FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'99)

Federico Bassetti, Kei Davis*, Bernd Mohr (Eds.)*

FZJ-ZAM-IB-9906

Juni 1999

(letzte Änderung: 07.06.99)

(*) Scientific Computing Group, CIC-19 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, U.S.A.

Part of European Conference on Object-Oriented Programming (ECOOP'99)

Preface

This report contains the Proceedings of the Workshop on Parallel / high-performance Object-Oriented Scientific Computing (POOSC'99) at the European Conference on Object-Oriented Programming (ECOOP'99) which is held in Lisabon, Portugal on June 15, 1999. The workshop is a joint organization by the Special Interest Group on Object Oriented Technologies of the Esprit Working Group EuroTools and Los Alamos National Laboratory.

While object-oriented programming is being embraced in industry, particularly in the form of C++ and Java, its acceptance by the parallel / high-performance scientific programming community is tentative at best. In this latter domain performance is invariably of paramount importance, where even the transition from FORTRAN 77 to C is incomplete, primarily because of performance loss. On the other hand, three factors together practically dictate the use of language features that provide better paradigms for abstraction: increasingly complex numerical algorithms, application requirements, and hardware (e.g. deep memory hierarchies, numbers of processors, communication and I/O).

In spite of considerable skepticism in the community, various small groups are developing significant parallel scientific applications and software frameworks in C++ and FORTRAN 90; others are investigating the use of Java. This workshop seeks to bring together practitioners and researchers in this emerging field to 'compare notes' on their work – describe existing, developing, or proposed software; tried and proposed programming languages and techniques; performance issues and their realized or proposed resolution; and discuss points of concern for progress and acceptance of object-oriented scientific computing.

By request of the publisher, only a report of the workshop will appear in the ECOOP'99 Workshop Reader. Because of this, we decided to publish a collection of selected papers ourselves. The papers included reflect the multidisciplinary character and broad spectrum of the field. We thank all contributors for their contribution and cooporation. The organizers also want to thank Ana Maria Moreira for the work and help organizing this workshop. Finally we thank all the attendees and contributors who made this workshop a high quality event!

June 1999

Kei Davis
Federico Bassetti
Bernd Mohr

Organization

The POOSC'99 workshop is part of ECOOP'99, the 13th European Conference on Object-Oriented Programming which is organized by the Department of Computer Science of the University of Lisbon, under the auspices of AITO (Association Internationale pour les Technologies Objets).

Workshop Organizers

Kei Davis
Federico Bassetti
Scientific Computing Group, CIC-19 MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545, U.S.A.
{kei,fede}@lanl.gv

Bernd Mohr
Research Center Juelich
John von Neumann Institute for Computing (NIC)
Central Institute for Applied Mathematics (ZAM)
52425 Juelich, Germany
b.mohr@fz-juelich.de

Sponsoring Institutions

This workshop is supported by the Esprit Working Group WG 27141 EuroTools.

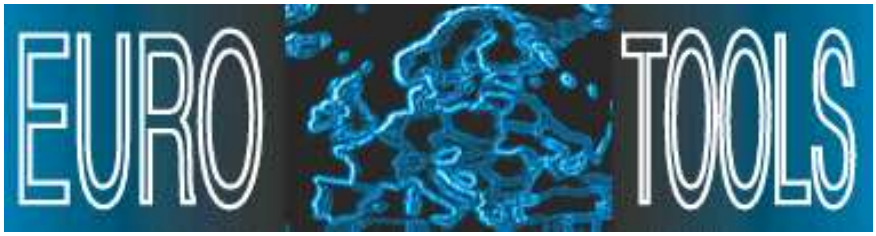

Table of Contents

Object Oriented Concepts for Parallel Smoothed Particle Hydrodynamics Simulations

Stefan Hüttemann[1] Michael Hipp[1] Marcus Ritt[1] Wolfgang Rosenstiel[1]

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Arbeitsbereich für Technische Informatik
Sand 13, 72076 Tübingen
e-mail: {hippm,hutteman,ritt,rosen}@informatik.uni-tuebingen.de

Abstract. In this paper we present our object oriented concepts for parallel smoothed particle hydrodynamics simulations based on a 3 year work experience in a government funded project with computer scientists, physicists and mathematicians.[1]

In this project we support physicists to parallelize their simulation methods and to run these programs on supercomputers like the NEC SX-4 and Cray T3E installations at HLRS Stuttgart (www.hlrs.de).

First we introduce our portable parallel (non object oriented) environment DTS. Benchmarks of simulations we parallelized are shown, to demonstrate the efficiency of our environment.

Based on these experiences we discuss our concepts developed so far, and future ideas for object oriented parallel SPH simulations at two different layers. An object oriented message passing library with load-balancing mechanisms for our simulations at the lower level, and an object oriented parallel application library for our physical simulations on the upper level.

1 Motivation

In a collaborate work of physicists, mathematicians and computer scientists, we simulate astrophysical systems. In this paper we present our object oriented concepts based on our experiences made in a government-funded project[1] in the last three years. Smoothed Particle Hydrodynamics (SPH) is the method used by the astrophysicists to solve a Navier-Stokes equation (see [3], [6]). SPH became widely popular in the last years. SPH is now also used as an alternative for grid based CFD simulations (e.g. in automobile industry).

The astrophysical problems are open boundary problems of viscous compressible fluids. SPH uses particles that move with the fluid instead of grid

[1] SFB 382: "Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern" (Methods and algorithms to simulate physical processes on supercomputers)

points as in other CFD simulation methods. This makes SPH much more difficult to parallelize than grid based CFD methods.

1.1 Portable Environment

Programming with threads showed to be well-known and simple enough to serve as a basis for a portable parallel programming environment. The first approach, named "Distributed threads system" (DTS) [5] generalized the notion of threads for distributed memory machines. A compiler was written to simplify the task of identifying and creating parallel threads.

A major drawback of this system was its pure functional programming model: the communication between threads running on different nodes was not well supported. We are investigating, whether a combination of (global) threads and distributed shared memory, i.e. a logical consistent memory for machines with physically distributed memory, is suitable for our applications.

Currently we are working at the task of combining the basic ideas of this system, namely distributed threads and shared memory, with object-oriented concepts. This system, written in C++, is described in section 3.

For Cray T3E we used another approach. We provide a high level application interface optimized for SPH-like programs. The library has a simple to use interface and hides near all of the parallelization and explicit communication. A programmer only has to give some hints to optimize the communication and load balancing. The library itself is based on the native SHMEM message passing for Cray T3E or alternatively on MPI.

1.2 The need for Object-Oriented techniques

There were reasons to redesign our simulation environment using object-oriented techniques:

1. using message objects on the lowest level to communicate between concurrent program units on distributed memory computers seems to be most natural and easy to use for our simulation methods.
2. the growing complexity of our simulation programs requires structural elements in the programming paradigm not offered by e.g. FORTRAN or C. Also using an object-oriented approach to describe the problem is closer to the physical model used.
3. to exploit the usual features promised by object-oriented programming (reusability etc.) our project partners tried programming in C++; which resulted in anything but reusable, modular software. It showed, that just by switching to C++ physicists do not gain much, and fall back to FORTRAN-style programming.

   Our goal is to provide a well documented library of reusable and extensible solutions for astrophysical simulation methods. This also should give a guideline on how to use object-oriented techniques for our simulation methods (e.g. SPH).

2 Parallelizing SPH

We could gain a lot of experience with two different types of parallelization of the SPH code for shared memory machines based on DTS and for machines with distributed memory for which we developed a portable procedural library.

