A Language for Object-Oriented Parallel Programming
Targeted at Cluster Computing Platforms

Eduardo Gurgel Pinho, Francisco Heron de Carvalho Junior

Abstract. In object-oriented (OO) 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 and encapsulation of concerns in OO languages, since a parallel computation cannot be encapsulated in an individual object. For reconciling object-orientation and distributed memory parallelism, we propose PObC++, a OO language that introduces innovative ideas for object-oriented parallel programming (OOPP).

1. Introduction

The dissemination of distributed-memory parallel computing platforms, such as clusters and grids, has motivated the emerging of a new class of large scale applications from computational sciences and engineering, which pose stronger requirements of productivity and high performance for development tools [Post and Votta 2005]. Unfortunately, parallel programming is still hard to incorporate into usual software development platforms [Bernholdt D. E. et al. 2004]. Automatic parallelization approaches are viable only for special cases, such as in parallel scientific computing libraries [Dongarra et al. 2003], not reaching both flexibility and efficiency of explicit message-passing. Other promising high-level approaches, such as skeletal programming [Kuchen and Cole 2006], have not achieved the expected dissemination. This is the main reason for the popularity of MPI (Message Passing Interface) [Dongarra et al. 1996] among parallel programmers.

In the context of corporative applications, object-oriented (OO) programming 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 languages, tools, and techniques to overcome the software crisis at the end of the 1960’s [Dijkstra 1972]. Many programming languages have been proposed to support OO, such as: C++, Java, C#, SmallTalk, Ruby, Objective-C, and so on. Despite their success in the software industry, object-oriented languages are not yet popular in HPC, dominated by Fortran and C, mainly due to the performance overheads caused by high abstraction mechanisms supported by these languages. When parallelism comes onto the scene, the situation is worse, due to the lack of elegant ways to incorporate explicit message-passing parallelism to these languages without breaking their main principles and features, mainly functional independence of objects and their encapsulation.

This paper presents PObC++, yet another parallel extension to C++ that introduces a new style of OOPP (Object Oriented Parallel Programming) that may easily be applicable to any object-oriented language. The decision to support C++ comes from the
wide acceptance of C++ in some HPC niches. The main premise that guided the design of POB\textsc{c++} was the preservation of the basic object-orientation principles while introducing a style of programming very close to MPI. Therefore, we propose a design where intrinsically parallel objects distinguish process interaction message-passing (intra-object), by using channel-based communication, from object coordination message-passing (inter-object), usually done by method invocation.

Section 2 overviews the main approaches to support parallelism in object-oriented languages. Section 3 presents the main concepts and intuitions of OOPP, showing how it is incorporated in POB\textsc{c++}. Section 4 discusses how the POB\textsc{c++} prototype has been implemented. Section 5 presents an example of a parallel program written in POB\textsc{c++}, comparing its performance with a C++/MPI implementation. Finally, Section 6 will present our conclusions, describe ongoing research, and plant ideas for further initiatives.

2. Parallelism Support in Object Oriented Languages

Most of programming languages support distributed-memory parallel programming by using MPI (message-passing Interface) [Dongarra et al. 1996], a portable interface of subroutines for fast message-passing among processes in a distributed memory architecture. Implementations of MPI for object-oriented languages have been developed. Some of these implementations are simple wrappers for native MPI implementations [Baker et al. 1998, Baker 2000, Mintchev 1997], whereas other ones adopt an object-oriented style [Zhang et al. 2005, Douglas and Matthias 2010, Gregor and Lumsdaine 2008]. Both approaches present two drawbacks. Firstly, they go against the original MPI designers 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, in a language where each object should be functionally independent, addressing separate application concerns.

Charm++ [Kale and Krishnan 1993] is a library for message-passing on top C++, portable across distributed and shared-memory parallel architectures. A program is built from a set of parallel processes called \textit{char}es, which can be dynamically created by other \textit{char}es. They can communicate via explicit message-passing and share data through special kinds of objects. The placement of \textit{char}es is defined by a dynamic load balancing strategy implemented in the runtime system. \textit{Char}es are special objects, bringing the drawback of using objects to encapsulate processes instead of concerns.

