator, since root
      is not in group_peers  */

   return null;
}
```

(a)

```
CartesianCommunicator
   Main:peer::create_comm_compute()
{
   Group group_all = comm.group();
   Group group_peers
     = group_all.exclude(ranksof(root));
   Communicator comm_peers
     = comm.create(group_peers);

   int size = comm_peers.size();
   int dim_size = sqrt(size);
   int[2] dims = {dim_size,dim_size};
   bool[2] periods = {true,true};

   return
      new CartesianCommunicator
         (comm_peers, 2, dims,
                  periods, false);
}
```

(b)

Fig. 9. create_comm_compute.

of-the-art object-oriented versions of the MPI interface, such as MPI.NET [21] and Boost.MPI [20], proposed by the same research group at Indiana University.

Let *A* be a *p-object*. Let *B* be a *p-object* that has a reference to *A*. *A* has been instantiated by *B* or a reference to *A* has been passed in a call to some parallel method of *B*. Thus, *B* may perform invocations to parallel methods of *A*. On each parallel invocation, *B* must provide a communicator, which may be either instantiated by *B* using the PObC++ API or received from another *p-object*. Communicators can be reused in distinct parallel invocations, possibly applied to distinct parallel methods of distinct *p-objects*.

The creation of communicators is illustrated in Fig. 8, where two communicators are created: comm_data, for invocations of parallel class methods distribute and collect; and comm_compute, for invocations of the parallel unit method compute. The communicator comm_data groups all the processes (units of the Main class). For this reason, it is created by cloning the global communicator (world_comm), which is automatically passed to the main function of each process. The communicator comm_compute is a *cartesian communicator*, created in an invocation to the private parallel class method create_comm_compute, whose code is presented in Fig. 9. It involves only the group of cell units. Using the MPI pattern, a communicator is always created from other communicators in a collective operation involving all the members of the original communicator. For this reason, create_comm_compute must also be executed by the root unit, returning null since root is not included in the group of the created communicator.[3]

Notice that a communicator is not explicitly passed to create_comm_compute. In this case, the communicator of the enclosing parallel method (world_comm) is implicitly passed to create_comm_compute, but renamed to my_comm in its scope.

The implicit declaration and passing of communicators are only syntactic sugar for simplifying the use of communicators in simple parallel programs. They are the reason for the use of brackets for delimiting declaration of communicators. In fact, there are two situations where it is useful to explicitly declare a communicator identifier:

- The parallel method receives more than one communicator;
- The programmer wants to use a distinct identifier for referencing the communicator, such as in the example of Fig. 8, where world_comm was used instead of comm.

The rankof and ranksof operators. In Figs. 8 and 9, the operators rankof and ranksof are used to determine the rank of given units in the communicator of a parallel class method. The rankof operator must be applied to the identifier of a singleton unit of the p-object, returning an integer that is the rank of the unit in the communicator of the parallel class method, whereas ranksof must be applied to the identifier of a parallel unit, returning an array of integers. They are useful and allowed only in the code of parallel class methods of *p-objects* that have distinct units. In the first call to rankof or ranksof, communication takes place to determine the ranks of the units of the *p-object* in the current communicator. For this reason, in the first call to a parallel class method all the involved units must call rankof/ranksof collectively. In the subsequent call, such synchronization is not necessary, since the units remember the calculated ranks.

[3] It is important to emphasize that this is a requirement of MPI for making possible to create communicators involving a subgroup of the group of process of an existing communicator. This requirement must be followed by any parallel programming language or scientific computing library implemented on top of MPI.

```
void MatrixMultiplier::peer::calculate_ranks_neighbors
        (CartesianCommunicator& comm, int shift_x, int shift_y
         int* west_rank, int* east_rank, int* north_rank, int* south_rank)
{
   int dim_size_z = comm.Dimensions[0];
   int dim_size_y = comm.Dimensions[1];

   int i = comm.coordinates[0];
   int j = comm.coordinates[1];

   int[2] west_coords = {(i-shift_x) % dim_size_x, j};
   int[2] east_coords = {(i+shift_x) % dim_size_x, j};
   int[2] north_coords = {i, (j-shift_y) % dim_size_y};
   int[2] south_coords = {i, (j+shift_y) % dim_size_y};