Both implementations can compete with other SPH codes for parallel machines such as the codes for Cray T3D, CM-5 or Intel Paragon [4,2].

2.1 Shared Memory Machines

The SPH code was implemented on the NEC SX-4 using DTS. The usage of DTS allows to run the same SPH code both on the NEC SX-4 and on other machines[1] without modifications. The parallel SPH code was used for benchmarking, using different numbers of CPUs. The results prove the high quality of the parallelization features of the NEC SX-4.

The right side of fig. 1 shows the parallel efficiency of the parallel SPH code on the NEC SX-4. For 10 000 particles the parallel efficiency decreases from 90% on two CPUs to 60% on 20 CPUs. For 100 000 particles the parallel efficiency is more than 90% for all 20 CPUs.

Further runtime improvement can be reached by vectorization of the code. In the left side of fig. 1 we present a comparison of the runtimes of SPH simulations of the same test problem with different codes. Together, the measurements in fig. 1 show, that already using as less as 2 CPUs on the NEC SX-4 a runtime improvement compared to the optimal sequential SPH code can be achieved. As the efficiency is excellent for 20 processors, very good runtime improvements can be expected using more CPUs.

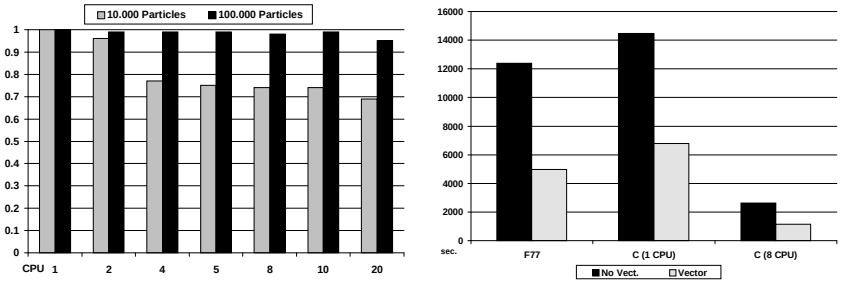

Fig. 1. *Left:* parallel efficiency on NEC SX-4. *Right:* Vectorization and F77 solution of a SPH simulation with 10 000 particles for optimized F77 sequential code, parallel SPH on 1 CPU and parallel SPH on 8 CPUs.

[1] e.g. SGI Onyx2, HP V-Class

2.2 Distributed Memory Architectures

Besides the DTS SPH code for shared memory machines, there are a few implementations for SPH on parallel machines such as the PTreeSPH [1]. The PTreeSPH code is based on MPI and, therefore, is portable to nearly every platform. We decided to go another way, because we see the need for a more efficient implementation on some architectures. We developed an abstraction layer optimized for SPH like problems with two different low level implementations for the communication.

The slower portable implementation is based on MPI. The other implementation is based on the Cray SHMEM message passing library, which provides functions oriented at the Cray T3E hardware capabilities and therefor gains a better performance.

Having two different implementations allowed us to test the flexibility of our SPH abstraction layer. We first wrote the SHMEM implementation and specified the interface on which we put our physical code. It showed that the layer was flexible enough to later add a MPI implementation without changing the interface or the physical application.

Parallelization An essential idea for parallelization was to use two different domain decompositions depending on the type of computation:

1. All computations without neighbor interaction are done on an equally sized subset on every node. The subset is selected by splitting the particle field into n parts for n nodes. A node also operates as a *relay* node for its subset. Information about a specific particle can always be found on its *relay* node.
2. For computations with neighbor interaction all particles are sorted according to their positions into a grid with equally sized cells. These cells are assigned to nodes in a way that every node holds the same number of particles.

The load balancing is good in both cases, because the computation takes about the same time for every particle. For the case of very unbalanced computations we have the option to do *load stealing* between nodes to optimize the load balancing.

This approach reduces communication overhead and memory consumption because the particle information resides on one node as long as possible. Also, the computation of this domain decomposition is fast enough to be done *on the fly*. This is important, because the particle positions change after every integration step.

Native SHMEM communication vs. MPI We measured the performance of the MPI code on the Cray T3E in Stuttgart and on the IBM SP system in Karlsruhe. On the SP we used 128 P2SC thin nodes with 120MHz. The tests showed, that the MPI implementation of the Cray T3E is worse

compared to the native SHMEM library (see fig. 2). Our tests show that Cray could easily improve the MPI performance by making better wrappers around existing SHMEM calls. For some communication parts, such as gather operations of large arrays, the throughput decreased from about 300MB/s using SHMEM to 120 MB/s using MPI. On the SP we achieved the expected performance of around 50MB/s. The whole code didn't perform this good on the SP system, because the implementation is optimized for Cray T3E and depends heavily on good communication bandwidth and latency in order to scale beyond 64 Processors.

Results It proved to be beneficial to use an abstraction layer, which allows the exchange of low level parts without changing the application to obtain the best performance on a given hardware platform. In the previous projects this was a procedural abstraction layer. For our current developments we choose an object oriented programming model for this abstraction. The crucial point for the parallelization is a smart domain decomposition optimized for both, the machine and the problem. An object oriented modeling of domain decompositions, which we want to describe in a later chapter is therefor one of the requirements for our parallel environment.

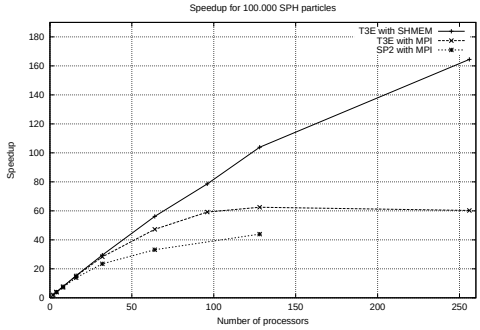

Fig. 2. Speedup of the SPH Simulation on Cray T3E with SHMEM and MPI and on IBM SP with MPI for a mid-size problem with 100 000 SPH particles. For larger node numbers the curves are dominated by the non parallelized communication parts such as gather/broadcast operations between all nodes. Please note the effect of the limitations of the MPI implementation on the Cray T3E beyond 128 nodes.

3 An object-oriented parallel runtime system

Based on the experiences with existing object-oriented parallel systems and our own object-oriented codes, we now describe our concepts for a object-oriented parallel runtime system.

There were numerous reasons, which motivated the redesign of these layers, despite of the existence of object-oriented message passing libraries like MPI++ or MPC++ [7]. The most important were the lack of thread-safe

implementations and the missing integration of modern C++ concepts like templates and the support for the standard template library.

To support object-orientation for parallel programming, we extended our model of parallel computing with threads on machines with distributed memory to C++ objects. In this model, an object – extended for architectures with distributed memory – is the basic entity of data communication. Objects can be migrated between different address spaces and replicated to improve performance. Migration and replication can be done explicitly by the user. For specific applications-domains, for example particle codes, we intend to provide tailored load-balancing components which free the user from explicitly specifying the data distribution. The methods for guaranteeing consistency are based on the well-known consistency protocols from distributed shared memory. In addition to this, objects support asynchronous remote method invocations. This corresponds to the asynchronous remote procedure call in our former approach, that is, a thread fork extended for machines with distributed memory. Based on these facilities we plan to integrate some library solutions for a couple of common problems, for example automatic parallelization and load-balancing for independent data-parallel problems. These libraries will be application-independent (in difference to the higher-level libraries described later, which support specific physical problem domains like particle simulations).

