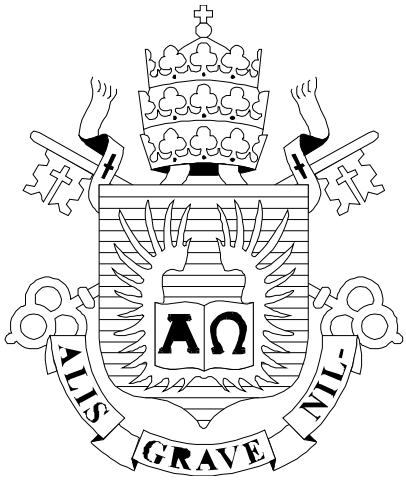

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 20/09

On Object and Component Design Approaches for Parallel Programming

Paulo Rogério da Motta Junior
Noemi de La Rocque Rodriguez
Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 20/09 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena July, 2009

On Object and Component Design Approaches for Parallel Programming

Paulo Rogério da Motta Junior Noemi de La Rocque Rodriguez
Carlos José Pereira de Lucena

{pjunior,noemi,lucena}@inf.puc-rio.br

Abstract. The evolution of the field of programming traditionally trades performance for more powerful abstractions that are able to simplify the programmer's work. It is possible to observe the effects of this evolution on the parallel programming area. Typically parallel programming focuses on high performance based on the procedural paradigm to achieve the highest possible throughput, but determining the point in which one should trade performance for more powerful abstractions remains an open problem. With the advent of new system level tools and libraries that deliver greater performance without programmer's intervention, the myth that the application programmer should optimize communication code starts to be challenged. As the growing demand for large scale parallel solutions becomes noticeable, problems like code complexity, design and modeling power, maintainability, faster development, greater reliability and reuse, are expected to take part on the decision of which approach to use. In this paper, we discuss the use of new paradigms that provide higher-level abstractions and may provide many benefits to parallel programming developers. We argue that the decision of whether or not one should choose to apply these techniques on an application project remains subjective and depends on many factors related to time to delivery, programmer experience, and complexity, among others.

Keywords: Parallel Programming, Software Engineering, Productivity.

Resumo. A evolução do campo de programação tradicionalmente troca desempenho por abstrações mais poderosas capazes de simplificar o trabalho do programador. É possível observar os efeitos dessa evolução na área de programação paralela. Tipicamente, programação paralela se concentra em alto desempenho baseado no paradigma procedural para atingir o mais alto rendimento possível, porém determinar o ponto em que deve-se trocar desempenho por abstrações mais poderosas continua um problema em aberto. Com o advento de novas ferramentas e bibliotecas de sistema que fornecem melhor desempenho sem a intervenção do programador, o mito de que o programador da aplicação deve otimizar o código de comunicação começa a ser questionado. De acordo com a crescente demanda por soluções paralelas de larga escala se tornam evidentes, problemas como complexidade de código, poder de modelagem e projeto, manutenibilidade, desenvolvimento rápido, maior segurança e reuso, deverão ser considerados quando for necessário decidir que abordagem usar. Nesse artigo, discutimos o uso de novos paradigmas que fornecem abstrações de mais alto-nível e que podem prover muitos benefícios para desenvolvedores de aplicações paralelas. Argumentamos que a decisão de usar ou não essas técnicas em uma aplicação permanece subjetiva e depende de muitos fatores relacionados a tempo para entrega, experiência dos programadores e complexidade entre outros.

Palavras-chave: Programação Paralela, Engenharia de Software, Produtividade.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

Table of Contents

1 Introduction

Parallel programming can be seen as a great mean for performance improvement, especially for large scientific domains such as weather forecast, weather simulation, Chemistry and Physics simulation, and so on. However, writing parallel programs demands specific design approaches due to the size of the collection of items that compose a parallel environment.

Not only the algorithm must be designed for a parallel environment, but also communication functions, which are non-functional aspects, must consider the underlying structure items such as network topology, memory locality, and processor distance, among others. All this, when taken together greatly increases the complexity of parallel programming.

Considering all the body of knowledge that Software Engineering has accumulated in the areas of complexity management, software development methodologies, and software testing, for the domain of information systems and business applications [PRESSMAN,1997][SOMMERVILLE,2001][PETERS and PEDRYCZ,2000], it is reasonable to consider applying these techniques for the sake of the evolution of parallel programming. Notably, the use of object-oriented and software component techniques have been extensively studied by the Software Engineering community and are known to improve productivity and decrease complexity through the use of high level abstractions. These lessons originated mostly from the sequential software development area, and much research is needed, as we will present in this paper, on the benefits that might be achieved by their use on the parallel programming arena. It should be understood upfront that these techniques impact performance due to the high level of their sophisticated abstractions.

Although performance will continue to be a major driver of the parallel programming arena [FOSTER,1995], we should start by considering that maybe some performance loss is acceptable when facing other important and hard-to-manage issues such as software complexity and size, version control, maintainability, reliability, faster development, greater design, modeling expressiveness, and so on.

The authors of [CARVALHO and LINS,2005] present an evolution cycle for parallel programs where they consider the steps from language birth to maturity. They consider that it takes three phases to achieve the level necessary to handle the complexity associated with large scale applications. The first one is characterized by the search for performance, the second phase is characterized by the search for portability and the third and last phase is characterized by the search for higher-level abstractions that enable developers to solve large scale applications. According to the authors we are starting the third phase of parallel programming, and so the time has come to identify which will be the abstractions and supporting environments that will allow programmers to solve larger problems with less effort.