   *west_rank = comm.getCartesianRank(west_coords);
   *east_rank = comm.getCartesianRank(east_coords);
   *north_rank = comm.getCartesianRank(north_coords);
   *south_rank = comm.getCartesianRank(south_coords);
}
```

Fig. 10. calculate_ranks_neighbors.

As in MPI, communicators carry topological information about the units. In the default case, units are organized linearly, with ranks defined by consecutive integers from 0 to *size* − 1, where *size* is the number of processes. Alternatively, units may be organized according to a *cartesian topology* in N dimensions, having P_i units in each dimension, for i from 0 to $N-1$. The most general topology is the *graph topology*, where each unit is associated to a set of adjacent units. In general, the unit topologies of *p-objects* follow the communication structure of the underlying algorithm their parallel methods implement. Such information can be useful by the runtime system to make a best mapping of units onto the processing nodes of the parallel computing platform, trying to load balance computations and minimizing communication overheads. In the example of Fig. 8, the communicator *comm_compute* has a cartesian topology with two dimensions and wraparound links, whereas comm_data has the default linear topology.

The communication operations supported by a communicator are those ones supported by communicator objects in Boost.MPI and MPI.NET, both point-to-point and collective subsets. Communicators also support operations to identify units and their relative location in the underlying communicator topology. These operations are dependent on the topology described by the communicator:

- **Linear topology:** The operation size returns the number of processes in the communicator group; the operation rank returns the process identification, which is an integer ranging from 0 to size − 1;
- **Graph topology:** The operations rank and size have the same meaning as for linear topologies. In addition, the operation neighbors returns an array of integers containing the ranks of the adjacent units of the current unit. Similarly, if the current unit wants to know the ranks of the adjacent units of another unit, using its rank, then it may use operation neighborsOf. The operation num_egdes returns the number of edges of the graph of units. Finally, the operation edges returns an adjacent matrix representing the graph.
- **Cartesian topology:** The operations rank, size, neighbors and num_edges are all supported by these kinds of communicators, since a cartesian topology is a special case of graph topology. In addition, the operation dimensions returns an array of integers containing the length of each dimension, whereas the operation coordinates returns the coordinate of the unit in each dimension. The number of dimensions is the length of these arrays. The periodic operation returns an array of boolean values that says if a dimension is periodic or not. Finally, there are the operations getCartesianRank, which returns the *rank* of a unit in a given set of *coordinates*, and getCartesianCoordinates, which returns the *coordinates* of a unit with a given *rank*. In the example of Fig. 8, the parallel method compute calls the unit method calculate_ranks_neighbors twice to calculate the ranks of the four neighbors of the current cell in the two-dimensional mesh at distance shift_x in the x direction (west-east) and shift_y in the y direction (north-south). The code of calculate_ranks_neighbors is shown in Fig. 10. The i and j coordinates of the current cell are determined by calling coordinates. After calculating the coordinates of the neighbor cells, their ranks are obtained by calling getCartesianRank;

In OOPP, objects cannot be transmitted through communicators, but only values of non-object data types, primitive and structured ones. This is not intended to simplify the implementation effort, since Boost.MPI already gives support for object transmission. In fact, we consider communication of objects a source of performance overheads which are difficult to predict, due to marshaling/unmarshaling (serialization) requirements. Performance prediction is an important requirement of HPC applications. Indeed, this restriction makes it possible for the optimization of communication primitives.

The above restriction is not too restrictive for programming, since object transmission may be easily implemented by packing the state of the object in a buffer, sending it through a message to the destination, and loading the packed state into the target object. However, we are considering introducing two linguistic abstractions to deal with use cases of object transmission through message-passing, so called *migration* and (remote) *cloning*. Migration is a variant of cloning where the reference to the object in the original address space is lost after the copying procedure.

3.3.2. Inter-object communication: parallel methods

Remember that inter-object messages are implemented as method invocations of two kinds (Section 3.1):

- *Unit methods*, defined in unit scope, with no access to the *p-object* communicator.
- *Parallel methods*, declared either in the class scope (*parallel class method*) or in the unit scope (*parallel unit method*).

As depicted in Fig. 4, a *parallel method invocation* is composed of a set of *local method invocations* between pairs of units of the caller and the callee *p-objects* that are located in the same processing nodes. Thus, there is no remote method invocation. In most of the parallel programs, since parallel methods are the only synchronization and communication points of the parallel program among units that reside in distinct processing nodes, it is expected that the calls to a parallel method performed by units of a *p-object* must complete together, avoiding excessive synchronization overheads and deadlocks caused by the execution of synchronous point-to-point operations and barrier synchronizations between units involved in a parallel method invocation. In such cases, the numbers of calls to the same parallel method by each unit would be the same.[4] However, such a restriction cannot be enforced statically, giving to the programmers the responsibility of the coherent use of parallel methods, such as in MPI programming. Fortunately, we view the flexibility of letting synchronization between parallel methods being explicitly controlled by the programmer as an interesting opportunity to investigate non-trivial parallel synchronization techniques.

Inter-object communication by method invocations makes it unnecessary to introduce the concept of *inter-communicators*, supported by MPI, in PObC++. According to the MPI standard, inter-communicators make possible message exchanging between groups of processes in disjoint communicators (*intra-communicators*) possible.

3.3.3. Communication and synchronization in the example

In the example of Fig. 8, parallel methods distribute, collect, and compute perform communication operations. They exemplify both collective and point-to-point communication operations through communicators. The first two operations perform collective patterns of communication (scatter and gather) between the manager and the set of worker cells for distributing the input among the cells and for collecting the results they calculate. In compute, point-to-point communication operations (isend and recv) are performed to implement the systolic computation. Before the main loop, the required realignment of elements of matrices *a* and *b* is performed. Each cell communicates with its adjacent cells in the square mesh, at the left, right, up, and down directions.

3.4. Instantiation

A *p-object* is instantiated by another *p-object* by the collective instantiation of each one of their units in distinct processing nodes, using the usual C++ *new* operator applied to the unit identification, which has the form ⟨*class_name*⟩::⟨*unit_name*⟩. This is illustrated in the code of the *p-class* Main of Fig. 8, where the units MatrixMultiplier::manager and MatrixMultiplier::cell are instantiated.

No communication occurs between the units during instantiation. As described before, communication only occurs in parallel method invocations and the caller is responsible for creating an appropriate communicator and passing it to the parallel method it wants to invoke.

The instantiation of a *p-object* is an indivisible operation. However, since there is no communication or synchronization in the instantiation procedure, units may be instantiated at different points of execution. The programmer is responsible to ensure that all units of a *p-object* are properly instantiated before invoking any of their parallel methods. It is a programming error to forget instantiation of one or more units. Notice that there is no restriction to the number of units of a parallel unit to be instantiated in distinct processing nodes. The identification of each unit is defined by the communicator passed to each parallel method invocation and may change across distinct invocations.

3.5. Implementation

PObC++ is an open source project hosted at http://pobcpp.googlecode.com, released under the BSD license. It is composed of a *compiler* and a *standard library*.

3.5.1. Compiler

In order to rapidly produce a fast and reliable compiler prototype, a source-to-source compiler written in C++ was designed by modifying Elsa, an Elkhound-based C/C++ parser. Elkhound [36] is a parser generator that uses the GLR parsing algorithm, an extension of the well-known LR algorithm that handles nondeterministic and ambiguous grammars. In

[4] It is not correct to say that the violation to this programming recommendation always lead to programming errors. In certain situations, a unit of a *p-object* may have no work to perform in a call to a parallel method. In such case, if the parallel method is not invoked for this unit, no error occurs. However, this is a exceptional situation, probably due to a bad parallel algorithm design, which may lead to programming errors since the programmer must infer statically which calls to the parallel method do not have participation of a given unit.