To realize this model, we started implementing a basic layer for object-oriented message-passing. This layer can be used independently from the higher-level layers. To keep it portable, it is designed to be easily implemented on different low-level communication primitives. One implementation is on top of MPI to support a broad range of parallel architectures. There also exists an UDP-based version for test runs in a local environments without MPI support. Currently we are porting the library to the Cray T3E to run performance tests. Due to the bad MPI performance on the Cray T3E (see fig. 2, we will also implement a native Cray SHMEM based version for production runs on this platform.

To simplify the migration from procedural codes written in MPI, the functions and methods are very similar to the MPI calls as far as suitable for the object-oriented interface. The main focus lied on extending the MPI functionality to objects without losing type-safety and the full integrations of the STL, i.e. transferring STL containers as well as using iterators for send and receive calls. To support the higher-level layers the library had to be programmed thread-safe.

The user interface for the object-oriented message-passing is straightforward with Communicator objects, send- and receive-methods. Composite objects like STL containers, arrays or user objects are decomposed into basic types by an overload resolution/traits technique [9]. Therefore, the user has not to deal with the unattractive concept of MPI data types, without loosing type-safety. To send and receive objects, the user has to provide serialize and deserialize methods, specifying how an object can be broken in components.

To minimize the communication overhead and prevent writing unnecessary serializer methods, objects with a trivial copy constructor can be handled directly by the library. We are also working an a code preprocessor which will generate the serializer methods for most objects automatically. Furthermore, note that the techniques used for sending objects and other C++ data types over the net, can be used without modification to implement persistent objects and application-level check-pointing.

A message-passing based version of an object-oriented SPH code will be our first test application. Based on these experiments, we plan to implement the higher-level layers by the end of this year. A portable object-oriented thread library will be integrated in the near future.

4 Towards object-oriented parallel SPH

4.1 Design Patterns

We cannot ignore the demand for programming in C or FORTRAN. To provide simply an implementation in C++ will not be accepted by our project partners. We had to find a way to write down our solutions in a "Meta-Language". Using Design Patterns serves this purpose best. We have an easy to understand way to document our solutions that is not bound to any programming language. Writing the design patterns in UML, we can use tools to implement the documented Design Patterns in e.g. C++ (almost) automatically.

As a first step towards an object-oriented SPH program, we used an easy to parallelize Monte Carlo simulation of the pulsar HER-X1. Looking at the problem as a programmer the Monte Carlo simulation and the SPH simulation are similar, because they are both particle simulation methods (in the case of the Monte Carlo simulation the particles are photons).

In the following we want to give an overview over the design patterns we used. The names for the patterns are taken from the Design Pattern book by E. Gamma (see [8]), but our patterns might differ from those in the book (we still need to give names to our patterns).

Composite Pattern for Domain-Decomposition Domain decomposition is the method used to parallelize particle simulations like SPH. The simulation area of all particles is decomposed into separate simulation-domains. The domain consists of particle lists that are being used to evaluate the equations. Since the particles interact, information must flow between the domains.

The main problem with the domains is the communication between the domains, and how to update the particle lists in each domain during the simulation.

To describe the solution for this problem, we used a pattern similar to the composite design pattern. The general behavior common to all simulation-domains, the ability to communicate with other domains, is defined in the

abstract root-class *SimulationArea* (see fig. 3). A concrete simulation area will inherit the basic communication methods from this parent class, and change the implementation to its own needs.

Using this pattern, you can write simulation programs with compatible simulation sub-domains without the need to rewrite the communication between the different simulation-domains. To parallelize a simulation build us-

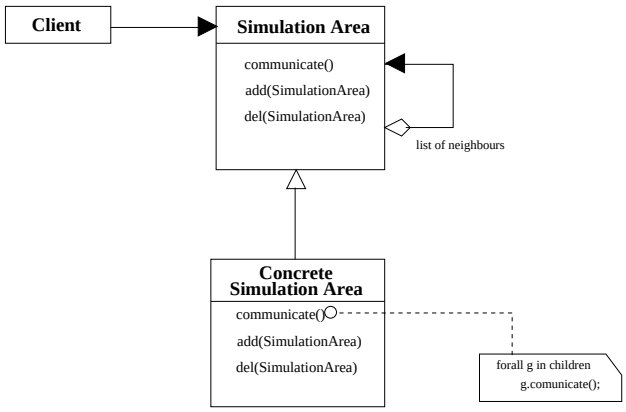

Fig. 3. Design Pattern used for Domain-Decomposition

ing the Domain-Decomposition, we make the *SimulationArea* root-class inherit from a class similar to a *Thread*-class, so that all simulation-area objects become *active* objects. (Active in the sense, that these objects run concurrently).

Iterator Pattern to step through neighbor-lists The iterator pattern is used to step through dynamic lists of neighbor simulation areas. This pattern is also available in C++ STL.

Strategy Pattern to select a numerical algorithm The core functionality of a simulation of physical processes always are some numerical algorithms. The selection of algorithms is stored in libraries for procedural languages like C or FORTRAN. To keep the advantage of a fine-grained selection of different algorithms we choose a strategy pattern for our numerical algorithms.

A strategy always works on a specific context. As an example we use the coordinate transformation of a vector (see fig. 4). Here the vector is the context of the strategy, and the different transformations are the concrete strategies, that have to be implemented for a simulation. The basic functionality is defined in the root-class for the coordinate transformation strategy. In this example the root-class contains the methods `toGeneric` and `toSpecial`. These methods transform the coordinates of a vector between a special coordinate system and a (pre-defined) generic coordinate system.

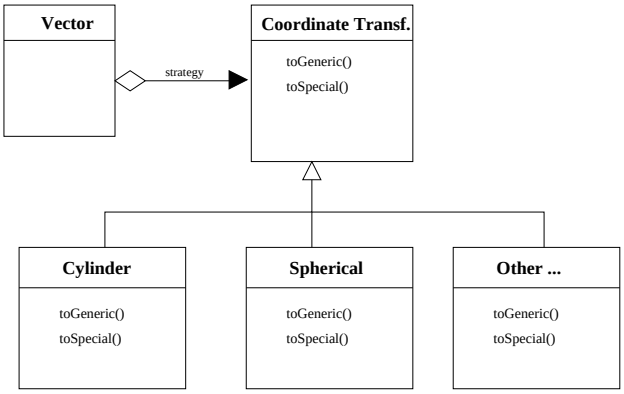

Fig. 4. Strategy Design Pattern for Coordinate Transformations

Facade Pattern to handle input parameters To handle user input parameters independent of the objects used in the simulation, we use the facade pattern. A facade object collects all input, and marshals the parameters to the correct object for this simulation.

Factory Pattern to create the particles A *Factory* pattern can be used to create the particles for the simulation. Communicating a single particle will not be efficient, therefore container classes for particles are necessary. Particles created using the *ParticleFactory* can now be collected, and put into container classes (see fig. 5).

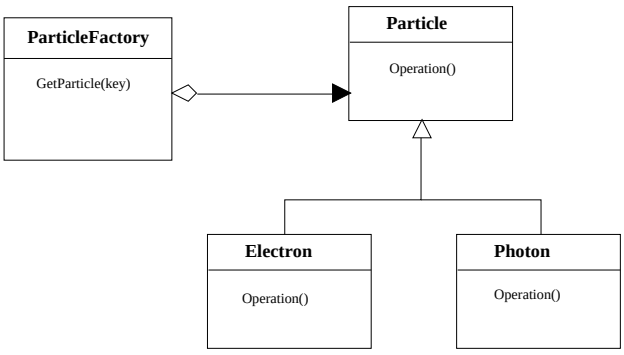

Fig. 5. Factory pattern for particles

Documentation Documenting the Design Patterns in a modern, easy to read way was achieved by using multi-frame HTML documents. Solutions that

are fun to read find generally better acceptance, even if there is no direct implementation in the favorite language of the programmer, e.g. FORTRAN. Also documenting the simplicity of our ready to use solutions motivates more physicists to take a look at a new programming paradigm (some even take a second look).