What we can expect then for upcoming research, is the adaptation of software engineering techniques for the specificities of the parallel programming field. Probably, a specific discipline of parallel software engineering will need to be developed to analyze all that has been proposed for sequential software engineering until now, and validate what can be applied to the parallel application development. When necessary, a corresponding technique or method should be proposed so that we can achieve the same level of maturity and productivity on parallel application projects.

Given the limited number of published material dedicated to exploring the Software Engineering aspects of parallel programming, specifically regarding the use of objects and components, the objective of the present work is to address these issues by proposing guidelines that should be considered when writing parallel programs. After exploring the technical and logical concepts regarding a each proposed guideline some questions are presented in a bullet list style to help reasoning about the concepts incorporated in each guideline.

For the purpose of the present study, the scope of parallelism considered is the use of clusters and/or grids composed of workstations that deliver processing power to users at a low costs.

This kind of architecture has been widely used in many high performance computing centers and institutions. For other types of high performance architectures, such as shared memory machines, most of what is presented here also holds, but other tools that explore hardware possibilities must also be considered. For the cluster and grid contexts, most of the solutions are based primarily on software since hardware tends to be independent among the nodes.

Among the types of approaches that are considered in the present paper, we emphasize tools and libraries that improve software performance in an automated fashion, supporting tools and environments for programming, software integration and wrapping, compiler techniques and the use of higher level abstractions to improve the expressiveness of models.

This paper is structured as follows. Section 2 will present the approaches to parallel programming that we consider; section 3 presents a proposed set of guidelines on how to quantify the choices; section 4 presents the conclusions.

2 Parallel Programming Approaches

For the scope of this paper, let us consider parallel computers that are composed of independent machines interconnected by a network with no shared memory. This type of environment provides a low-cost alternative for parallelism, making high performance computing available for different users, even at the undergraduate level. For this type of environment, the most common approach is to use message passing as the mean of communication among the processes that participate of a given computation. We will consider two classes of parallel programming: 1) Traditional Procedural Programming and 2) Object and Component Programming. This classification takes more into consideration the Software Engineering viewpoint than the Parallel and Distributed Software Development usual approach. This is justified because our main goal is to achieve a generic model rather than a specific implementation for a small set of parallel problems. However, this classification does not bias our vision when it is applied to a specific problem, it only requires that more details be considered.

For the sake of clarity, environments that have little or no concern with Software Engineering aspects are considered a subgroup of the first class even if the programs are not truly structured.

As described in [ANDREWS,2000], a parallel program is characterized by the use of many processes to solve a given problem in less time than it would take to solve the sequential version, or to solve larger instances of the same problem in the same amount of time. Either way, we can derive that an important factor regarding parallelism is performance, since its main goal is to improve processing time against sequential programming. However, as discussed in [FOSTER,1995], performance alone can not be considered the only metric for parallel programming. Some other aspects like parallel efficiency, memory requirements, throughput, latency, input/output rates, network throughput, design costs, implementation costs, verification costs, potential for reuse, hardware requirements, hardware costs, maintenance costs, portability, and scalability should be considered when quantifying a parallel program.

2.1 Traditional Procedural Programming

The most common and widespread paradigm is to use the message passing interface in a structured procedural style. Both send/receive and collective operations, which focus on groups of processes, are in use. This technique is diffused due to its many libraries for procedural languages like C and FORTRAN, which are in major use among scientific applications. We also should consider that many other areas of science like Physics and Chemistry have professionals that develop their own models and programs independently of software engineering professionals [GROPP, LUSKL and SKJELLUM,1999][DANIS,2006].

These scientist-programmers tend to learn parallel programming in an on-the-job manner and most of the time refuse to incorporate programming best practices into their daily work. When

they do so, they end up getting away from their research objectives and become closer to a domain specific programmer.

As we can see in [DANIS,2006] it is sometimes possible to have scientists and programmers working together both as a team or in a consultant/client relationship.

2.2 Object and Component Programming

The use of objects in parallel programming with C++ and Java was first explored by wrapping message passing interface into objects, but dependent and bonded to the structured procedural vision. Notably, there are C++ implementations of MPI available, however, due to its compatibility with the C programming language, the primitives are used directly and not in an object oriented manner [GROPP, LUSKL and SKJELLUM,1999].

Later, the object paradigm started to be explored with MPI implementations both in C++ [GROPP, LUSKL and SKJELLUM,1999], Java[BAKER et al.,1999][MOHAMED et al.,2002], Ruby [ONG,2002] and Python[MILLER,2002], and also with parallel environments that allow for message passing communication styles based entirely on objects. In these works, techniques exploring groups of objects and group method calls were proposed.

The availability of studies indicating the real effort to adopt this style of parallel programming is still incipient, but due to the quantity of tools and frameworks to help in this direction we can expect to have some data being published soon.

Our major interest rests on techniques that can deliver complexity management and design and modeling capabilities together with separation of concerns for larger problems.

2.2.1 Objects Applied to SPMD

In [BADUEL, BAUDE and CAROMEL,2005], the authors present the experience of applying the SPMD[1] programming model to an object oriented environment. It is presented as an evolution from typed group communication [BADUEL, BAUDE and CAROMEL,2002] where a group of objects exposes a type that can be used by the client object. The authors use the idea of topology and neighborhood so that the notion of neighbor position is encapsulated instead of controlled by the user.

One characteristic that is worth mentioning is that the support environment used in [BADUEL, BAUDE and CAROMEL,2005] implements the use of active objects; this ensures that each object that takes part in a computation is granted its own individual thread of execution. When a method is invoked, it may execute as an asynchronous call which improves the overall execution time since method dispatching may result in an almost immediate parallel execution.

Barriers are provided, not only for the set of objects but also for more local scopes. It is possible to synchronize a group of objects that are local to a certain node and even to synchronize on a certain group of methods, which indicates that the processing can only continue once all the methods are executed.