A common language design, supported by Java\textit{P}arty [Philippsen 1997], Paro\textit{C}++ [Nguyen and Kuonen 2003], POP-\textit{C}++ [Nguyen and Kuonen 2007], relies on the common-sense idea of deploying objects, representing processes, across the nodes of a distributed architecture, 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 serve as processes, whereas most of the parallel algorithms assume peer-to-peer relations among processes. For this reason, Paro\textit{C}++ proposed forms of asynchronous method invocations in order to improve programmability of common process interaction patterns. POP-\textit{C}++ extended Paro\textit{C}++ for grid computing.
Figure 1. Processes vs. Objects

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 [Aridor et al. 2000]. 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 to distribute 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 the languages X10 [Sarkar 2005], Chapel [Chamberlain et al. 2007], and Fortress. Such languages have been developed under the HPCS (High Productivity Computer Systems) program of DARPA [Lusk and Yelick 2007] since 2002. The HPCS program has two goals: to boost 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. Each language has its parallel semantics to express task and data parallelism through different forms, such as: asynchronous method invocation, explicit process spawn, dynamic parallelism to handle “for loops” and partitioned arrays across different locations.

3. Towards a Language for OOPP - Intuitions and Concepts

We found that the main reason for the difficulties was in reconciling object-orientation and parallel programming in HPC programming languages in the usual practice to mixture concerns and processes in the same dimension of software decomposition [Carvalho Junior et al. 2007]. In distributed-memory parallel programming, a concern must be implemented by a team of processes. Consequently, an individual object must be a distributed entity, residing in a set of nodes of a parallel computing platform.

The Figure 1 helps to clarify our ideas. On the left-hand side, the common practice of parallel programming in object-oriented languages is illustrated, where individual objects execute in a single memory 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 encapsulation of objects. On the right-hand side, the practice that we argue to be the best suited one for object oriented parallel programming (OOPP) is depicted, focusing on concerns. Objects that cooperate to implement a parallel concern now constitute a parallel object (pob). Each individual object is a unit of the pob. Application concerns are now encapsulated in a pob, where parallel interactions are no longer clandestine. In fact, parallel interaction and object coordination are now differentiated on different hierarchical levels, leading to the concepts of intra-object and inter-object communication. Intra-object communication may be performed using channel-based message passing, which is better suited for parallel interaction between peer units, whereas inter-object communication may still use the
```cpp
class MatrixMultiplier {
public: int compute();

unit Manager {
private: int **a, **b, **c; // pointers to n x n matrices
int n;
public: void set_ab(int n, int **a, int **b)
{ n = n; a = a; b = b; }
int** compute()
{ channel.scatter<Manager>(a, n, n);
channel.scatter<Manager>(b, n, n);
channel.gather<Manager>(c, n, n);
return c;
}
};

unit Cell [i:n] [j:n]
{ private: int a, b, c = 0;
int* compute()
{ // receiving input matrices elements
channel.scatter<Manager>(a);
channel.scatter<Manager>(b);
// initial alignment
channel.sendCell(i+j, i, j, a);
channel.sendCell(j-i, j, b);
b = channel.recvCell(i-j, j+1);
// start systolic calculation
c += a * b;
for (k=0; k < n-1; k++)
{ channel.sendCell(i+1, i, j, a);
a = channel.recvCell(i-1, j, a);
channel.sendCell(i, j+1, b);
b = channel.recvCell(i, j-1, b);
c += a * b;
}
// send the resultant matrix element
channel.gather<Manager>(c);
return &c;
}
};

class Main {
public: int main();

unit Root {
int main()
{ int a, b;
MatrixMultiplier::Manager *m = new MatrixMultiplier::Manager();
m->channel.initialize();
read_input(&n, &a, &b);
set_ab(n, &a, &b);
int* c = m->compute();
write_output(n, c);
}
}
unit Peer [j:n]
{ int main()
{ MatrixMultiplier::Cell *c = new MatrixMultiplier::Cell();
c->channel.initialize();
int* my_c = c->compute();
printf("I calculated %ld \n", *my_c);
}
}

Figure 2. The MatrixMultipler(a) and MatrixMultipler(b) Classes
```

usual method invocations.