Prototyping in JAVA We also tried using JAVA for prototyping. Implementing our design patterns written in UML is fast to do in JAVA. The JAVA prototypes cannot be used for the real problem (in our case because there are no JAVA environments on Cray and NEC computers). The prototype is really just a prototype.

References

1. Davé, R., Dubinski, J., Hernquist, L.: *Parallel TreeSPH.* New Astronomy volume **2** number **3** (1997) 277–297
2. Dubinski, J.: *A Parallel Tree Code.* Board of Studies in Astronomy and Astrophysics, University of California, Santa Cruz (1994)
3. Lucy, Leon B.: *A Numerical Approach to Testing the Fission Hypothesis.* Astron. J volume **82** (1977) 1013–1924
4. Warren, S. Michael, Salmon, K. John: *A portable parallel particle program.* Comp. Phys. Comm. volume **87** (1995) 266–290
5. Bubeck, T., Hipp, M., Hüttemann, S., Kunze, S., Ritt, M., Rosenstiel, W., Ruder, H., Speith, R.: *Parallel SPH on Cray T3E and NECSX-4 using DTS* High Performance Computing in Science and Engineering '98, Springer (1999) 396–410
6. Gingold, R. A., Monaghan, J. J.: *Smoothed particle hydrodynamics: theory and application to non-spherical stars* Mon. Not. R. astr. Soc. volume **181** (1977) 375–389
7. Wilson, Gregory V.,Lu, Paul: *Parallel Programming using C++* The MIT Press, Cambridge (1996)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: elements of reusable object-oriented software* Addison-Wesley (1995)
9. *Programming languages – C++. International Standard 14882.* ISO/IEC (1998)

Using Collections to Structure Parallelism and Distribution

Pascale Launay and Jean-Louis Pazat

IRISA, Campus de Beaulieu, F35042 RENNES Cedex
Pascale.Launay@irisa.fr, Jean-Louis.Pazat@irisa.fr
http://www.irisa.fr/CAPS/PROJECTS/Do

Abstract. The aim of the *Do!* project is to ease the task of programming scientific parallel and distributed applications using Java. We use collections to provide an unified way to structure parallelism and distribution. We have defined a parallel framework, based on collections, to write parallel applications in a centralized fashion (as if all objects were on one single host) and we provide "distributed" collections for distributed execution of these applications (objects are mapped on distinct hosts). We have developed both static collections (arrays) and dynamic collections (lists) for this purpose. This framework has been targeted to the Java language and runs on any standard Java environment.

1 Introduction

Two main approaches are used to introduce parallelism and distribution in object-oriented scientific applications: task parallelism [1,7] and data parallelism [4,6]. Our project integrates both approaches:

Data parallelism has been integrated in an object oriented language through the use of collections: collections are large object aggregates, responsible for the storage and the accesses to their elements. The data parallel model is well suited to express global operations on large data aggregates. This model has been widely used in many scientific applications, but it does not allow a convenient description of task parallelism which also exists in those applications. The collection libraries are usually restricted to static collections (arrays). We have used both static and dynamic collections.

Task parallelism has been introduced through the use of active objects. Using collections to store active objects allows us to structure parallelism and to provide global distribution specifications.

In section 2, we show how structured parallel constructs are introduced through the use of collections. We present the use of distributed collections to run parallel applications in distributed environnements in section 3. Finally, we conclude in section 4 with the performance concerns, the current status of our project, and the extensions we are planning to develop.

2 Collections and parallelism

We *introduce* parallelism through active objects: an active object has its own sequential activity and private attributes. A parameter can be passed to our active objects that can be used for communications and synchronizations with other objects. Active objects are implemented in our framework using Java THREADS.

We introduce *structured parallel constructs* through collections. We express global operations on collections through *operators* and we have defined a parallel framework, represented by a PAR class (or a descendant), implementing a parallel model (parallel construct).

2.1 Global operations on collections

Iterators and operators We use the *operator design pattern* [5] (figure 1) to express global operations on collections. Accesses and global operations over collection elements are separate entities: *iterators* and *operators*. From a collection, an *iterator* provides a sequence of elements to the *operator*, that processes elements. An operator accesses elements independently from the collection and can use iterators to traverse collections in various ways. There are several kinds of operators, depending on:

- how many sequences they need as entry. Default operators process elements from one sequence; CROSS operators process elements from two sequences, applying operations to elements from one sequence with the corresponding elements from the second one;
- if they give a result or not. The result may be a single element (REDUCE operator, for example) or a sequence of elements (FILTER operator, for example). Operators giving a sequence of elements can be viewed as iterators, and composed with other operators.

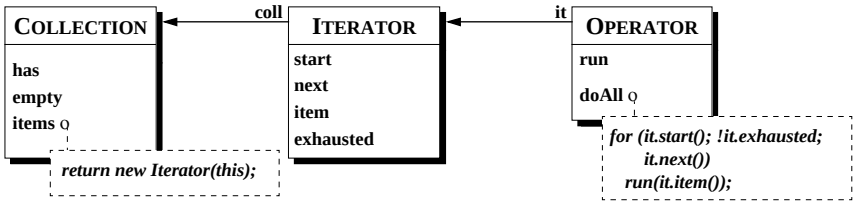

Fig. 1. The operator design pattern

The operator design pattern has been designed in order to model computations over large data structures. It provides us with a modular way to express

parallelism and distribution. We use collections to store active objects (Java THREADS) and we define operations over active objects (activation, synchronization) through the operator design pattern.

Robust iterators and reactive operators In a concurrent environment, it is important to ensure that iterators support insertions and removals in collections during a traversal; such iterators are called *robust iterators* [2]: *"a robust iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the collection"*. By using iterators in the operator design pattern, robustness can be managed both by the iterator and by the operator: in some cases, the iterator can react properly towards an insertion or a removal in the collection, but in other cases the operator has more information to manage the modification correctly.

Moreover, if the operation duration is not limited to the collection traversal, it may be important to react to a modification in the collection even after the end of the traversal. For example, an operator whose purpose is to display a graphical representation of a collection should be able to react properly to a modification in the collection during the whole collection visualization, and modify properly the representation being displayed. We call such operators *reactive operators*.

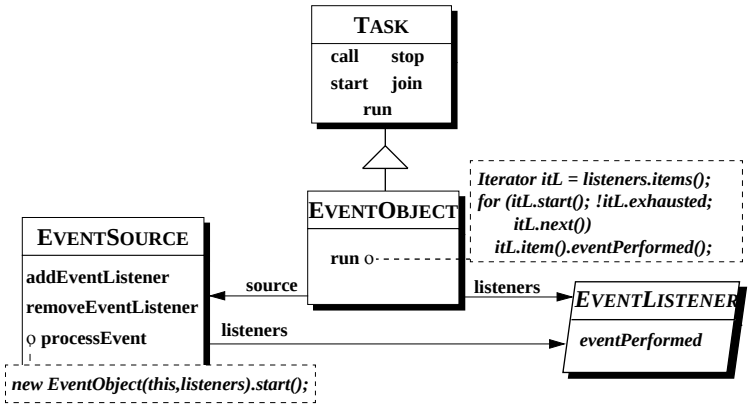

Fig. 2. The event pattern

We have implemented a general solution to manage robustness in a concurrent environment as well as reactivity for operators, relying on an asynchronous event pattern, based upon the *observer design pattern* [2] (figure 2). The event pattern's main components are the source (EVENTSOURCE) that generates events, and the listener (EVENTLISTENER) that reacts to events.