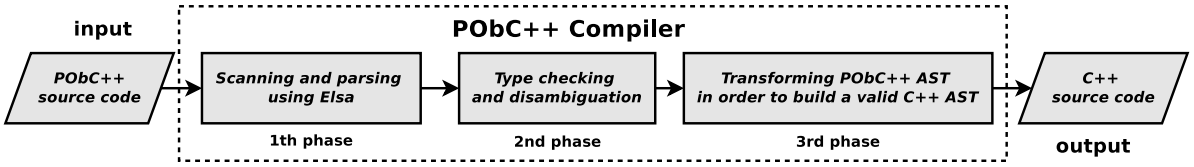

Fig. 11. PObC++ compiler phases.

particular, the Elkhound's algorithm performs disambiguation during the type checking process in order to generate a valid AST (Abstract Syntax Tree). The phases of the PObC++ compiler are depicted in Fig. 11.

The modifications performed during the third phase transform each unit of a *p-class* into a valid C++ class, augmented with meta-information about the units defined within the *p-class*. These adjustments generate only C++ valid code and do not interfere with the rest of the program. In fact, only the code containing *p-class* declarations needs to be compiled by the PObC++ compiler.

C++ code can be used without modification in PObC++ programs. Thus, virtually any C/C++ library can be integrated with PObC++ programs. Since the programmer uses the PObC++ compiler to generate C++, any C++ compiler may be used to generate the native code. Such features are essential to promote the straightforward integration of PObC++ programs with scientific libraries and legacy code written in C and C++. This is the motivation for the case study that will be presented in Section 4.2.

3.5.2. Standard library

The standard PObC++ library is composed of implementations of:

- support for communicators, particularly the comm communicator;
- helper functions to initialize and finalize the environment.

All functions are defined in the scope of the *namespace pobcpp*.

4. Case studies

The case studies presented in this section intend to give an overview of programming techniques, expressiveness, and the potential performance of programs written in the OOPP style.

The first case study presents a parallel numerical integrator that demonstrates skeletal-based parallel programming in PObC++. Also, it presents a performance evaluation where a PObC++ program is compared to its version written purely in C++/Boost.MPI, aiming at providing evidence that the OOPP abstractions supported by PObC++, which makes the use of high-level programming techniques such as skeletons possible, do not add significant overhead to the performance of parallel programs, using C++/MPI programming as a baseline.

The second case study illustrates the integration of PObC++ with existing scientific computing libraries, an important requisite for the acceptance of this language among programmers in scientific and engineering domains. For this, we have created an OOPP interface for a subset of PETSc, a widely used library of subroutines for a solution of sparse algebraic systems for Fortran, C, and C++. According to its developers, PETSc follows an object-based design, despite using a procedural interface since it must be supported by Fortran and C. For this, it supports matrices, vectors and solvers as objects, providing an interface for their instantiation and handling. Therefore, OOPP provides a complete object-oriented alternative for PETSc interface and implementation, providing all the benefits of OOP to PETSc programmers.

The third case study illustrates abstraction and encapsulation of parallelism interaction in *p-objects*. An abstract *p-class* named Sorter has two subclasses named BucketSort and OddEvenSort. They implement distinct parallel algorithms for sorting an array of integers, which uses different patterns of process interactions. The user of a Sorter *p-object* does not need to know how interaction between sorting units takes place. Using the IS kernel of NAS Parallel Benchmarks (NPB) [40] for implementing the bucketsort algorithm, a performance evaluation compares the performance of a PObC++ program with its version written in C/MPI, ignoring not only the overheads of OOPP abstractions, but also the overheads of object-orientation and Boost.MPI indirections.

4.1. 1st case study: farm-based parallel numerical integration

We have implemented a parallel numerical integrator using two *p-classes*: Farm and Integrator. The former one is an abstract class, implementing the skeleton *farm*, which implements an abstraction for a well-known parallelism strategy, including its pattern of inter-process communication. The class Integrator extends Farm aiming to implement a parallel numerical integration using farm-based parallelism. We have reused the numerical integration algorithm implemented by the NINTLIB library, based on the Romberg's Method [37]. NINTLIB is not parallel. Therefore, it has been necessary to parallelize it.

Skeletal programming. The term *algorithmic skeleton* was first coined by Murray Cole two decades ago to describe reusable patterns of parallel computations whose implementation may be tuned to specific parallel execution platforms [38]. Skeletons have been widely investigated by the academic community, being considered a promising approach for high-level parallel programming [6]. Skeletal programming is an important parallel programming technique of OOPP.

4.1.1. The Farm class (skeleton)

The Farm class is presented in Fig. 12(a). It declares two parallel methods, synchronize_jobs and synchronize_results, which are invoked by the units of a farm object to transmit the jobs, from the manager to the workers, and the results of the jobs, calculated by the method work, from the workers to the manager. Despite work being implemented by each worker, it is still a unit method, and not a parallel method. In fact, the work of a worker is a local procedure in its nature. For this reason, no communication is needed among the workers of the farm when they are working.

The methods synchronize_jobs, synchronize_results, and perform_jobs have default implementations in the Farm, which are not presented in Fig. 12(a). The implementation is supposed to be the best possible for the target execution platform, freeing programmers from the burden of knowledge about details of the target parallel architecture in order to choose the best collective communication algorithm for distributing jobs and collecting results. Indeed, the programmers may link their farm-based parallel programs to any Farm class implementation that is specialized for some architecture.

The methods add_jobs, get_next_result and get_all_results also have default implementations. The last two blocks until an unread result have arrived at the manager from the workers and until all results from the workers are available, respectively. get_next_result returns null if all results have arrived. These methods are prepared for the concurrency among the methods synchronize_jobs, synchronize_results, and perform_jobs, which have been developed with thread safety in mind. So, the user may call these methods on distinct threads to overlap computation and communication. However, according to the characteristics of the target parallel computing platform, the implementation will decide how many jobs must be received by add_jobs before beginning to send jobs to the workers, as well as deciding how many results must be calculated by a worker before sending a result, or a set of results, to the manager.

In order to use a farm, the programmer must extend the Farm class, by inheritance. Then, it must implement the virtual unit methods pack_job, unpack_result, pack_result, unpack_job, and work. The first four methods are necessary to pack/unpack job and result objects to/from arrays of bytes, in order to be transmitted through the communicator. The actual Result and Job types must be defined. Remember that objects cannot be transmitted through communicators, but only data values. Packing and unpacking procedures are necessary to keep the generality of the Farm. The method work defines the computation performed by each work, which essentially defines the problem under solution.

4.1.2. The Integrator class

The Integrator class, as declared in the header file *integrator.h*, is presented in Fig. 12(b). Besides the virtual methods inherited from the Farm class, it will implement the manager methods generate_subproblems, which will partition the integration interval, build the job objects, and call add_jobs in order to feed the farm's job list, and combine_subproblems_results, which will call get_next_result or get_all_results to retrieve the numerical results calculated by the workers, which will be added.

4.1.3. The main program

The main class of the integrator program is outlined in Fig. 13. The implementation tries to exploit all the concurrency available in the farm, by forking threads using OpenMP [39], a library of subroutines and preprocessor directives for shared-memory parallel programming. This is an example of how multi-threading parallelism can be exploited inside units of *p-objects*.

The reader may notice that a communicator is implicitly passed to the parallel method invocations in the code of main. As pointed out in Section 3.3.1, if a communicator is not explicitly provided, the communicator of the enclosing parallel invocation is implicitly passed. In the example, such communicator is the global communicator, which is received by the main method. This is only possible if the enclosing parallel method has only one communicator. Also, notice that the communicator identifier of main is not explicitly informed. In such cases, the communicator has the default identifier comm.

4.1.4. Performance evaluation: the weight of OOPP abstractions

Tables 1 and 2 summarize the performance results obtained by executing the PObC++ numerical integrator and its C++/MPI version on the CENAPAD-UFC cluster,[5] installed at the National Center for High Performance Computing (CENAPAD) at Universidade Federal do Ceará (UFC), by varying the number of processors (P) and dimensions (n).

This experiment aims at providing evidences that the weight of new OOPP languages abstraction added to C++ by PObC++ is insignificant compared to the gains in modularity and abstraction they give support.

[5] CENAPAD-UFC is composed by 48 Bull 500 blades, each one with two Intel Xeon X5650 processors and 24 GB of DDR3 1333 MHz RAM memory. Each processor has 6 Westmere EP cores. The processors communicate through a Infiniband QDR (36 ports) interconnection. More information about CENAPAD at http://www.cenapad.ufc.br and https://www.lncc.br/sinapad/.