Based on the above observations and premises, we proposed a language for OOPP, called POoC++, since it is a parallel extension to C++, implemented on top of MPI for enabling process instantiation and synchronization. The reason for adopting C++ relies on its high acceptance and dissemination among applications in the HPC domain, mainly in computational sciences and engineering, due to its high performance. However, the parallel concepts introduced in C++ may be easily introduced in Java or C#, the two most used programming languages in general application domains.

We argue that POoC++ differs from their counterparts in the following aspects: objects maintain encapsulation of concerns since each unit of a pob can address the role of a process with respect to a parallel concern; objects send messages to other objects only by usual method calls, avoiding backdoor communication ports through low level message passing; full explicit parallelism is supported by means of an explicit notion of process and intra-object message-passing communication, providing full control over parallel processing concerns (load balancing, data locality, and so on); adherence to a parallel programming style that has been established since the mid 1990’s (i.e. MPI).

The following paragraphs will introduce the main concepts and abstractions be-
hind OOPP, by using the example of pob class of Figure 2, written in PObC++, which im-
plement a well-known algorithm for matrix multiplication, based on the systolic pattern of
synchronization among parallel processes. Remember that PObC++ attempts to reconcile
full-explicit message-passing parallel programming, as in MPI, with object-orientation in
its essential form, by introducing a minimal set of new concepts and abstractions. For
this reason, many pragmatic decisions have been made when adopting the programming
style of MPI, but without breaking the main principles of 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 yet for parallel programmers.

3.1. Parallel Objects

A Parallel Object (pob) is defined as a set of units, each one residing in a processing
node of a distributed memory parallel computing platform. A pob is an object in the
pure sense of object-oriented programming, addressing some application concerns and
communicating with other objects by means of messages (method invocations). Distinct
parallel objects may reside in distinct subsets of processing nodes, overlapped or disjoint.

Parallel Object Classes In the context of OOPP, a class is a prototype of a pob, which
specifies their units, attributes and methods. Attributes and methods may be defined in
the overall scope of the class or in the scope of some unit. Class attributes have indepen-
dent copies in each unit, whereas unit attributes only may only be accessed by methods
declared within the unit scope. Class methods are called parallel methods. A deeper
discussion about parallel methods and unit methods will be presented in Section 3.2.2.

Inheritance between pob classes is similar to inheritance between usual C++
classes, with the restriction that all units of the superclass must be declared in the subclass
with the same names.

In the example of Figure 2(a), the class of parallel objects named MatrixMultiplier
has a unit named Manager and an enumeration of \( n^2 \) units named Cell\([i, j]\), for indexes
\( i \) and \( j \) varying from 0 to \( n - 1 \), are employed. In fact, the cell units form a square mesh.
The indexes \( i \) and \( j \) are implicitly declared as private integer variables.

The manager distributes the elements of the two input square matrices \( (n \times n) \),
named \( a \) and \( b \), among the set of cells. Cells receive their corresponding elements from
the matrices \( a \) and \( b \), then they calculate their corresponding elements of the square matrix
\( c \).

3.2. Communication and Synchronization

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

- *inter-object communication*: messages exchanged between parallel objects, im-
  plementing the orchestration among the set of application concerns, concretely
carried out by 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 objects;
• *intra-object communication*: messages exchanged among the units of parallel objects, usually by means of message-passing, defining peer-to-peer relationships among units of a parallel object. Such messages define the interactions among application processes, required by most of parallel algorithms.