Collections and iterators are sources, iterators and operators are listeners (figure 3): when a modification occurs in a collection, an event is generated

and thrown to iterators operating on that collection. An iterator catching such an event can either manage it properly or throw it to the operator using that iterator: if the element has already been traversed, the iterator throws the event to the operator that takes it into account. As an example, if we define an operator to compute the sum of integers stored in a collection, and if modifications in the collection are allowed during the computation, it is important that the operator takes modifications in the collection into account: if an integer that has already been computed is removed from the collection, the operator has to subtract the integer's value before returning.

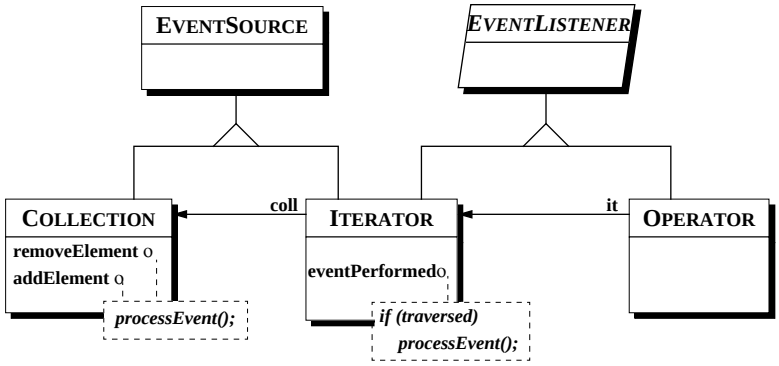

Fig. 3. Robust iterators and reactive operators

2.2 Collections to structure parallelism

Active objects Active objects are objects having their own sequential activity: when using active objects, several control flows run concurrently through different objects, leading to *inter-object* concurrency; when active objects access other objects of the program, they can communicate (through shared objects), leading to *intra-object* concurrency when concurrent control flows run into a single shared object. We have implemented active objects using Java THREADS. Active objects are represented by the TASK class (figure 2); they are activated synchronously or asynchronously through method invocation (`call` or `start`). It is possible to stop their activity, or wait for their termination (`stop` or `join`). Active objects communicate through shared objects, passed as parameters at object activation. The shared object's type is the generic type OBJECT, so the synchronization mechanisms use to manage shared accesses are those provided by the Java language ([3], chapter 17)

Parallel constructs Parallelism is encapsulated in *task collections*, that store active objects. A first traversal of a task collection asynchronously activates each

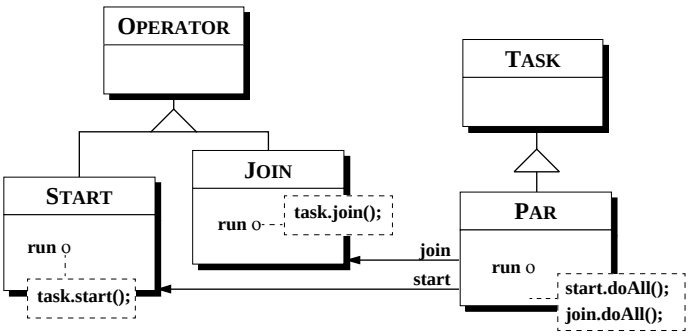

Fig. 4. PAR construct

element leading to intra-collection parallelism. A second traversal is processed to make a synchronization for tasks termination. We have defined a parallel construct, the PAR class, that relies on two specific operators, the START class (to activate tasks) and the JOIN class (to synchronize with task terminations). From a task collection (storing active objects), the parallel construct implements the parallel activation of tasks and the synchronization with task terminations, using the START and JOIN operators (figure 4). Nested parallelism is introduced by the fact that a PAR object is an active object, that can be stored in a task collection and activated in parallel with otheractive objects.

Using different types of collections and iterators, we implement different models of parallelism as parallel constructs represented by PAR subclasses. Figure 5 shows different models of parallelism: the PAR class defines independent parallel task, while the SHAREDPAR subclass implements parallel tasks accessing a single shared object. The DATAPAR subclass implements copies of a single task accessing elements of a *data collection* (coarse grain data parallelism).

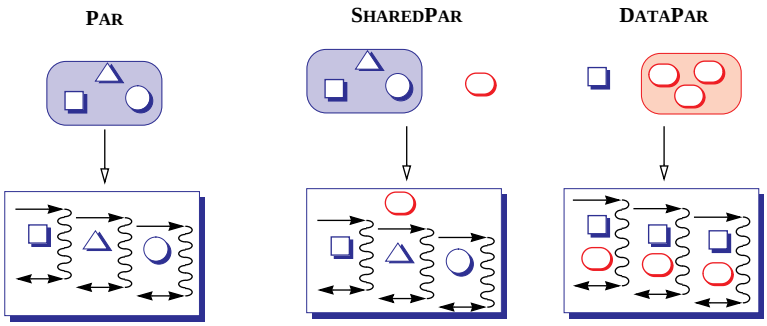

Fig. 5. Different models of parallelism

3 Collections and distribution

We *introduce* distribution through the mapping of objects on distinct hosts (JVMs). We process remote creations to map objects on hosts using runtime classes, and remote method invocations to implement communications between remote objects using the Java RMI. Because using Java RMI requires important modifications in the code of classes, we provide seamless location of objects through code transformations of classes (*accessible* classes).

We *structure* distribution through distributed collections (collections which elements are located on distinct hosts). The distribution of collections is described in a specific class called a *distribution layout manager* (DLM): DLMs encapsulate all computations depending on the type of distribution (owner retrieving and key transformation).

3.1 Seamless location of objects through classes transformations

We transform components in order to allow transparent locations of objects: transparent mapping of objects on remote hosts, and transparent communications with objects from remote hosts. The only classes we transform are those that can be accessed from remote hosts: they are marked by the programmer as *accessible* (they implement the ACCESSIBLE interface). From an *accessible* class, we generate three main classes:

- the *proxy* class, that catches the clients requests and sends them to the *implementation* class or to the *static* class;
- the *implementation* class contains all instance attributes and method implementations;
- the *static* class contains all static attributes and method implementations.

To instanciate an *accessible* class, the programmer gives an additional parameter to the constructor, identifying the *accessible* object location. A *proxy* object is created localy, and an *implementation* object is created remotely, depending on the *accessible* object location (through the `remoteNew` method). Their is only one instance of the *static* class shared by all *proxy* objects (the `staticNew` method implements this *single instance creation*). The *proxy*, *implementation* and *static* objects communicate using Java RMI.

Figure 6 shows the transformation of an *accessible* class. All classes and interfaces in grey are classes from the Java RMI packages.