```
template<typename Job, typename Result>
class Farm
{
  public:
    void synchronize_jobs();
    void synchronize_results();

  unit manager
  {
    private:
      Job* all_jobs;
      Result* all_results;

    public:
      void add_jobs(Job* job);
      Result get_next_result();
      Result* get_all_results();
      virtual void* pack_jobs(Job* jobs);
      virtual Result unpack_result(void* result);
  };

  parallel unit worker
  {
    private:
      Job* local_jobs;
      Result* local_results;

    public:
      parallel void perform_jobs();
      virtual Result work(Job job);
      virtual Job unpack_jobs(void* jobs);
      virtual void* pack_result(Result* result);
   };
};
```

(a)

```
class Integrator:
    public Farm<IntegratorJob, double>
{
  unit manager
  {
    private:
      int inf, sup;
      int dim_num, partition_size;

    public:
      Manager(int inf, int sup,
              int dim_num, int psize) :
          inf(inf), sup(sup),
          dim_num(dim_num),
          partition_size(psize) { }

    public:
      void generate_subproblems();
      double combine_results();
  };

  parallel unit worker
  {
    private:
      int number_of_partitions;
      int next_unsolved_subproblem;
      double (*function)(double*);

    public:
      Worker(double (*f)(double*),
             int tol, int nop) :
          function(f),
          number_of_partitions(nop),
          next_unsolved_subproblem(0),
          tolerance(tol) { }
  };
 };
```

(b)

Fig. 12. The Farm class (a) and (b). The Integrator class (b).

```
class IntegratorMain
{
  public: int main();

  unit root
  {
    int main()
    {
      #pragma omp parallel
      {
        Integrator::manager *m
             = new Integrator::manager
                          (0.0, 1.0, 5, 2);
        double result;
        #pragma omp sections
        {
            #pragma omp section
            m -> generate_subproblems();
            #pragma omp section
            m -> synchronize_jobs();
            #pragma omp section
            m -> synchronize_results();
            #pragma omp section
            result = m->combine_results();

        }
        cout >> "Result is ", result;
      }
    }
  };

(...)
```

```
(...)

parallel unit peer
{
    int main()
    {
      #pragma omp parallel
      {
        Integrator::worker *w
             = new Integrator::worker
                      (function, 1D-5, 16);
        #pragma omp sections
        {
            #pragma omp section
            w -> synchronize_jobs();
            #pragma omp section
            w -> perform_jobs();
            #pragma omp section
            w -> synchronize_results();
        }
      }
    }
  };
};
```

Fig. 13. Main classes (farm-based numerical integrator).

Table 1
Average execution times (in seconds) and confidence interval ($\alpha = 5\%$) for comparing PObC++ to C++ for different workloads and number of processing nodes in parallel multidimensional integration.

	$n = 6$				$n = 7$				$n = 8$			
P	PObC++	C++/MPI	δ_{min} δ_{max}	Diff	PObC++	C++/MPI	δ_{min} δ_{max}	Diff	PObC++	C++/MPI	δ_{min} δ_{max}	Diff
1	$5.93_{\pm.001}$	$5.83_{\pm.007}$	+1.5% +1.8%	y	$109.1_{\pm.018}$	$107.8_{\pm.089}$	+1.0% +1.2%	y	$1971_{\pm.354}$	$1942_{\pm2.09}$	+1.4% +1.7%	y
2	$3.00_{\pm.002}$	$2.95_{\pm.003}$	+1.3% +1.6%	y	$55.0_{\pm.010}$	$54.5_{\pm.031}$	+1.0% +1.1%	y	$997_{\pm.311}$	$983_{\pm.848}$	+1.4% +1.6%	y
4	$1.50_{\pm.001}$	$1.49_{\pm.001}$	+1.1% +1.4%	y	$27.7_{\pm.013}$	$27.3_{\pm.019}$	+0.9% +1.1%	y	$500_{\pm.144}$	$493_{\pm.641}$	+1.1% +1.4%	y
8	$0.76_{\pm.001}$	$0.75_{\pm.001}$	+0.6% +1.1%	y	$13.8_{\pm.040}$	$13.9_{\pm.113}$	−1.2% +1.1%	n	$252_{\pm.228}$	$247_{\pm.337}$	+1.2% +1.6%	y
16	$0.38_{\pm.001}$	$0.39_{\pm.005}$	−5.1% −2.2%	y	$6.9_{\pm.011}$	$7.0_{\pm.033}$	−1.9% −0.7%	y	$125_{\pm.058}$	$124_{\pm.156}$	−0.5% −0.1%	y
32	$0.20_{\pm.003}$	$0.20_{\pm.003}$	−2.9% +2.6%	n	$3.6_{\pm.046}$	$3.5_{\pm.054}$	−3.2% +2.4%	n	$64.9_{\pm.911}$	$63.4_{\pm.764}$	−0.3% +4.9%	n
Seq	5.43				104.1				1921			

Table 2
Speedup of parallel multidimensional integration with PObC++ and C++.

	$n = 6$		$n = 7$		$n = 8$	
P	PObC++	C++/MPI	PObC++	C++/MPI	PObC++	C++/MPI
1	0.9	0.9	0.9	0.9	0.9	0.9
2	1.8	1.8	1.8	1.9	1.9	1.9
4	3.6	3.6	3.7	3.8	3.9	3.8
8	7.1	7.1	7.5	7.4	7.8	7.7
16	14.2	13.9	15.0	14.8	15.7	15.4
32	27.1	27.1	28.9	29.7	30.3	30.2
N = number of dimensions						

The average execution times and confidence intervals presented in Table 1 have been calculated from a sample of 40 executions. For the purposes of the analysis, it is supposed that the distribution of the observed execution times is normal. For improving reliability, outliers have been discarded using the Box Plot method for $k = 4.0$ (length of the upper and lower fences). Without the outliers, each sample has more than 30 observations, which is recommended for ensuring statistical confidence according to the literature [41]. The clusters have been dedicated to the experiment, resulting in relatively low standard deviations and, as a consequence, tight confidence intervals, contributing to the reliability of the analysis. All the measures and relevant statistical summaries for this experiment can be obtained at http://pobcpp.googlecode.com.

In the experiment, the function

$$f(x_1, x_2, x_3, \ldots, x_n) = {x_1}^2 + {x_2}^2 + {x_3}^2 + \cdots + {x_n}^2$$

is integrated in the interval [0, 1] in parallel. For that, the parameter *dim_partition_size* defines the number of partitions of the interval in each dimension, of the same size, yielding $num\_jobs = dim\_partition\_size^n$ subproblems of integration. It is also necessary to define the parameter *number_of_partitions* required by the procedure NINTLIB.romberg_nd, as a multiple of *num_jobs*, in order to preserve the amount of computation performed by the sequential version. Empirically, we set $dim\_partition\_size = 2$ and $number\_of\_partitions = 8$, by varying n between 6 and 8.

In Table 1, the lower (δ_{min}) and upper (δ_{max}) estimations of the overhead of PObC++ in relation to C++/MPI, for each pair (P, n), is presented. They are calculated as following: Let $[x_0, x_1]$ $[y_0, y_1]$ be the confidence interval for the observations of a pair (P, n) for PObC++ and C++/MPI, respectively. Then, $\delta_{min} = (y_1 - x_0)/\bar{x}$ and $\delta_{max} = (y_0 - x_1)/\bar{x}$, where $\bar{x}$ stands for the average execution time of C++/MPI, since overhead is a relative measure of the additional execution time when PObC++ is used in alternative to C++/MPI. Thus, a negative value for an overhead estimation means that PObC++ outperformed C++/MPI for the given workload (n) and number of processing nodes (P). If the lower and upper overhead estimations have different signals, it means that it is not possible to state, for statistical confidence at 5%, that either PObC++ or C++/MPI outperformed the other. On the other hand, if both estimations have the same signal, either PObC++ (negative) or C++/MPI (positive) is considered better.

The overhead estimations in Table 1 show that C++/MPI always outperforms PObC++ for 8 or less processing nodes. For 16 and 32 nodes, either PObC++ outperforms C++/MPI or neither one is better than the other. Also, it is important to note that only 2 out of the 18 positive overhead estimations are greater than 2.0% (+2.4% and 4.9%). These is strong evidence that PObC++ is a good alternative in relation to C++/MPI, since the performance degradations are insignificant compared to the gains in modularity and abstraction offered by object-orientation for the programming practice. The performance equivalence of PObC++ and C+/MPI is not a surprise, since the PObC++ version is translated to a C++/MPI program that is almost equivalent to a C++/MPI version built from scratch, except for call indirections from the communicator interface to the MPI subroutines. The essential difference is the better object-oriented properties achieved by PObC++, with potential