These facilities enable the user to focus on the application at hand instead of worrying too much about the details of the communication and infrastructure needed by the program. Also, it becomes possible to even inherit some behavior defined at super classes. The relations and interactions between objects may be isolated and the computing part of the algorithm may be programmed almost as if it was a sequential version of the system.

Even when the underlying infrastructure does not offer new capabilities like those present in [BADUEL, BAUDE and CAROMEL,2005] and [BADUEL, BAUDE and CAROMEL,2002], the user can benefit from the object technology as evidenced on the JOPI [MOHAMED et al.,2002] environment that enables the programmer to move from a simple message passing paradigm to a more powerful object passing paradigm. The result is that communication code is

1 SPMD refer to Single Program Multiple Data where a single program runs on multiple nodes and each node receives a distinct subset of data to process.

separated from functional code, complex data structures are exchanged in a simple way and, since this tool is based on Java, heterogeneity is inherited by the use of the Java virtual machine that can be deployed on many different operating systems and architectures. Although this is not the ultimate solution for all programming problems, it greatly simplifies the task by allowing programmers to expose ideas in a more abstract manner. The study also presents data that shows that for small sized data exchange, from 1 to 16 KB, a simple C program with MPI achieves greater overall performance, however when data size increases, from 16 to 4096 KB, the overall performance tends to become almost equal. For a matrix multiplication problem, both performance plots are logarithmic, with the Java version achieving a slightly better performance as the number of processors increase.

An intermediary solution between objects and components is presented in [RENÉ and PRIOL,1999], where CORBA is used together with MPI to create SPMD behavior. The proposed tool uses a modified IDL to express the multiplicity of nodes that will compose the parallel object. MPI is used for object communication among the processing objects that are part of the collection. One benefit of this approach is to have legacy procedural code being wrapped into CORBA objects and serving new applications. Also, having MPI isolated in an underlying layer presents an opportunity to use some of the existing optimized versions of MPI like the ones proposed in [KARWANDE,YUAN and LOWENTHAL,2003] and [KE,BURTSCHER and SPEIGHT,2004].

Finally, a completely different approach, seen in [CHARLES et al,2005], is the design of a whole new object oriented language. In this work, the authors present X10, aiming at non uniform cluster computing. The idea behind the language is to have a new set of tools that are designed from scratch to handle the specificities that we find in parallel application development. The project offers a whole new programming model that presents new constructs offering functionality that we find on current libraries, but with new concepts that promise to promote better comprehension and productivity. The project is in its mid-stage and offers a compiler version together with a preliminary virtual machine. The team expects to have a fully implemented version in the near future.

2.2.2 Evolving into Components

As software grew larger, and the object oriented approach became insufficient to cope with complexity, the reuse of previous objects seemed a promising solution [SOMMERVILLE,2001] [SUGUMARAN, TANNIRU and STOREY,1999][YUAN, DUAN and LIU,2006]. Developers started to create units of software that were capable of deployment into certain types of container environments. According to [SZYPERSKI,2003], these deployable unities are called *components* and, in most cases, define a concrete and self-contained unit of work. For the purpose of creating bigger systems, the component approach helps to manage the complexity of having many objects communicating and interacting.

One of the major benefits of using components is the possibility of having large repositories of tested software that is guaranteed to work properly and may be reused [SOMMERVILLE,2001]. This characteristic improves reliability and reduces process risks associated with the software being developed.

Another benefit of the component model is that it is based on having some pieces of software running inside containers. Containers are responsible for handling component interactions and life cycle, but are not capable of any application-specific computation. This leads us to a model in which the functional code is not bound to infrastructure issues, being served by the container when necessary. Associating the model with parallel programming enabled environments allows to deliver the power of complex software without the intricate communication libraries details.

In [BAUDE et al.,2007], the authors present the use of *Collective Interfaces*, which are definitions of *types* in the object oriented sense of abstract data types, but that describe which strategies of parallelism are going to be used. Instead of writing the code that handles the interaction between objects, the user indicates what is necessary to achieve the goal and the environment uses the information to apply communication code among the objects that are part of a computa-

tion group. Parallel programming is then taken to a higher abstraction level where the user doesn't need to code the recurring communication patterns of code and focuses only on the problem that is being resolved by the parallel computer at hand. This could lead to more effective and clear algorithms given that the interactions are provided by the container.

As mentioned before, the authors of [CARVALHO and LINS,2005] also support the use of components as a mean of enabling higher abstractions for parallel programming languages. The idea of composing new modules allows applications with tested modules to be used as part of new composed modules.

Regarding the effort necessary to write a component-based version of an existing application, in [PARLAVANTZAS et al.,2006] the authors present a study conducted in order to change an object oriented parallel application into a component based one. The authors conclude that the change did not degrade the application's performance and qualitatively increased reuse by the creation of components. Noting that although the authors present a guideline for parallel application componentization, they assume that the original application is object-oriented already.

Finally, in [BIGOT and PEREZ,2007], the authors present another approach to achieve parallel component applications based on CORBA and MPI that is capable of modeling the different strategies of group operations. This approach differs from [BAUDE et al.,2007] in the sense that it provides an abstraction of the underlying infrastructure but does not provide its own communication infrastructure. In [BAUDE et al.,2007], parallelization is provided by the middleware with its own resources. Both approaches are valid and indicate that components may help manage software complexity.

3 How to Quantify?

Changing the programming paradigm of a certain area implies the same difficulty level for both sequential and parallel programming. However, because parallel programming is used by a certain class of users that are performance-driven, it may be a bit harder to expose the benefits of using higher abstraction levels when writing this type of applications. Nevertheless, it is important to consider the benefits that software engineering derived from the use of more sophisticated techniques allowing better programmer productivity and larger software life time for traditional sequential applications.