In usual parallel approaches of OO languages, there is no clear distinction between such kinds of messages. Consequently, one of the following approaches is adopted:

• parallel synchronization and communication is implemented by means of method invocations between objects, which is inappropriate for parallel programming, since method invocations induce 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 first approach is the main source of difficulties to attack the $M \times N$ coupling problem [Bertran et al. 2005], where two sets of $N$ and $M$ processes, possibly residing in distinct clusters, want to share some data structure whose distribution may differ in the two sets. Using OOPP, such coupling could be implemented by a pob object, whose $M + N$ units are scattered among all the processes. In the second approach, objects tend to lose their functional independence. Therefore, they cannot be analyzed in isolation, breaking important modularity principles behind object-orientation.

### 3.2.1. Intra-object communication: Message-Passing among Units

Intra-object communication, between units of a pob, is implemented by means of message-passing through a *channel*. A channel is a primitive pob in PObC++, whose methods are the operations for communication among the units of a pob. A single channel exists in a pob, accessed as a global attribute using the reserved word `channel`.

The communication operations supported by a channel are that ones supported by communicator objects in Boost.MPI, including collective communication operations. For that, each unit of a pob needs an identification. In PObC++, the identification of a unit is defined by its type and *enumeration index*, a sequential number that is implicitly mapped to an MPI rank by the compiler. In MPI, a rank is a number from $0$ to $N - 1$ used to identify a process in the scope communicator it participates in. This is illustrated in the code of the parallel method `compute` of the `MatrixMultiplier` class.

In OOPP, objects cannot be transmitted through channels, but only values of non-object data types, primitive and structured ones. This is not intended to simplify the implementation effort, since Boost.MPI gives support for object transmission through channels. In fact, we believe that the possibility of transmission of objects through communication channels is a source of overheads that are not explicit in the source code, due to serialization. We are evaluating the possibility of introducing primitives for object *migration* and *remote cloning* for the usual needs of object transmission. Migration differs from remote cloning because the reference to the object in the original memory space is lost after the operation. Using this approach, it is possible to optimize communication primitives to work only with raw data values, which is common in HPC applications.
3.2.2. Inter-object communication: Parallel Methods

Like C++, inter-object messages are implemented as method invocations of two kinds:

- **Unit methods** are defined in the scope of units, accessing only unit attributes. Since `channel` is a class attribute, communication may not be done in unit methods;
- **Parallel methods** are declared in the scope of the pob class. Inside each unit, an implementation of each parallel method must exist, which may access class and unit attributes. Therefore, intra-communication may occur through the channel.

The standard OOPP programming practice assumes that parallel methods are the synchronization and communication points of the parallel program. Thus, it is convenient that the calls to a parallel method performed by units of a pob occur at about the same time, avoiding excessive synchronization among units. As a corollary, the numbers of calls to the same parallel method by each unit must be equal. Obviously, such restriction can not be enforced statically. Therefore, the programmer must ensure the coherent use of parallel methods. 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. When necessary, the programmer may use the `synchronize` modifier in the declaration of parallel method signature, which instructs the compiler to create a barrier at the start of the parallel method. The method is only started when all units of the pob arrive at a call to the method.

3.2.3. Communication and Synchronization in the Example

In the example of Figure 2, there is a simple parallel method, `compute`, which involves all communication activities between the manager and the workers and among the workers, as well as the computation of the matrices product.

The code of `compute` illustrates calls to collective and point-to-point communication primitives of the channel object. Notice how the cell units address their neighbor cells, required by implementing the data movements in the systolic computation, using arithmetic on the enumeration indexes $i$ and $j$.

3.3. Instantiation

A pob is instantiated by a typical C++ constructor call for each unit constructor of the pob. Units are referred to using the convention `{class_name}`::`{unit_name}`. As usual, in C++, the user may define constructors for any pob class, except for C++ copy constructors.