```
class ParallelVec
{
  public:
    PetscErrorCode Create();
  private:
    Vec vec;

  parallel unit pvec {};
};

PetscErrorCode
   ParallelVec::pvec::Create()
{
  return
    VecCreate(comm->get_mpi_comm(),
              &vec);
}
```

```
class ParallelKSP
{
   public:
     PetscErrorCode
        Create();
     PetscErrorCode
        SetOperators(ParallelMat::pmat& Amat,
                     ParallelMat::pmat& Pmat,
                     MatStructure flag);
     PetscErrorCode
        Solve(ParallelVec::pvec& b,
              ParallelVec::pvec& x);

     /* Other methods: tolerance control,
        preconditioning, etc */
     ...

   private: KSP ksp;

   parallel unit ksp {};
};
```

Fig. 14. Case study ParallelVec (a) and PKSP (b).

gains in modularity, abstraction, and usability. The same techniques used by an MPI programmer may be almost directly applied using PObC++.

Table 2 is a measure of the scalability of the implementation of the parallel multidimensional integration used in the experiments. Notice that the speedup is almost linear and increases as the workload (n) increases.

4.2. 2nd case study: an OOPP interface for PETSc

PETSc is a library for scientific computing designed as a set of data structures and subroutines to solve linear and non-linear algebraic systems in parallel, using MPI. It is widely used by engineering and scientific programmers in the numerical solution of partial differential equations (PDE), which describe phenomena of their interest. PETSc follows an object-based design on top of non-OOP languages, such as Fortran and C. This section shows a real OOPP interface for PETSc, restricted to three important modules, demonstrating the integration of PObC++ with scientific computing libraries.

4.2.1. *ParallelVec* and *ParallelMat* class

ParallelVec and ParallelMat are *p-classes* that represent the `Vec` and `Mat` data structures of PETSc, respectively. `Vec` represents a vector that is distributed across the computing nodes, whereas `Mat` represents a distributed matrix. In Fig. 14(a), the code of ParallelVec is presented. The code of ParallelMat is similar. They have parallel units pvec and pmat, respectively. It is important to note that PETSc requires an MPI communicator for the creation of vectors and matrices. This is often necessary in parallel scientific libraries on top of MPI. In PObC++, this is supported by declaring `Create` as a parallel method. Then, a primitive communicator can be obtained by calling the method `get_mpi_comm` of the communicator `comm`, which may be passed to `VecCreate` and `MatCreate` underlying PETSc subroutines. This is a valid assumption, since PObC++ is implemented on top of the MPI standard by design. In ParallelVec, the call to `VecCreate` will assign a `Vec` data structure to the private attribute vec, representing the underlying PETSc vector that stores the local partition of the vector that is inside the address space of the unit.

4.2.2. The *ParallelKSP* class

ParallelKSP declares a parallel unit named ksp, representing a parallel solver for linear systems using the Krylov subspace methods. Fig. 14(b) outlines the essential elements of the interface of a *p-object* of *p-class* ParallelKSP. After instantiation of the solver, by calling the constructor of ParallelKSP, the user must call the Create method to initialize the solver data structures across computing nodes. Then, it must call SetOperators to provide the operator matrices, Amat and Pmat, which are reused across many subsequent calls to the method Solve, to which the most specific part of the problem parameters, vectors b and x, are informed. The ksp attribute is the underlying PETSc solver.

4.3. 3rd case study: parallel sorting

This section presents an abstract *p-class* called Sorter, which is specialized in two subclasses of p-objects that implement well-known sorting algorithms, *bucketsort* and *odd–even sort*. Such *p-classes* are called BucketSort and OddEvenSort,

```
class Sorter
{
  public: void virtual sort() = 0;

  parallel unit worker
  {
    public:
      void set(int* items, int size);
    protected:
      int* items;
      int size;
    };
};
```