We present now a set of guidelines that should drive the user when choosing the correct tool for the project at hand. Some issues are quantifiable, but others are more subjective and can lead to tricky decision situations.

3.1 Performance

Parallel programming is traditionally focused on aspects such as execution time and scalability, and the performance of a system is often measured in these terms. However, one should always keep in mind that the absolute maximum achievable performance[2] may be difficult to obtain in terms of development techniques.

As mentioned before, a series of parameters are defined in [FOSTER,1995] that should be considered when defining the performance of a parallel program. Considering only execution time as the metric for every parallel application will lead the user to an over simplification and, many times, to a poor conclusion.

Regarding the approach to use when focusing on parallel programming, in [CARRIERO and GELERNTER,1989], the authors suggest that we should first try to develop a parallel program in a decomposed natural way, for example, allocating many processor nodes to the computation. If the implementation doesn't achieve the expected performance, we should iterate through the

2 When exploring the performance of parallel programs it is common to refer to the theoretical maximum performance which is not achievable in practice due to hardware physical limitations. However, for the scope of this work, we consider as the absolute maximum performance the top limit performance that can be achieved in practice.

code applying optimization techniques – that in turn will make the code less readable and maintainable – in order to achieve our best possible performance.

We should note tAs mentioned before, a series of parameters are defined in [FOSTER,1995] that should be considered when defining the performance of a parallel program. Considering only execution time as the metric for every parallel application will lead the user to an over simplification and, many times, to a poor conclusion.

Regarding the approach to use when focusing on parallel programming, in [CARRIERO and GELERNTER,1989], the authors suggest that we should first try to develop a parallel program in a decomposed natural way, for example, allocating many processor nodes to the computation. If the implementation doesn't achieve the expected performance, we should iterate through the code aphat there is a frontier delimiting code organization that will have to be crossed to achieve the final performance limit. But is this the performance needed for every parallel application? When using the traditional structured procedural approach, the ultimate performance solution will, in most cases, break the structure of the application [KERNIGHAN and PIKE,1999]. It is worth mentioning that, at present time, there are techniques that may decrease or even eliminate this problem with compiler optimizations.

When using object or component oriented approaches, one should consider the fact that these solutions use a higher degree of indirection and subroutine calls. Needless to say this will incur on performance loss. However, if the communication code is not mixed with the application code, it is much simpler to apply optimization techniques to both parts of the code without compromising the structure and organization of each part. It is also important to note that communication code should be optimized by system developers [BENTLEY,2000].

A modified version of the MPI library is presented in [KE,BURTSCHER and SPEIGHT,2004] implementing a compression/decompression scheme on MPI messages before sending and after receiving, that is totally transparent to the user. The authors claim a 98% improvement on performance on large sized messages exchange. This kind of approach may be well suited for use together with other performance improvement techniques allowing an increase that is independent from the programmer and relies on system developers.

Research shows ways of overcoming performance problems with the improvement of the supporting environment like in [KARWANDE,YUAN and LOWENTHAL,2003] where a MPI variant is capable of compiling communication code in a form that enhances the performance for switched clusters. On the other hand, in [TAN et al.,2003] and [FARAJ and YUAN,2005] we can see an approach based on code generation for parallel environments in order to achieve not only better quality and performance, but also better resource usage and programmer's productivity.

In [FARAJ,YUAN and LOWENTHAL,2006] and [VADHIYAR,FAGG and DONGARRA,2000] the authors explore a different path for improving the performance of collective operations. Both projects explore the automatic tuning of this type of operations by analyzing the environment where the application is going to run. This way, the algorithms used for the communication may be selected from a set of known algorithms using statistical information to drive this selection. Experiments showed that, for several cases, the performance achieved by these approaches were better to non-optimized versions. This type of research plays an important role in the field of performance improvement because the underlying hardware configuration may greatly influence the overall system performance. Furthermore, this type of analysis is better executed in an automatic way.

Either way, be it compiler based or code generated, we can infer from these studies that having experts optimizing libraries and infrastructure frameworks is more reliable, productive and promotes a higher degree of abstraction, freeing the application programmer from handling low-level interactions that in most cases deviate attention from the real problem. Moreover, it is hard to believe that developing custom communication code to handle low level details will be better if done by the application programmer than by library developers that are used to the intricate details of this kind of task [GORLATCH,2004].

When the functional and non-functional code are separated, we can even consider algorithm changes for better performance in a more natural way, since the communication code will not be exposed.

To decide when it is worthwhile to trade performance for other aspects like readability, maintainability and ease of use, one should take into account that there are parts of the algorithm that can not be parallelized, communication costs of the chosen infrastructure, the efficiency of the implementation that is achievable on the target architecture, and supporting tools that can automatically improve performance.

1. When is the achieved performance enough?
2. Is it possible to improve performance by applying changes on the non-functional code, or should we also consider changing the algorithm?
3. Considering that communication cost may decrease overall performance, would it be possible to improve the organization and maintainability by using a higher level abstraction like objects or components in a way that indirection can be covered by latency?
4. What kind of tools and libraries are available that may improve the overall performance in a automated way ?

3.2 Software Size

Dealing with software size is also an important issue to be considered as part of our guidelines. As presented in [McCONNELL,1993], as software grows bigger in size[3], the amount of effort spent on different tasks also changes correspondingly. Communication among team members, system testing and module integration, together with a bigger effort on architecture planning should be considered as factors when considering the methodology that will be used. The author presents a list of items that should be carefully considered when writing software. This list will lead to the level of formality that should be applied to the methodology. Among all the items proposed by the author, when writing parallel applications, one should pay attention to: equipment complexity, personnel assigned, criticality and programming languages. These factors will contribute to increase the formality needed to handle the project, and in the case of parallel applications these factors almost take the project to the third level of formality, out of a five level scale. Whenever possible, the use of techniques to help decrease the weight of these factors will greatly improve the project's development. The effects of software size on error density and programmers productivity are also discussed.