3.2 Distributed collections to structure distribution

Distributed collections are collections which elements are located on distinct hosts. Distributed collections manage access to *accessible* distributed elements. The distribution of collections is described in a specific class called a *distribution layout manager* (DLM): DLMs encapsulate all computations depending on

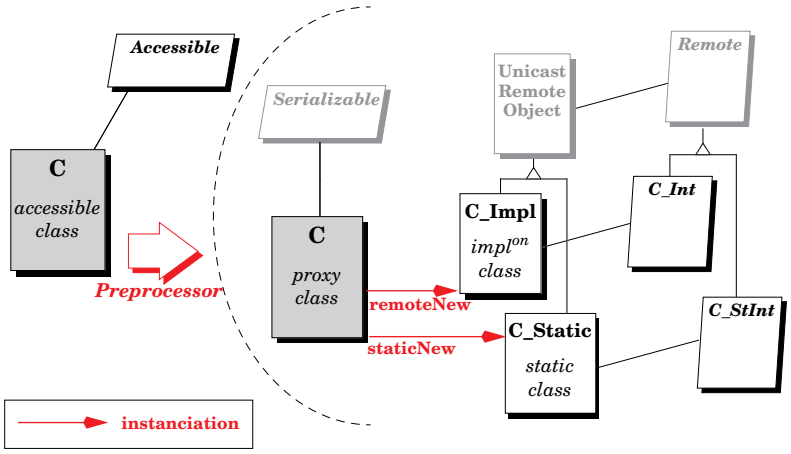

Fig. 6. *Accessible* class transformation

the type of distribution (owner retrieving and key transformation). For each type of collection, there are several DLMs, to describe different kinds of distributions (for example, for arrays, we define "HPF-like" distributions through the BlockArrayDLM or CyclicArrayDLM).

In a distributed program, a specific DLM is associated to each distributed collection. It is used by the collection to process accesses, and it is used to determine where to create an object that will be stored in a distributed collection. DLMs are associated to non distributed collections as well. In that case, they have no effect; they are included in a program to specify how to distribute it. The operators process operations on elements of distributed collections. Accesses to distributed elements are managed by the distributed collection and the seamless location of accessible objects, so there is no need to modify iterators, operators and parallel constructs.

We have implemented dynamic DLMs for dynamic collections (lists): dynamic DLMs are able to take into account dynamic informations to compute the mapping of objects at runtime. Thus, we can implement load balancing strategies through the use of dynamic DLMs: when a new active object is inserted in an "active" list in order to be activated, the DLM chooses the host where to map this object according to the processor loads. More sophisticated load balancing strategies can be implemented and tested by redefining one method of the DynamicListDLM, without any other change in the program code.

3.3 From parallel to distributed

The programmer writes his application as a non distributed parallel program, without bothering with the object locations. To transform a non distributed parallel program into a distributed program, the programmer:

- describes the objects' mapping by associating appropriate DLMs to collections,
- marks objects that can be accessed from remote hosts as *accessible* (the distributed collections' elements, for example).

Then, a preprocessor:

- transforms *accessible* classes to allow seamless location of *accessible* objects;
- adds a parameter for the distributed collection elements' creation, according to the collection DLM, to identify the object location;
- replaces non distributed collections by distributed collections

4 Performance concerns, current status and future work

Performances are one of the main concerns in scientific computing. Our environment is targeted to the Java language, using THREADS and RMI. The current implementation of the Sun JVM is not satisfying in terms of performances, but some environments now available will make it possible to use Java for high performance computing [8, 9]. The over-cost of using our framework and our code transformations for *accessible* classes is negligible.

A first release of the *Do!* environment is currently available online (`http://www.irisa.fr/CAPS/PROJECTS/Do`), comprising arrays and static DLMs. A second release integrating lists and dynamic DLMs has already been implemented, and will be available soon. We have implemented some applications using our environment and we intend to use high performance Java environments in a future release. A foreseen extension is to include object migration, in order to allow load balancing during active objects execution. This technique will also be ported to a CORBA environment, Java being used only as a coordination language.

References

1. D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, September–November 1998.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
3. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java[tm] Series. Computer and Engineering Publishing Group, 1996.
4. J.-M. Jézéquel, F. Guidec, and F. Hamelin. Parallelizing object oriented software through the reuse of parallel components. In *Object-Oriented Systems*, volume 1, pages 149–170, 1994.
5. J.-M. Jézéquel and J.-L. Pacherie. Parallel operators. In Pierre Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 275–294, Linz, Austria, July 1996. Springer Verlag.
6. E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the parallel standard template library. In *11th International Conference on Supercomputing*, pages 124–131. ACM Press, July 1997.

7. K.-P. Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, September 1993.
8. J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An efficient implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, May 1999.
9. S. Matsuoka and S. Itoh. Is Java suitable for portable high-performance computing? In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC'98)*, July 1998.

An Agent-Based Design for Problem Solving Environments

Dan C. Marinescu
(dcm@cs.purdue.edu)
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

Abstract

In this paper we examine alternative means to exploit the advantages of code mobility, object-oriented design, and agent technology for high performance distributed computing. We describe an infrastructure for Problem Solving Environments based upon software agents.

1 Introduction

A number of new initiatives and ideas for high performance distributed computing have emerged in the last few years. Object-oriented design and programming languages like Java open up intriguing new perspectives for the development of complex software systems capable to simulate physical systems of interest to computational sciences and engineering. The Java Grande initiative aims to add new constructs and to support optimization techniques needed to make the Java language more expressive and efficient for numerical simulation. If successful, this effort will lead to more robust scientific codes and increased programmer productivity.

An important side effect of the use of Java in scientific computing is code mobility. This brings us to another significant development, computing grids. Informally, a computing grid is a collection of autonomous computing platforms with different architectures, interconnected by a high-speed communication network. Computing grids are ideal for applications that have one or more of the following characteristics: (a) are naturally distributed, data collection points and programs for processing the data are scattered over a wide area network, (b) need a variety of services distributed over the network, (c) have occasional or sustained needs for large amounts of computing resources e.g. CPU cycles, large memory, vast amounts of disk space, (d) benefit from heterogeneous computing environments consisting of platforms with different architectures, (e) require a collaborative effort from users scattered over a large geographic area.

Many problems in computational sciences and engineering could benefit from the use of computing grids. Yet, scientific code mobility, a necessary condition for effective use of heterogeneous computing environments is a dream waiting to materialize. Porting a parallel program from one system to another, with a different architecture and then making it run efficiently are tedious tasks. Thus the interest of the high performance distributed computing comunity for Java.

We are cautiously optimistic regarding both aspects outlined above. At the time of this writing, Java code is still inefficient, it runs 10 to 20 times slower than C code. It is rather unlikely that the Java language will ever include a comprehensive support for numerical computations because scientific and engineering applications represent only a relatively small fraction of the intended audience for Java. Even if Java becomes the language of choice for writing scientific and engineering codes, we still have a large body of legacy codes written along the years and an "ab inito" approach, re-writing them in Java is unlikely. We need also to keep in mind that often, parallel codes require new algorithms to execute efficiently, thus code mobility in a heterogeneous system has inherent limitations. Resource management in a network of autonomous nodes and security pose formidable challenges that need to be addressed before computing grids could become a reality.

In this paper we examine alternative means to exploit the advantages of code mobility, object-oriented design, and agent technology for high performance distributed computing, at a time when Java Grande and computing grids are only a promise. To bridge the gap between promise and reality we propose to develop an infrastructure for Problem Solving Environments, PSEs, based upon software agents.

To use a biological metaphor [2], software agents form a nervous system and perform command and control functions in a Problem Solving Environment. The agents themselves relay on a distributed object system to communicate with another. Though the agents are mobile, some of the components of the PSE are tightly bound to a particular hardware platform and cannot be moved with ease.

In this paper we first review basic requirements for designing complex systems like Problem Solving Environments and the role of software agents, then we introduce a software architecture that has the potential to facilitate the design and implementation of PSE and to make the resulting systems less brittle.