```
class BucketSort : Sorter
{
  public: void sort();

  parallel unit worker
  {
    private:
      void local_sort(int* items,
                      int size);
      void fill_buckets(int* key_array_1,
                        int* key_array_2,
                        int* bucket_ptr,
                        int* bucket_size)
  };
};
```

Fig. 15. Sorter (a) and BucketSort (b).

respectively. They illustrate abstraction and encapsulation in OOPP, in relation to communication and synchronization among units of a *p-object*, since the communication sequences among units of the two sorters use distinct parallelism interaction patterns.

4.3.1. The Sorter class

The source code of the Sorter *p-class* is presented in Fig. 15(a). It declares a virtual parallel function sort. Also, it comprises a parallel unit worker that declares an array of integers to be sorted across workers, represented by variables items and size. Thus, any specialization of Sorter needs to implement the sort method according to its chosen algorithm.

Notice that items and size are declared as unit attributes, instead of class attributes. This is because they represent a local partition of the array to be sorted, which is initially distributed across the units. At the end of execution, each element in the array items of unit i is less or equal to any element in the array items of unit j, provided that $i < j$. In the alternative design, letting items and size be class attributes, it is implicitly supposed that all items are known by all the units, at the beginning and at the end of the sorting process, giving to the sort method the responsibility to perform an initial distribution of the items across the units. For the reasons discussed in Section 3.2, such restrictions must be enforced by the programmer, since both syntactical ways to declare items and size have the same semantics in PObC++.

4.3.2. The BucketSort class

Bucketsort is an efficient ($\Theta(n)$) sorting algorithm, provided that the items are uniformly dispersed in the input array. Let n be the number of items. There is a set $\{b_0, \ldots, b_{k-1}\}$ of buckets, where $n >> k$. The bucketsort algorithm has the following steps:

1. The items are distributed across the set of buckets, such that if $p \in b_i \wedge q \in b_j$ then $p < q$ iff $i < j$;
2. The items inside each bucket are locally sorted using some sorting algorithm.

The *p-class* BucketSort implements a parallel version of the bucketsort algorithm. For that, it inherits from Sorter, by implementing the sort method. There are two strategies for implementing BucketSort, based on two ways of parallelizing the bucketsort algorithm, leading to two alternative implementations of sort:

1. There is a *root* worker that initially knows all the items, performing the first step of the sequential algorithm. Then, it distributes the buckets across all workers using the method scatter of comm, itself included. The workers perform sequential sorting inside the buckets they received. Finally, the workers send the sorted buckets back to the root worker using the method gather of comm;
2. The items are initially distributed across the workers, such that each worker has local versions of each one of the k buckets and distributes its local items across their local buckets. After that, the workers distribute the buckets, by concatenating local versions of the same bucket in a single bucket in the address space of a single worker, using the method alltoallflattened of comm. Let r and s be the rank of two arbitrary workers and let b_i and b_j be buckets stored by r and s, respectively. In the final distribution, $r < s$ iff $i < j$. Finally, the workers perform sequential sorting inside each bucket they received. The items are now sorted across the workers.

The latter strategy is more appropriate when there is a fast computation to decide which bucket an item must be assigned to. Also, the set of items must fit the address space of the root worker. The computation is dominated by the sequential sorting performed by each worker. However, such strategy breaks the assumption of the Sorter *p-class* that the items must be distributed across the workers at the beginning of the sorting process.

The former strategy is better if workers do not have enough address space to store all the items. Thus, the items must stay distributed across the workers during execution, as required by *p-class* Sorter. Fig. 16 outlines the implementation of

```
void BucketSort::worker::sort()
{
    /* Main data structures */
    int* key_array = items; // array of local keys
    int* key_buffer_send;   // array of local keys in their buckets
    int* bucket_ptr;        // bucket pointers in key_buffer_send
    int* bucket_size;       // bucket sizes
    int* key_buffer_recv;   // array of global keys in their buckets

    /* Initialize data structures */
    ...
    /* Copy the elements of key_array to key_buffer_send, partially sorted
       according to their buckets. The value bucket_ptr[i] is the start index
       of the i-th bucket in key_buffer_send. The value bucket_size[i] is the
       number of items in the i-th bucket. */
    fill_buckets(key_array, key_buffer_send, bucket_ptr, bucket_size);
    /* Determine the global size of each bucket */
    comm.allreduce(bucket_size, Operation<int>.Add, bucket_size_totals);
    /* Determine how many local items will be sent to each process */
    comm.alltoall(send_count, recv_count);
    /* Send items to each process */
    comm.alltoallflattened(key_buffer_send,send_counts,key_buffer_recv,outValues);
    /* Sort the buckets */
    local_sort(key_buffer_recv, size);
}
```

Fig. 16. BucketSort *p-class* (sort method).

the sort method of BucketSort using the second strategy. The source code is based on the IS (Integer Sorting) kernel of the NAS Parallel Benchmarks [40], which implements bucketsort for evaluating the performance of clusters and MPP's (Massive Parallel Processors).

There are two important points to note about this case study:

- An object-oriented parallel programmer must take care with implicit assumptions of abstract *p-classes* concerning distribution of input and output data structures, when deriving concrete *p-classes* from them, such as the assumption of Sorter that requires items to be distributed across the workers;
- The two alternative solutions use collective communication methods of the communicator, but the user of the sorter does not need to be aware of the communication operations performed by the worker units. This is completely transparent from the perspective of the user of the sorter. Indeed, it is safe to change the parallelism strategy by choosing a distinct subclass of the Sorter *p-class* that implements another parallelization strategy.

4.3.3. The OddEvenSort class

The odd–even sort is a comparison sorting algorithm based on the well-known bubblesort algorithm [42]. Thus, it has $\Theta(n^2)$ complexity, being essentially distinct from bucketsort. Whereas bubblesort performs a sequence of sequential traversals of the array of items until they are sorted, by comparing the neighboring elements and exchanging them if they are in the wrong order, odd–even sort alternates two kinds of traversals. In the *odd–even* traversals, each odd element is compared to its next neighbor. In *even–odd* traversals, each even element is compared to its next neighbor. The compared elements are still exchanged when they are in the wrong order, such as in bubblesort. Odd–even is an intrinsically shared-memory parallel algorithm, since *odd–even* and *even–odd* traversals may be performed concurrently.

The *p-class* OddEvenSort implements a distributed variant of odd–even sort. As BucketSort, it is derived by inheritance from Sorter, implementing the sort method, where comparisons, exchanges and communication/synchronization operations take place. The sort method of OddEvenSort is outlined in Fig. 17. The items (array items) are distributed among the workers, such as in the second bucketsort implementation. First of all, the workers sort their local items using some sorting algorithm (e.g. quicksort). Then, the traversals of odd–even sort are alternated across the workers. In the *odd–even* traversals, the workers with an odd rank and their respective next neighbors exchange their items. In the *even–odd* traversals, the workers with an even rank and their respective next neighbors exchange their items. The exchange of items between workers is performed through calls to point-to-point operations on comm, isend and recv. After a traversal, each pair of workers involved in an exchanging has knowledge about the items of both workers. Let n be the number of items. Then, the worker with the smaller rank copies the $n/2$ smaller items in its array items, whereas the other worker copies the $n/2$ greater ones. The process is repeated until the items are sorted across all workers.