The instantiation of a pob is illustrated in the `Main` class of Figure 2(b), the entry class of a POBc++ program that instantiates a pob from the class `MatrixMultiplier`. To this end, it declares a `main` parallel method for each unit, representing the processes of the application. The `MatrixMultiplier` object is created by calling the unit constructors inside each enclosing unit. After that, the program must call the synchronized parallel method `channel_initialize`, implemented by any pob, before any call to a parallel method, causing the instantiation of the channel.
4. Implementation

PObC++ is an open source project hosted at http://pobcpp.googlecode.com, released under the BSD license, composed of three parts: compiler, standard Library, and lightweight runtime environment. The next paragraphs present brief discussions about them, helping the readers to understand the development decisions behind their implementation.

4.1. Compiler

In order to achieve a fast and reliable prototypic implementation, a source-to-source compiler written in C++ was designed by modifying Elsa, an Elkhound-based C/C++ parser. Elkhound [McPeak 2003] is a parser generator that uses GLR parsing algorithm, an extension of LR to handle nondeterministic and ambiguous grammar. Particularly, Elkhound’s algorithm uses the type checking process to disambiguate and generate a valid AST (Abstract Syntax Tree). The basic steps are:

1. **Input**: PObC++ source;
2. Scanning and parsing using Elsa;
3. Type checking and disambiguation;
4. Transforming PObC++ AST in order to build a valid C++ AST;
5. **Output**: C++ source.

The modifications performed during the fourth step transform each unit of a pob class into a valid C++ class, augmented with meta-information about the units defined within the pob class. These adjustments generate only C++ valid code and do not interfere with the rest of the program. Essentially, only the code containing pob 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++.

4.2. Standard Library

The standard PObC++ library is composed of channel implementation and helper functions to initialize and finalize the environment. All functions are defined in the scope of the `namespace pobcpp`.

As we have described in Section 3, instead of using sequential integer ranks to address pob units in communication operations through channels, like MPI, the unit type and its respective enumerations are used. Our implementation uses a C++ feature named RTTI (Run-time Type Information) to grab objects’ data type information and address to the correct unit. This approach eliminates the cumbersome work of using ranks to identify pob units in channel operations.

The code below contrasts the what basic send and receive operations must be called in PObC++ and pure MPI:

- **PObC++**:
  ```
  channel.send<Unit_Type>(index_enum, data, tag);
  DataType data = channel.recv<Unit_Type, DataType>(index_enum, tag);
  ```
• MPI:
  MPI_Send(buffer, size, mpi_datatype, rank_dest, tag, comm);
  MPI_Recv(buffer, size, mpi_datatype, rank_src, tag, comm, stat);

It is important to notice the datatype information removal on channel.send. The use of Boost.MPI is the main reason for such modification, since some data structures and functions implemented in this library have been used to facilitate the standard library development and future modifications.

4.3. Light Weight Runtime Environment

The runtime environment is responsible for instantiating the pobs and their channels. In fact, a channel is an MPI intra-communicator which defines the communication scope of the pob. For this reason, the instantiation of pobs is a parallel call. After instantiation, each unit can compute and communicate with other units of the same pob without any runtime participation. Therefore, if we ignore the pob initialization, no extra computation will be executed by the environment (light weight). Just the algorithm itself and MPI calls (through the channel) will be processed, not adding any performance overhead.

In our implementation, the method channel.initialize, explicitly called by the programmer for units of a pob that desire to exchange messages, implicitly execute two MPI collective operations. Firstly, units exchange their typing information, possible enumerations, and global MPI ranks. Then, each unit determines whether the group can be safely created and then execute the MPI operations for creating the channel. After channel.initialize returns, the computation may continue without any runtime interference.

5. Case Study: Farm-Based Parallel Numerical Integration

Aiming to provide the first example of a PObC++ program, as well as to evaluate its performance compared to pure C++/MPI, we have implemented a class of parallel objects representing numerical integrators, called Integrator\(^1\). Notice that the case study does not aim to compare productivity of programmers between PobC++ and pure C++/MPI.