Dealing with simpler pieces of code is much easier than managing chunks of logic and algorithms[McCONNELL,1993]. However, sometimes the modularization ends in very fine grained units that may be difficult for a programmer that didn't participate in the modeling process to understand [PETERS and PEDRYCZ,2000]. But this is mostly related to software engineering aspects of modularization rather than parallelization.

Although both [PETERS and PEDRYCZ,2000] and [McCONNELL,1993] are not based on studies on parallel programming, we can consider that the observations presented by the authors could also affect this area.

1. Will the application handle many different entities and abstract data types?
2. Would it be possible to have a generalization of behavior in order to inherit or compose it along with an hierarchy of inheritance or chains of composition?

3.3 Software Domain

This aspect will determine whether or not application components may be reused in their binary form. This is important because, when reusing, we do not want to copy source code from project

3 In this case software size is measured in SLOC – Source Lines of Code. Although this kind of measure may not reflect reality in some cases, it still can deliver a quantitative understanding of software size.

to project. Besides that, binary code, in most cases, has been tested in other projects and is probably more reliable then recompiled source code.

As explained in [SOMMERVILLE,2001], software domain relates to the overall behavior of the application apart from the specific entities that are being handled. This may lead to a generic model for processing different instances of the problem at hand.

In [PRESSMAN,1997], the author describes the steps needed to adopt domain engineering and argues that, ultimately, the use of this technique will lead to a library of components, which in turn may be used later. In fact, if analysis is employed with domain characterization in mind, it will separate generic features from specificities and this may improve artifact reuse. A large set of artifacts may be reused, but considering the present scope, we can focus specifically on binary component reuse.

If it is possible to isolate the domain specifics into some entities modeling the application, and by not mixing the specifics with the processing part we could achieve a third level of decoupling, namely communication infrastructure layer, algorithm processing layer, and domain specific layer.

As we can see in [SZYPERSKI,2003], the use of domain specific entities must not be considered indiscriminately because this may lead to early compromise with certain design decisions that may not fully satisfy the problem at hand. However, if correctly applied, this approach may greatly improve the software development cycle by providing read-to-use, test-proof binary code that in most cases will have only to be configured for the new problem. It is important to note that sometimes customization may not require component recompilation, but only component specialization through inheritance.

Considering, for instance, the Chemistry domain, if we have some algorithm that processes some molecule representation that could be modeled into a Molecule class, it could be possible to represent different instances of the application by changing the Molecule subclass that is passed to the algorithm processor. This is a very natural programming technique when using objects, but we can argue that at runtime the algorithm processor would need Molecule information that would have to be accessed in an indirect fashion, which again leads us to the decision point of worthiness.

As reported in [MATTHEY et al.,2004], the use of a higher abstraction and flexible design on an object-oriented framework for molecule dynamics helps users to understand existing applications and develop new algorithms focusing on the specificities of this task, regardless of the infrastructure support needed to make a new algorithm run. Although the main goal is not parallelization, the framework presented allows simple master/slave parallelization that is implemented by the use of an interface. When no parallel version of the concrete method is encountered, the sequential version is used. The authors argue that they are working on parametrization of the parallelism capabilities, but we can consider that, since the parallelism infrastructure is decoupled, many optimizations may be applied in an independent fashion, without affecting applications that use this framework.

Similar approaches were used in [JIAO,CAMPBELL and HEATH,2003][GERTZ and WRIGHT,2003] and [NORTON,SZYMANSKI and DECYK,1995]. All the authors considered that the use of object oriented techniques improved software development, and the use of inheritance allowed for greater reuse. Clean interfaces made it easy for modules from multidisciplinary teams to be integrated and helped to increase the abstraction level, allowing faster development of new instances of problems. Specially in [NORTON,SZYMANSKI and DECYK,1995] the authors present a comparison between procedural and object oriented programming paradigms considering parallelization as a main factor concluding that the use of more powerful abstractions greatly improves implementation due to its better support to express complex concepts. Even further, the use of procedural programming, while dealing with large scale problems, may increase complexity beyond the capabilities of this paradigm.

1. Will it be possible to isolate the specifics of the domain being modeled in order to achieve more general algorithm processors that could be reused?

2. Having general solutions will improve the time needed to have a new instance of parallel application up and running?

3.4 Complexity

Software complexity is always a controversial topic and difficult to approach. As presented in [EVANGELIST,1983] we have quantitative approaches that will measure the computational complexity, but this can be misleading since it may point high complexity on easy-to-understand logic and, on the other hand, low complexity for an intricate logic that will demand greater understanding effort. Although the approaches presented in [PRESSMAN,1997] are not considering the specifics of parallel and distributed software development, they may improve the track of software development. Also in [BHANSALI,2005], the author presents an approach to relate control flow with data flow, which seems to be more realistic since it considers the data coupling that may be present. Moreover, if we consider the development of parallel applications, the relationship between data and control flows is crucial, since in most cases this kind of application tends to process large amounts of data. Besides, having an initial understanding of the complexity associated with a certain development may guide the team on better choices.