```
void OddEvenSort::worker::sort()
{
  /* declaration and initialization of local variables */
  ...
  /* Sort the local elements using the built-in quicksort routine */
  std::local_sort(items, nlocal);
  /* Determine the rank of neighbors */
  oddrank  = rank % 2 == 0 ? rank - 1 : rank + 1;
  evenrank = rank % 2 == 0 ? rank + 1 : rank - 1;
  /* Get into the main loop of the odd-even sorting algorithm */
  for (j = 0; j < nworkers-1; j++)
  {
    if (j%2 == 1 && oddrank != -1 && oddrank != nworkers)
    {   // Odd-Even traversal
        comm.isend(oddrank, 0, items, nlocal);
        comm.receive(oddrank, 0, ritems, nlocal);
    }
    else if (evenrank != -1 && evenrank != nworkers)
    {   // Even-Odd traversal
        comm.isend(evenrank, 0, items, nlocal);
        comm.receive(evenrank, 0, ritems, nlocal);
    }
    compareSplit(nlocal, items, ritems, wspace,rank < status.MPI_SOURCE);
  }
  ...
}
```

Fig. 17. OddEvenSort *p-class* (sort method).

Table 3
Average execution times (in seconds) and confidence interval ($\alpha = 5\%$) for comparing PObC++ with C for different workloads and number of processing nodes using the kernel IS (Integer Sort) of NPB (NAS Parallel Benchmarks).

	Class B				Class C				Class D			
P	PObC++	C/MPI	δ_{min} δ_{max}	Diff	PObC++	C/MPI	δ_{min} δ_{max}	Diff	PObC++	C/MPI	δ_{min} δ_{max}	Diff
1	$6.61_{\pm 0.004}$	$5.92_{\pm 0.025}$	+11.0% +12.0%	y	$26.83_{\pm 0.012}$	$24.21_{\pm 0.091}$	+10.4% +11.3%	y	–	–	– –	y
2	$3.33_{\pm 0.002}$	$3.04_{\pm 0.010}$	+9.1% +9.9%	y	$13.49_{\pm 0.006}$	$12.35_{\pm 0.046}$	+8.7% +9.6%	y	–	–	– –	y
4	$1.70_{\pm 0.004}$	$1.56_{\pm 0.006}$	+8.4% +9.7%	y	$6.91_{\pm 0.002}$	$6.34_{\pm 0.024}$	+8.6% +9.4%	y	$133.0_{\pm 0.029}$	$111.7_{\pm 0.334}$	+18.7% +19.4%	y
8	$0.88_{\pm 0.001}$	$0.80_{\pm 0.003}$	+8.7% +9.8%	y	$3.60_{\pm 0.000}$	$3.31_{\pm 0.010}$	+8.3% +8.9%	n	$69.05_{\pm 0.016}$	$58.90_{\pm 0.184}$	+16.9% +17.6%	y
16	$0.47_{\pm 0.001}$	$0.43_{\pm 0.001}$	+8.5% +9.7%	y	$1.90_{\pm 0.002}$	$1.75_{\pm 0.006}$	+7.7% +8.5%	y	$37.70_{\pm 0.008}$	$33.29_{\pm 0.108}$	+12.9% +13.6%	y
32	$0.27_{\pm 0.001}$	$0.25_{\pm 0.001}$	+7.3% +8.0%	y	$1.11_{\pm 0.000}$	$1.03_{\pm 0.003}$	+6.9% +7.6%	y	$23.21_{\pm 0.007}$	$14.05_{\pm 0.055}$	+8.9% +9.5%	y
64	$0.19_{\pm 0.000}$	$0.18_{\pm 0.001}$	+5.1% +6.0%	y	$0.75_{\pm 0.002}$	$0.70_{\pm 0.002}$	+5.3% +6.4%	y	$15.01_{\pm 0.004}$	$14.05_{\pm 0.040}$	+6.6% +7.2%	y

4.3.4. Performance evaluation (the overall weight of object-orientation)

Table 3 presents the performance results obtained by PObC++ for its version of the IS kernel of NPB. The goal of this experiment is slightly different from the experiment described in Section 4.1.4 with the numerical multidimensional integrator. It aims to compare the performance of a PObC++ program with the best native implementation written in C. Thus, ignoring the weight of both object-orientation and the additional OOPP abstractions. Also, remember that PObC++ is implemented on top of Boost.MPI, an object-oriented interface to MPI, being an indirection subject to performance penalties.

As expected, C/MPI outperforms PObC++ for all the pairs (P, n). The overhead estimations are also greater than the overheads observed in multidimensional integration, varying between 5% and 12%. However, it is possible to observe that the overheads decrease as the number of processors increases in this experiment, which suggests that the performance of PObC++ is comparable to the performance of C/MPI for a large number of processors. Finally, from the results of this experiment, the most important conclusion is that the gains in modularity and abstraction compensate for the overheads.

All the measures and relevant statistical summaries for this experiment can also be obtained at http://pobcpp.googlecode.com.

5. Conclusions and further work

Due to the increasing interest in HPC techniques in the software industry, mainly parallel processing, as well as the increasing importance of scientific and engineering applications in modern human society, the increasing complexity of applications in HPC domains has attracted the attention of a significant number of researchers on programming models, languages and techniques. They are faced with the challenging problem of reconciling well-known techniques to deal with software complexity and large scale in corporative applications with the high performance requirements demanded by applications in science and engineering.

Object-oriented programming is considered one of the main responses of programming language designers for dealing with high complexity and scale of software. Since the 1990s, such programming style has become widespread among programmers. Despite their success among programmers in business and corporative application domains, object-oriented languages do not have the same acceptance among programmers in HPC domains, mainly among scientists and engineers. This is usually explained by the performance overhead caused by some features present in these languages for supporting higher levels of abstraction, modularity and safety, and by the additional complexity introduced by parallel programming support.

The results presented in this paper, including design, implementation and performance evaluation of the first PobC++ prototype, are very promising. The examples are evidence that the proposed approach may coherently reconcile the common programming styles adopted by parallel programmers and by object-oriented programmers, making it possible for a programmer well educated in both parallel programming using MPI and in OOP, to take rapid advantage of OOPP features. Moreover, the performance results evidence tolerable performance overheads, despite the gains in modularity and abstraction when compared to direct MPI programming.

Acknowledgments

This work has been sponsored by CNPq, grant numbers 475826/2006-0 and 480307/2009-1.

References

[1] M. Baker, R. Buyya, D. Hyde, Cluster computing: a high performance contender, IEEE Computer 42 (7) (1999) 79–83.
[2] I. Foster, The Grid: Blueprint for a New Computing Infrastructure, first ed., Morgan Kaufmann, 1998.
[3] D.E. Post, L.G. Votta, Computational science demands a new paradigm, Physics Today 58 (1) (2005) 35–41.