5.1. The Farm Class

The class Integrator inherits from the abstract class Farm, which encapsulates the interaction pattern between the manager and the workers. Farm is a kind of skeleton. The term algorithmic skeleton was firstly coined by Murray Cole two decades ago to describe reusable patterns of parallel computations whose implementation may be tuned to specific parallel execution platforms [Kuchen and Cole 2006]. Skeletons have been widely investigated by the academic community, being considered a promising approach for high-level parallel programming. This section tries to evidence that skeletal programming is a natural programming technique supported by OOPP.

The Farm class is presented in Figure 3(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

\(^1\)The complete source code of the case study is available at http://pobcpp.googlecode.com.
class Farm {  
  public:  
    void synchronizeJobs();  
    void synchronizeResults();  
  
  unit Manager {  
    private:  
      Job∗ allJobs;  
      Result∗ allResults;  
    public:  
      void addJobs(Job∗ job);  
      Result getNextResult();  
      Result∗ getAllResults();  
      virtual void∗ packJobs(Job∗ jobs);  
      virtual Result unpackResult(void∗ result);  
  };  
  
  unit Worker [i:n] {  
    private:  
      Job∗ localJobs;  
      Result∗ localResults;  
    public:  
      void performJobs();  
      virtual Result work(Job job);  
      virtual Job unpackJobs(void∗ jobs);  
      virtual void∗ packResult(Result result);  
  };  
};

class Integrator : public Farm<IntegratorJob, double> {  
  public:  
    unit Manager {  
      private:  
        int inf, sup, 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 generateSubproblems();  
        double combineSubproblemResults();  
        void∗ packJob(IntegratorJob∗ jobs);  
        double∗ unpackResult(void∗ result);  
    };  
    
    unit Worker [i:n] {  
      private:  
        int numberOfPartitions;  
        int nextUnsolvedSubproblem;  
        double (∗function)(double∗);  
      public:  
        Worker(double (∗f)(double∗, int tol, int cmp),  
          function(f), numberOfPartitions(cmp),  
          nextUnsolvedSubproblem(0), tolerance(tol) {}  
      public:  
        double work(IntegratorJob∗ job);  
        IntegratorJob∗ unpackJob(void∗ jobs);  
        void∗ packResult(double∗ result);  
    };  
};

(a) The Farm Class   (b) The Integrator Class

Figure 3. Case Study Main Classes

In fact, the work of a worker is a local procedure in its nature, reflecting the fact that 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. 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. We are still working on the design of a type system to deal with such adaptability features dynamically, based on our experience on the design of a type system for parallel components with similar characteristics.

The methods setJob, getNextResult and getResults 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. getNextResult 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 setJob before beginning to sending 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.
parallel class IntegratorMain {
    public: int main();
}

unit Root
{
    int main()
    {
        #pragma omp parallel
        {
            Integrator::Manager *m = new Integrator::Manager(0.0, 1.0, 5, 2);
            m->channel_initialize();
            double result;
            #pragma omp sections
            {
                #pragma omp section
                m->generate_subproblems();
                #pragma omp section
                m->synchronize_jobs();
                #pragma omp section
                m->work();
                #pragma omp section
                result = m->combine_subproblems_results();
            }
            cout >> "Result is ", result;
        }
    }
}(...)

unit Peer[j:n]
{
    int main()
    {
        #pragma omp parallel
        {
            Integrator::Worker *w = new Integrator::Worker(function, ID=-5, 16);
            w->channel_initialize(j);
            #pragma omp sections
            {
                #pragma omp section
                w->synchronize_jobs();
                #pragma omp section
                w->work();
                #pragma omp section
                w->synchronize_results();
            }
        }
    }
}(...)

Figure 4. Case Study Main Classes

In order to use a farm, the programmer must extend the Farm class, by inheritance, by implementing the virtual unit methods packJob, unpackResult, packResult, unpackJob, 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 channel. Therefore, the Result and Job types must be defined. Remember that objects cannot be transmitted through channels, 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 being solved.