The basic design philosophy of the Bond system is described in [1], [?] [2], the security aspects of Bond are presented in [8], an application of Bond to the design of software agents for a network of PDE solvers is discussed in [11] and an in depth view of the design of Problem Solving Environments using Bond is given in [10]. The Bond system was released in mid March 1999.

2 Agents, Problem Solving Environments, and Software Composition

Software agents seem to be at the center of attention in the computer science community. Yet different groups have radically different views of what software agents are, [6], what applications could benefit from the agent technology, and many have a difficult time sorting out the reality from fiction in this rapidly moving field and accepting the representation that software agents provide a "magic bullet" for all problems. The concept of an agent was introduced by the AI community a decade ago, [3]. An AI agent exhibits an autonomous behavior and has inferential abilities. A considerable body of work is devoted to agents able to meet the Turing test by emulating human behavior [9]. Such agents are useful for a variety of applications in science and engineering e.g. deep space explorations, robots, and so on.

Our view of an agent is slightly different [1]. For us a software agent is an abstraction for building complex systems. An agent is an *active mobile object that may or may not have inferential abilities.* Our main concern is to develop a constructive framework for building collaborative agents out of ready-made components and to use this infrastructure for building complex systems including Problem Solving Environments, PSEs. The primary function of a Problem Solving Environment is to assist computational scientists and engineers to carry out complex computations involving multiple programs and data sets. We use the term workflow and metaprogram interchangeably, to denote both the static and the dynamic aspects of this set of computations. We argue that there are several classes of high performance computing applications that can greatly benefit from the use of agent-based PSEs:

- Naturally distributed applications,
- Data intensive applications.
- Applications with data-dependent or non-deterministic workflows.
- Parallel applications based upon domain data decomposition and legacy codes.

Many problems in computational science and engineering are naturally distributed, involve large groups of scientists and engineers, large collections of experimental data and theoretical models, as well as multiple programs developed independently and possibly running on systems with different architectures. Major tasks including coordination of various activities, enforcing a discipline in the collaborative effort, discovering services provided by various members of the team, transporting data from the producer site to the consumer site, and others can and should be delegated to a Problem Solving Environment. The primary functions of agents in such an environment are: scheduling and control, resource discovery, management of local resources, use-level resource management, and workflow management.

Data-intensive applications are common to many experimental sciences and engineering design applications. As sensor-based applications become pervasive, new classes of data-intensive applications are likely to emerge. An important function of the PSE is to support data annotation. Once metadata describing the actual data is available, agents can automatically control the workflow, allow backtracking and restart computations with new parameters of the models.

Applications like climate and oceanographic modeling often relay on many data collection points and the actual workflow depends both upon the availability of the data and the confidence we have in the data. The main function of the agents in such cases is the dynamic generation of the workflows based upon available information.

Last, but not least, in some cases one can achieve parallelism using sequential legacy codes. Whenever we can apply a divide and conquer methodology based upon the partitions of the data into sub-domains, solve the problem independently in each sub-domain, and then resolve with ease the eventual conflicts between the individual workers we have an appealing alternative to code parallelization. The agents should be capable to coordinate the execution and mediate conflicts.

The idea of building a program out of ready-made components has been around since the dawn of the computing age, backworldsmen have practiced it very successfully. Most scientific programs we are familiar with, use mathematical libraries, parallel programs use communication libraries, graphics programs rely on graphics libraries, and so on.

Modern programming languages like Java, take the composition process one step further. A software component, be it a package, or a function, carries with itself a number of properties that can be queried and/or set to specific values to customize the component according to the needs of an application which wishes to embed the component. The mechanism supporting these functions is called *introspection*. Properties can even be queried at execution time. *Reflection* mechanisms allow us to determine run time conditions, for example the source of an event generated during the computation. The reader may recognize the reference to the Java Beans but other *component architectures* exists, Active X based on Microsoft's COM and LiveConnect from Netscape to name a few.

Can these ideas be extended to other types of computational objects besides software components, for example to data, services, and hardware components? What can be achieved by creating *metaobjects* describing *network objects* like programs, data or hardware? We use here the term "object" rather loosely, but later on it will become clear that the architecture we envision is intimately tied to object-oriented concepts. We talk about network objects to acknowledge that we are concerned with an environment where programs and data are distributed on autonomous nodes interconnected by high speed networks.

Often the components are legacy codes that cannot be modified with ease. In this case we can wrap around legacy programs newly created components called software agents. Each wrapper is tailored to the specific legacy application. Then interoperability between components is ensured by federation of agents.

3 An Infrastructure for Problem Solving Environments

Bond, [1], is a distributed-object, message-oriented system, it uses KQML [5], as a meta-language for inter-object communication. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. A performative is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order-independent.

The infrastructure provided by Bond supports basic object manipulation, inter-object communication, local directory and local configuration services, a distributed awareness mechanisms, probes for security and monitoring functions, and graphics user interfaces and utilities.

Shadows are proxies for remote objects. Realization of a shadow provides for instantiation of remote objects. Collections of shadows form *virtual networks* of objects.

Residents are active Bond objects running at a *Bond address*. A resident is a container for a collection of objects including communicator, directory, configuration, and awareness objects. *Subprotocols* are closed subsets of KQML messages. Objects inherit subprotocols. The discovery subprotocol allows an object to determine the set of subprotocols understood by another object. Other subprotocols, monitoring, security, agent control, and the property access subprotocol understood by all objects.

The transport mechanism between Bond residents is provided by a *communicator* object with four interchangeable communication engines based upon: (a) UDP, (b) TCP, (c) Infospheres, (info.net), and (d) IP Multicast protocols.

Probes are objects attached dynamically to Bond objects to augment their ability to understand new subprotocols and support new functionality. A *security probe* screens incoming and outgoing messages to an object. The security framework supports two authentication models, one based upon *username, plain password* and one based upon the*Challenge Handshake Authentication Protocol, CHAP*. Two access control models are supported, one based upon the *IP address (firewall)* and one based upon an *access control list*. *Monitoring probes* implement a subscription-based monitoring model. *An autoprobe* allows loading of probes on demand.

The *distributed awareness* mechanism provides information about other residents and individual objects in the network. This information is piggy-backed on regular messages exchanged among objects to reduce the overhead of supporting this mechanism. An object may be aware of objects it has never communicated with. The distributed awareness mechanism and the discovery subprotocol reflect our design decision to reduce the need for global services like directory service and interface repositories.

Several distributed object systems provide support for agents. Infospheres

(//www.infospheres.caltech.edu/) and Bond are academic research projects, while IBM Aglets (www.trl.ibm.co.jp/aglets/index.html) and Objectspace Voyager (//www.objectspace.com) are commercial systems.

A first distinctive feature of the Bond architecture, described in more detail in [1] is that agents are native components of the system. This guarantees that agents and objects can communicate with one another and the same communication fabric is used by the entire population of objects. Another distinctive trait of our approach is that we provide middleware, a software layer to facilitate the development of a hopefully wide range of applications of network computing. We are thus forced to pay close attentions to the software engineering aspects of agent development, in particular to software reuse. We decided to provide a framework for assembly of agents out of components, some of them reusable. This is possible due to the agent model we overview now.

We view an agent as a finite-state machine, with a strategy associated with every state, a model of the world, and an agenda as shown in Figure 1. Upon entering a state the strategy or strategies associated with that state are activated and various actions are triggered. The model is the "memory" of the agent, it reflects the knowledge the agent has access to, as well as the state of the agent. Transitions from one state to another are triggered by internal conditions determined by the completion code of the strategy, e.g. success or failure, or by messages from other agents or objects.