We can consider complexity, for the sake of clarity, as the amount of hard-to-implement pieces of algorithms that are related to the domain being modeled, thus taking into account here the difficulty associated with a certain algorithm for the programmer to develop. When considering difficulties associated with the communication parts, we could always succeed by resorting to the experience of an infrastructure specialist programmer, but full understanding of the problems related to the application itself will demand a higher degree of work. Ultimately, we could consider that communication patterns – no matter how hard – are in most cases presented and repeated, but the application that is being developed may not rely on any previous model.

If we are dealing with difficult algorithms, it becomes very interesting to apply simplification techniques that could help to divide the problem, for that, both object and composition approaches, by increasing the abstraction level, can deliver gradual simplifications to the problem at hand.

Isolating the hard problem parts can simplify the task by having a domain specialist helping to write a small sequential-like software part and letting the programmer integrate this solution into the more generic application.

1. Is it possible to isolate the hard problem parts using specialized objects or components?
2. Is it possible to have a specialist to help on the development of complex objects or components?

3.5 Maintenance Cycles

When considering software development, it is always a good idea to keep in mind changes and evolutions. It is somehow difficult to predict when these are going to happen, and it is even possible to have the application completely replaced by a new one. But if they do happen, how hard maintenance has to be is a decision that the programmer can make in the beginning.

It is a myth that software won't have to evolve, and having intricate logic and communication code mixed up is always a bad idea for this task in particular. In [SOMMERVILLE,2001], the author shows some information on maintenance costs regarding business applications that indicates that it may be compared to the cost of the system development. However, for real-time embedded systems this cost can reach a factor of being four times higher. Although there was no evidence for parallel software, due to its complexity, we may infer that costs will be similar to real-time applications, as described by the author. A design that facilitates later maintenance should be considered even if the development costs are a little higher. It is also shown that maintenance costs over a poor design grow exponentially.

For professional teams, the changes on team structure are less usual. However for academic level teams, if we consider graduate students, we must keep in mind that a higher degree of team changes occur due to students leaving and arriving. Having this kind of scenario may contribute to degrade the quality of code. Having maintenance in mind, programmers will make the integration of new programmers a much simpler task, improving productivity.

Maintaining code that was developed only with performance in mind may be as hard as writing a new application all from scratch; moreover, it is common not to have the optimization decisions documented which in turn leads to confusion about the code implemented.

1. Is it possible to predict how often it is going to happen?
2. Is the maintenance team aware of implementation issues and decisions?
3. Is the maintenance team composed of the original developers or do they have access to the original developers?

3.6 What are the options ?

Throughout this paper, many options have been presented an independent guidelines, focusing on each possibility alone, for understanding what can be done to achieve large scale parallel applications. Now, these options are put together in an systematic fashion to make it easier for decisions to be made.

The first things we need to rank are the implementation alternatives which are presented below in an ascending degree of abstraction:

1. High performance tools applied to traditional procedural programming – the improvement is achieved only on the performance level with little or no improvement on the programmer productivity.
2. Wrapping of existing procedural programs with components – this alternative may deliver better productivity when the legacy code starts to be used only as black-box components. Development is held in a mixed mode between procedural and object based.
3. Use of CORBA with underlying MPI – greatly improves productivity since it allows object and component modeling, however the application code is still coupled with communication code that is mostly designed in a procedural way.
4. Conservative componentization based on MPI – similar to the previous approach, but may take advantage of the use of components from other domains to improve software development.
5. MPI-like implementation with object technology – breaks the limiting boundaries of the mapping between objects to MPI when communication takes place, but essentially the difference relies on the exchanges of values of complex types in a simplified manner.
6. Componentization with Collective Interfaces – greatly improves design and modeling, separating the concerns of communication in a way that it can be configured and exposed without explicit coding.
7. New Parallel Programming Language – this is the highest level of abstraction possible when the language itself incorporates the concepts necessary to express all the recurring problems that are encountered daily by parallel software developers.

Moving through these levels of abstraction may gradually deliver many benefits both for the teams and software that is developed.

Beyond the levels of abstraction that were presented here, we should also consider the supporting tools that help deliver these benefits. Four items presented may be incorporated on the daily development routine for at least the first five levels of abstractions. They are:

1. The use of a compression mechanism for exchanging messages improving network performance – as presented earlier, this is done by the library in a programmer independent way.

2. Automatic tuning of collective operations based on the analysis of the underlying hardware and communication pattern – this may be achieved in a static or dynamic way, overall performance improvement or at least the same result as the manual development.

3. Use of compiler optimizations – this depends on the availability of the compiler.

4. Use of debugging and testing tools – as presented earlier there are a set of MPI related errors that are recurring and there are tools that are capable of identify these patterns regardless of code execution improving code reliability.

Finally, the remaining guidelines to consider are a little more subjective than the previous ones. There is still little or no data available to be used on comparisons that could guide the choice between features. However, we can base our experiments on the previous results of sequential programming that were studied and documented by software engineering researchers. Our remaining guidelines are:

1. Use separation of concerns for independent evolution of components – be it object or component based, development may be greatly improved by having layers providing services among each other, which ultimately leads to independent improvements that will have global impact.

2. Estimate real performance needs – sometimes code is optimized too early, consuming development effort. Most importantly, we must ensure that, when optimization takes place, it is executed on the real important parts of the code and that it delivers greater impact. Spending too much effort to deliver a small fraction of performance improvement may not be a good idea if the final cost is too high.

3. Estimate real workload – if application's workload is overlooked we may end up with performance issues that will be hardly overcome with simple techniques. It will mostly break all the poorly designed structures to achieve performance improvements that could be planned from beginning.

4. Estimate software complexity – if experts are going to be needed it is better to know exactly which parts will need their intervention.

5. Level of formality that will be employed on the methodology chosen – this is an important consideration that has to be made and agreed upon, once defined, the team must comply with it.