[4] D.E. Bernholdt, J. Nieplocha, P. Sadayappan, Raising level of programming abstraction in scalable programming models, in: IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC), Madrid, Spain, IEEE Computer Society, 2004, pp. 76–84.
[5] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, Sourcebook of Parallel Computing, Morgan Kauffman Publishers, 2003 (Chapters 20–21).
[6] H. Kuchen, M.e. Cole, Algorithm skeletons, Parallel Computing 32 (2006) 447–626.
[7] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.
[8] J. Dongarra, S.W. Otto, M. Snir, D. Walker, A message passing standard for MPP and workstation, Communications of ACM 39 (7) (1996) 84–90.
[9] E. Dijkstra, The humble programmer, Communications of the ACM 15 (10) (1972) 859–866.
[10] O.J. Dahl, SIMULA 67 Common Base Language, Norwegian Computing Center, 1968.
[11] O.J. Dahl, The birth of object orientation: the simula languages, in: Software Pioneers: Contributions to Software Engineering, Programming, and Operating Systems Series, Springer, 2002, pp. 79–90.
[12] A. Goldberg, D. Robson, Smalltalk-80: the Language and its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
[13] H. Milli, A. Elkharraz, H. Mcheick, Understanding separation of concerns, in: Workshop on Early Aspects — Aspect Oriented Software Development, AOSD'04, 2004, pp. 411–428.
[14] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (1996) 438–479. URL: http://doi.acm.org/10.1145/243439.243441.
[15] B. Meyer, Object-Oriented Software Construction, Prentice Hall, Upper Saddle River, NJ, USA, 1988.
[16] M. Baker, B. Carpenter, G. Fox, S.H. Ko, X. Li, mpijava: a java interface to mpi, in: Procedings of the First UK Workshop on Java for High Performance Network Computing, 1998.
[17] M. Baker, B. Carpenter, Mpj: a proposed java message passing api and environment for high performance computing, in: IPDPS'00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, Springer-Verlag, London, UK, 2000, pp. 552–559.
[18] S. Mintchev, Writing programs in javampi, Tech. Rep. MAN-CSPE-02, School of Computer Science, University of Westminster, London, UK, Oct. 1997.
[19] B.-Y. Zhang, G.-W. Yang, W.-M. Zheng, Jcluster: an efficient java parallel environment on a large-scale heterogeneous cluster, Concurrency and Computation: Practice and Experience 18 (12) (2005) 1541–1557. http://dx.doi.org/10.1002/cpe.986.
[20] G. Douglas, T. Matthias, Boost.mpi website, May 2010. URL: http://www.boost.org/doc/html/mpi.html.
[21] D. Gregor, A. Lumsdaine, Design and implementation of a high-performance mpi for c# and the common language infrastructure, in: PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, New York, NY, USA, 2008, pp. 133–142. http://doi.acm.org/10.1145/1345206.1345228.
[22] L.V. Kale, S. Krishnan, Charm++: a portable concurrent object oriented system based on C++, Tech. rep., Champaign, IL, USA, 1993.
[23] M. Philippsen, M. Zenger, JavaParty — transparent remote objects in java, Concurrency and Computation: Practice and Experience 9 (11) (1997) 1225–1242.
[24] T. Nguyen, P. Kuonen, ParoC++: a requirement-driven parallel object-oriented programming language, in: International Workshop on High-Level Programming Models and Supportive Environments, IEEE Computer Society, Los Alamitos, CA, USA, 2003, p. 25. http://doi.ieeecomputersociety.org/10.1109/HIPS.2003.1196492.
[25] T. Nguyen, P. Kuonen, Programming the grid with pop-C++, Future Generation Computer Systems 23 (1) (2007) 23–30. http://dx.doi.org/10.1016/j.future.2006.04.012.
[26] Y. Aridor, M. Factor, A. Teperman, T. Eilam, A. Schuster, A high performance cluster jvm presenting pure single system image, in: JAVA'00: Proceedings of the ACM 2000 conference on Java Grande, ACM, New York, NY, USA, 2000, pp. 168–177. http://doi.acm.org/10.1145/337449.337543.

[27] V. Sarkar, X10: an object oriented aproach to non-uniform cluster computing, in: OOPSLA'05: Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, New York, NY, USA, 2005, p. 393. http://doi.acm.org/10.1145/1094855.1125356.
[28] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312. http://dx.doi.org/10.1177/1094342007078442.
[29] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessn, S. Ryu, G. Steele Jr., S. Tobin Hochstad, The Fortress Language Specification Version 1.0, Mar. 2008.
[30] K.A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P.N. Hilfinger, S.L. Graham, D. Gay, P. Colella, A. Aiken, Titanium: a high-performance Java dialect, in: Java for High-performance Network Computing, Concurrency: Practice and Experience 10 (11–13) (1998) 825–836 (special issue).
[31] E. Lusk, K. Yelick, Languages for high-productivity computing — the DARPA HPCS language support, Parallel Processing Letters 1 (2007) 89–102.
[32] F.H. Carvalho Jr., R. Lins, R.C. Correa, G.A. Araújo, Towards an architecture for component-oriented parallel programming, Concurrency and Computation: Practice and Experience 19 (5) (2007) 697–719.
[33] A. Grama, A. Gupta, J. Karypis, V. Kumar, Introduction to Parallel Computing, Addison-Wesley, 1976.
[34] S.L. Johnsson, T. Harris, K.K. Mathur, Matrix multiplication on the connection machine, in: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Supercomputing'89, ACM, New York, NY, USA, 1989, pp. 326–332. URL: http://doi.acm.org/10.1145/76263.76298.
[35] F. Bertran, R. Bramley, A. Sussman, D.E. Bernholdt, J.A. Kohl, J.W. Larson, K.B. Damevski, Data redistribution and remote method invocation in parallel component architectures, in: 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2005.
[36] S.G. McPeak, Elkhound: a fast, practical glr parser generator, Tech. rep., Berkeley, CA, USA, 2003.
[37] J. Burkardt, NINTLIB — Multi-dimensional quadrature, web page. http://people.sc.fsu.edu/~burkardt/f_src/nintlib/nintlib.html.
[38] M. Cole, Algorithm Skeletons: Structured Management of Paralell Computation, Pitman, 1989.
[39] OpenMP Architecture Review Board, OpenMP: Simple, Portable, Scalable SMP Programming, 1997. URL: www.openmp.org.
[40] D.H. Bailey, T. Harris, W. Shapir, R. van der Wijngaart, A. Woo, M. Yarrow, The NAS Parallel Benchmarks 2.0, Tech. Rep. NAS-95-020, NASA Ames Research Center, Dec. 1995, http://www.nas.nasa.org/NAS/NPB.
[41] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, Wiley-Interscience, New York, NY, 1991, ISBN: 0471503361.
[42] K.E. Batcher, Sorting networks and their applications, in: Proceedings of AFIPS Spring Joint Computing Conference, vol. 32, 1980, pp. 307–314.