5.2. The Integrator Class

The Integrator class, as declared in the header file integrator.h, is presented in Figure 3(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 addJob in order to feed the farm’s job list, and combine_subproblems_results, which will call getNextResult or getResults to retrieve the numerical results calculated by the workers, which will be added.

5.3. The Main Program

The main class of the integrator program is presented in Figure 4. Notice that it tries to exploit all the concurrency available in the farm, by forking threads using OpenMP [OpenMP Architecture Review Board 1997], a library of subroutines and preprocessor directives for shared-memory parallel programming.

5.4. Performance Evaluation

The Table 1 summarizes the results obtained by executing the PObC++ numerical integrator and its best tuned pure C++/MPI version on the cluster Castanhão2, installed

---

2Castanhão contains 28 nodes Intel Xeon 1.8GHz, 2GB Memory per node, and 100Mbs network (http://castanhao.lia.ufc.br). At the time of the experiment, only 12 nodes were available.
Table 1. Performance Results Summary (Execution Time and Speedup)

<table>
<thead>
<tr>
<th></th>
<th>POoC++</th>
<th>C++/MPI</th>
<th>δ</th>
<th>POoC++</th>
<th>C++/MPI</th>
<th>δ</th>
<th>POoC++</th>
<th>C++/MPI</th>
<th>δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.99 (×1.0)</td>
<td>4.98 (×1.0)</td>
<td>0.2%</td>
<td>90.7 (×1.0)</td>
<td>89.7 (×1.0)</td>
<td>1.2%</td>
<td>1633. (×1.0)</td>
<td>1628. (×1.0)</td>
<td>0.3%</td>
</tr>
<tr>
<td>2</td>
<td>2.47 (×2.0)</td>
<td>2.50 (×1.9)</td>
<td>-1.1%</td>
<td>44.9 (×1.9)</td>
<td>44.9 (×2.0)</td>
<td>2.6%</td>
<td>826. (×2.0)</td>
<td>814. (×2.0)</td>
<td>1.5%</td>
</tr>
<tr>
<td>4</td>
<td>1.27 (×3.8)</td>
<td>1.25 (×3.9)</td>
<td>1.8%</td>
<td>23.1 (×3.9)</td>
<td>22.7 (×3.9)</td>
<td>1.6%</td>
<td>411. (×3.9)</td>
<td>408. (×4.0)</td>
<td>0.8%</td>
</tr>
<tr>
<td>8</td>
<td>0.66 (×7.4)</td>
<td>0.64 (×7.6)</td>
<td>2.4%</td>
<td>11.7 (×7.7)</td>
<td>11.5 (×7.8)</td>
<td>1.3%</td>
<td>207. (×7.8)</td>
<td>205. (×7.9)</td>
<td>0.7%</td>
</tr>
<tr>
<td>seq</td>
<td>4.84s</td>
<td>89.4s</td>
<td>1621s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The experiment aimed to collect empirical evidences about the performance equivalence of POoC++ and C++/MPI. We argue that such a conclusion could be made only by observing that the parallel programming model of both versions are equivalent. In fact, the POoC++ version is compiled for a C++/MPI program that is virtually equivalent to the best tuned C++/MPI version, excepting for call indirections from the channel interface to the MPI subroutines. The essential difference is the better object-oriented properties achieved by POoC++, with potential gains in modularity, abstraction, and usability.

6. Conclusions and Further Works

The support for OOPP, the style of object-oriented parallel programming proposed in this paper, mainly regarding their validation as a tool for reconciling high productivity of object-orientation and high performance of parallel processing, is a long-term project, the first product of which was presented in this paper. POoC++ is only the first prototype of OOPP. For this reason, this paper aimed to introduce and to evaluate the design and implementation of POoC++, which serves to introduce and to validate the OOPP concepts.

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 coherently 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 OO programming, 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.
7. Acknowledgments
We thank CAPES and CNPq (grant number 480307/2009-1) for their financial support.

References