6. Identify the opportunities to isolate domain specific entities and generic algorithm processors – this consideration will contribute for the creation of a library of components for reuse.

4 Conclusions

The use of new paradigms that provide higher levels of abstraction may provide many benefits to parallel programming developers. However the decision of whether or not one should choose to apply these techniques on an application project remains subjective and depends on many factors related to time to delivery, programmer experience and complexity among others.

It is important to note that, although the traditional procedural approach is widely spread, it may incur in many difficulties for the application developer by not providing greater complexity management capabilities. Object and component technologies are being successfully applied to other computer science areas and delivering better results to the management of projects.

More than simply helping with project management, it is possible to deliver better reuse of various parts of parallel applications that tend to be copied. Objects and components may allow the reuse of binary code already tested and ensured to work properly. Moreover, the whole environment could offer parts of communication infrastructure, liberating the programmer from this responsibility. Besides, having the communication capabilities offered by the environment may

offer better quality components produced by the environment developers, which in most cases are more experienced with the issues related to system development [GORLATCH,2004].

Also, the use of objects and components may deliver more generic algorithms that could be applied to a certain set of entities pertaining to some specific domain of applications. This could in turn be a great ally in terms of rapid application development on the field of parallelism.

In [MILLER,2002], the author presents a consideration about performance stating that an order of magnitude slowdown may be acceptable when we consider the benefits that programmers can derive from a higher level of abstraction. However, when facing this type of scenario, one should always consider which type of application is being developed. An order of magnitude slowdown may be acceptable for a *certain* set of applications when considering the improvement on the programming perspective. We can consider that this idea is supported by the authors in [SKILLICORN and TALIA,1998] where it is stated that the ultimate performance is unnecessary, specially if it is achieved by compromising maintainability and at a high development cost.

As we discussed, performance may always be improved by many optimization techniques and so should not be considered alone when deciding which tools to use on the development of parallel applications. The improvement that may result by using more powerful abstractions may be associated with other areas like reuse, development time and maintainability, to name a few.

Together with the analysis of the issues and possibilities associated with the concerns of performance, software size, software domain, complexity and maintenance cycle, we have proposed sets of guidelines for parallel program developers expressed in the form of strategic questions. Later we grouped the related concerns to present an hierarchy of existing alternatives to parallel programming from lower to higher levels of abstraction capabilities. Together with this hierarchy, the classification of technical choices and other more subjective aspects were presented and related in order to provide some initial guidelines for use on future developments in the area.

As a final consideration, we should be aware that maybe the development of a specific branch of the software engineering discipline will be needed to deal with the problems shared by parallel application developers if we are to achieve the same level of productivity that is experienced on the information systems and business applications fields.

References

ANDREWS G.R. **Foundations of multithreaded, parallel, and distributed programming** Addison-Wesley, 1st Ed., 2000.

BAUDE, F., CAROMEL, D., HENRIO,L., MOREL,M. Collective Interfaces for Distributed Components. In: PROCEEDINGS OF THE 7TH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID(CCGRID07), Rio de Janeiro – Brasil, IEEE Computer Society, May 2007. p. 599-610.

BADUEL, L., BAUDE, F., CAROMEL, D. Efficient, flexible, and typed group communications in Java. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, Seattle, Washington, USA, New York:ACM, 2002, p. 28-36.

BADUEL, L., BAUDE, F., CAROMEL, D. Object-oriented SPMD. In: PROCEEDINGS OF THE 5TH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID(CCGRID05), 2., Washington, DC, USA, IEEE Computer Society, May 2005, p. 599-610.

BAKER, M., CARPENTER,B., FOX,G., KO,S. H., LIM,S. mpi-Java: An Object-Oriented Java interface to MPI. In: INTERNATIONAL WORKSHOP ON JAVA FOR PARALLEL AND DISTRIBUTED COMPUTING, San Juan, Puerto Rico, 1999.

BENTLEY, J. **Programming Pearls.** Addison-Wesley, 2000.

BHANSALI, P.V. Complexity measurement of data and control flow. ACM SIGSOFT Software Engineering Notes, v.30, I.1, ACM Press, USA, Jan 2005.

BIGOT, J., PEREZ, C. Enabling collective communications between components. In: PROCEEDINGS OF THE 2007 SYMPOSIUM ON COMPONENT AND FRAMEWORK TECHNOLOGY IN HIGH-PERFORMANCE AND SCIENTIFIC COMPUTING, ACM Press, Canada, 2007, p. 121-130.

CARRIERO, N., GELERNTER, D. How to write parallel programs: a guide to the perplexed. ACM Computing Surveys (CSUR), New York, v. 21, i. 3, p. 323-357, Sep 1989.

CARVALHO, F.H., LINS, R.D. The # model: separation of concerns for reconciling modularity, abstraction and efficiency in distributed parallel programming, In: PROCEEDINGS OF THE 2005 ACM SYMPOSIUM ON APPLIED COMPUTING, ACM Press, USA, 2005, p. 1357-1364.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., PRAUN, C. Von, SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In: PROCEEDINGS OF THE 20TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, ACM Press, USA, 2005, p. 519-538.

DANIS, C. Forms of collaboration in high performance computing: exploring implications for learning. In: PROCEEDINGS OF THE 2006 20TH ANNIVERSARY CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK, ACM Press, Canada, 2006, p. 501-504.

EVANGELIST, W.M. Relationships among computational, software, and intuitive complexity. ACM SIGPLAN Notices archive, v.18, I.12, New York, USA, p. 57-59, Dec 1983.

FARAJ, A., YUAN, X. Automatic generation and tuning of MPI collective communication routines. In: PROCEEDINGS OF THE 19TH ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ACM Press, USA, 2005, p. 393-402.

FARAJ, A., YUAN, X., LOWENTHAL, D. STAR-MPI: self tuned adaptive routines for MPI collective operations. In: PROCEEDINGS OF THE 20TH ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ACM Press, USA, 2006, p. 199-208.

FOSTER, I. **Designing and Building Parallel Programs.** Addison-Wesley, USA, 1995. Available at http://www-unix.mcs.anl.gov/dbpp/text/book.html. Accessed in June, 29, 2009

GERTZ, E.M., WRIGHT,S.J. Object-oriented software for quadratic programming. ACM Transactions on Mathematical Software (TOMS), v.29, I.1, ACM Press, USA, p. 58-81, Mar 2003.

GORLATCH S. Send-receive considered harmful: Myths and realities of message passing. ACM Transactions on Programming Languages and Systems (TOPLAS), New York , v. 26 , i.1, p. 47-56, January 2004.

GROPP, W., LUSKL, E., SKJELLUM, A. **Using MPI: portable parallel programming with message passing interface**, MIT Press, 2nd Ed.,1999.

JIAO, X., CAMPBELL, M.T., HEATH,M.T. Roccom: an object-oriented, data-centric software integration framework for multiphysics simulations. In: PROCEEDINGS OF THE 17TH ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ACM Press, USA, 2003, p. 358-368.

KARWANDE, A., YUAN, X., LOWENTHAL, D.K. CC–MPI: a compiled communication capable MPI prototype for ethernet switched clusters. In: PROCEEDINGS OF THE NINTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, ACM Press, USA, 2003, p. 95-106.

KE, J., BURTSCHER, M., SPEIGHT, E. Runtime Compression of MPI Messages to Improve the Performance and Scalability of Parallel Applications. In: PROCEEDINGS OF THE 2004 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, IEEE Computer Society, USA, 2004.

KERNIGHAN B.W., PIKE, R. **The Practice of Programming**, Addison-Wesley, 1st Ed., 1999.

MATTHEY, T., CICKOVSKI, T., HAMPTON, S., KO, A., MA, Q., NYERGES, M., RAEDER, T., SLABACH, T., IZAGUIRRE, J.A. ProtoMol, an object-oriented framework for prototyping novel algorithms for molecular dynamics. ACM Transactions on Mathematical Software (TOMS), v.30, I.3, ACM Press, USA, p. 237-265, Sep 2004.

McCONNELL S. **Code Complete: A Practical Handbook of Software Construction**, Microsoft Press, 1st Ed., 1993.

MILLER, P. Parallel, Distributed Scripting with Python. Linux Clusters: The HPC Revolution, October, 2002, Accessible at http://www.democritos.it/activities/IT-MC/cluster_revolution_2002/PDF/10-Miller_P.pdf Accessed in June, 29, 2009

MOHAMED, N., AL-JAROODI, J., JIANG, H., SWANSON, D. JOPI: a Java object-passing interface. In: PROCEEDINGS OF THE 2002 JOINT ACM-ISCOPE CONFERENCE ON JAVA GRANDE, ACM Press, USA, 2002, p. 37-45.

NORTON, C.D., SZYMANSKI, B.K., DECYK,V.K. Object-oriented parallel computation for plasma simulation. Communications of the ACM, v.38, I.10, ACM Press, USA, p. 88-100, Oct 1995.

ONG, E. MPI Ruby: Scripting in a Parallel Environment. Computing in Science and Engineering, v.4, I.4, New Jersey, USA, p. 78-82, Jul 2002.

PARLAVANTZAS, N., GETOV, V., MOREL, M., BAUDE, F., HUET, F., CAROMEL, D. Componentising a scientific application for the grid. In: Technical Report TR-0031. Institute on Grid Systems, Tools and Environments, CoreGRID, 2006.

PRESSMAN R. **Software Engineering: A Practitioner's Approach**, McGraw Hill Book, 4th Ed., 1997.

PETERS J.F., PEDRYCZ,W. **Software Engineering: An Engineering Approach**, Wiley, 1st Ed., 2000.

RENÉ, C., PRIOL, T. MPI Code Encapsulating using Parallel CORBA Object. In: PROCEEDINGS OF THE 8TH IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, IEEE Computer Society, USA, 1999.

SKILLICORN, D.B., TALIA, D. Models and languages for parallel computation. ACM Computing Surveys, v.30, I.2, ACM Press, USA, p. 123-169, Jun 1998.

SOMMERVILLE I. **Software Engineering**, Addison-Wesley, 6th Ed.,2001.

SUGUMARAN, V., TANNIRU, M., STOREY, V. C. Identifying software components from process requirements using domain model and object libraries. In: PROCEEDING OF THE 20TH INTERNATIONAL CONFERENCE ON INFORMATION SYSTEMS, North Carolina, United States, ACM Press ,1999, p. 65-81.

SZYPERSKI, C. Component Technology - What, Where, and How? In: 25TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE'03), 2003, p.684.

TAN, K., SZAFRON, D., SCHAEFFER, J., ANVI, J., MACDONALD, S. Using generative design patterns to generate parallel code for a distributed memory environment. In: PROCEEDINGS OF THE NINTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, ACM Press, USA, 2003, p. 203-215.

VADHIYAR, S. S., FAGG, G. E., DONGARRA, J. Automatically tuned collective communications. In: PROCEEDINGS OF THE 2000 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, IEEE Computer Society, USA, 2000.

YUAN, X., DUAN,S., LIU, Z. Exploring robust component-based software. In: PROCEEDINGS OF THE 2006 INTERNATIONAL WORKSHOP ON SOFTWARE QUALITY, Shanghai, China, ACM Press, 2006, p. 75-80.