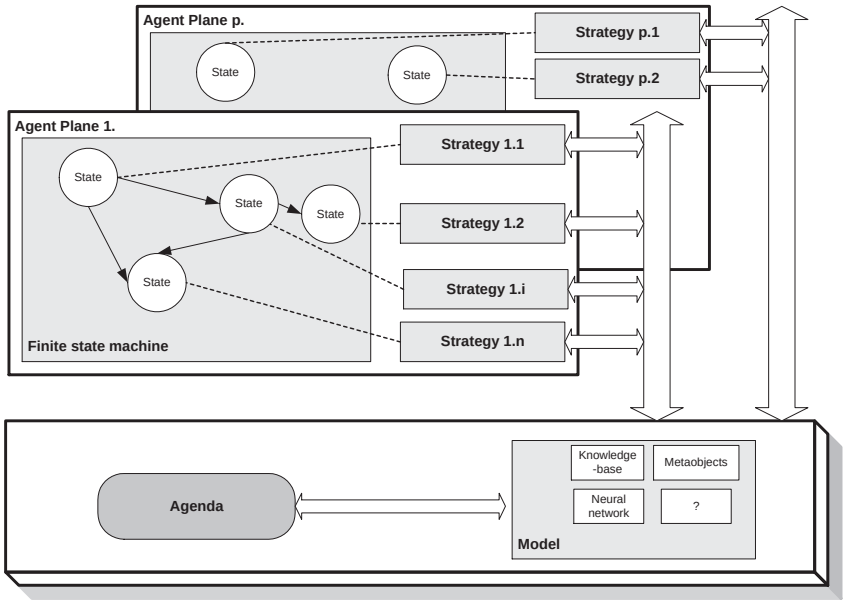

Figure 1: The abstract model of a Bond Agent

The finite-state machine description of an agent can be provided at mul-

tiple granularity levels, a course-grain description contains a few states with complex strategies, a fine-grain description consists of a large number of states with simple strategies. The strategies are the reusable elements in our software architecture and granularity of the finite state machine of an agent should be determined to maximize the number of ready made strategies used for the agent. We have identified a number of common actions and we started building a strategy repository. Examples of actions packed into strategies are: starting up one or more agents, writing into the model of another agent, starting up a legacy application, data staging and so on. Ideally, we would like to assemble an agent without the need to program, using ready-made strategies from repositories.

Another feature of our software agent model is the ability to assemble an agent dynamically from a "blueprint", a text file describing the states, the transitions, and the model of the agent. Every Bond-enabled site has an "agent factory" capable to create an agent from its blueprint. The blueprint can be embedded into a message, or the URL of the blueprint can be provided to the agent factory. Once an agent was created, the agent control subprotocol can be used to control it from a remote site.

In addition to favoring reusability, the software agent model we propose has other useful features. First, it allows a smooth integration of increasingly complex behavior into agents. For example, consider a scheduling agent with a mapping state and a mapping strategy. Given a task and a set of target hosts capable to execute the task, the agent will map the task to one of the hosts subject to some optimization criteria. We may start with a simple strategy, select randomly one of the target hosts. Once we are convinced that the scheduling agent works well, we may replace the mapping strategy with one based upon an inference engine with access to a database of past performance. The scheduling agent will perform a more intelligent mapping with the new strategy. Second, the model supports agent mobility. A blueprint can be modified dynamically and an additional state can be inserted before a transition takes place. For example a "suspend" new state can be added and the "suspend" strategy be concatenated with the strategy associated with any state. Upon entering the "suspend" state the agent can be migrated elsewhere. All we need to do is send the blueprint and the model to the new site and make sure that the new site has access to the strategies associated with the states the agent may traverse in the future. The dynamic alteration of the finite state machine of an agent can be used to create a "snapshot" of a group of collaborating agents and help debug a complex system.

We have integrated into Bond the JESS expert shell developed at Sandia National Laboratory as a distinct strategy able to support reasoning. Bond messages allow for embedded programs written in JPython and KIF.

Agent security is a critical issue for the system because the ability to assemble and control agents remotely as well as agent mobility, provide unlimited opportunities for system penetration. Once again the fact that agents are native Bond objects leads to an elegant solution to the security aspect of agent design. Any Bond object, agents included, can be augmented dynamically with a security probe providing a defense perimeter and screening all incoming and

outgoing messages.

The components of a Bond agent shown in Figure 1 are:

- The **model of the world** is a container object which contains the information the agent has about its environment. This information is stored in the form of dynamic properties of the model object. There is no restriction of the format of this information: it can be a knowledge base or an ontology composed of logical facts and predicates, a pre-trained neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc).

- The **agenda** of the agent, which defines the goal of the agent. The agenda is in itself an object, which implements a boolean and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The distance function may be used by the strategies to choose their actions.

- The **finite state machine** of the agent. Each state has an assigned strategy which defines the behavior of the agent in that state. An agent can change its state by performing *transitions*. Transitions are triggered by internal or external *events*. External events are messages sent by other agents or objects. The set of external messages which trigger transitions in the finite state machine of the agent defines the *control subprotocol* of the agent.

- Each state on an agent has a **strategy** defining the behavior of the agent in that state. Each strategy performs actions in an infinite cycle until the agenda is accomplished or the state is changed. Actions are considered atomic from the agent's point of view, external or internal events interrupt the agent only between actions. Each action is defined exclusively by the agenda of the agent and the current model. A strategy can terminate by triggering a transition by generating an internal event. After the transition the agent moves in a new state where a different strategy defines the behavior.

All components of the Bond system are objects, thus Bond agents can be assembled dynamically and even modified at runtime. The behavior of an agent is uniquely determined by its model (the model also contains the state which defines the current strategy). The model can be saved, transferred over the network.

A `bondAgent` can be created statically, or dynamically by a factory object `bondAgentFactory` using a *blueprint*. The factory object generates the components of the agent either by creating them, either by loading them from persistent storage. The agent creation process is summarized in Figure 2

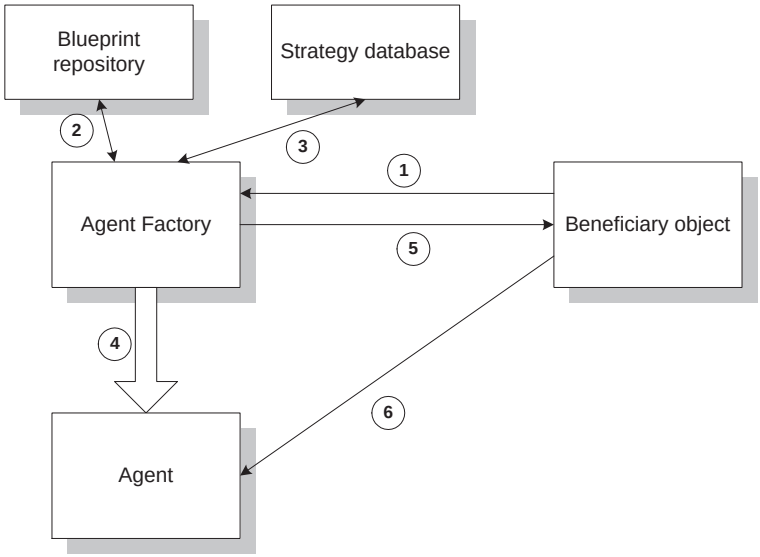

Figure 2: Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory (2) The blueprint is fetched by the agent factory from a repository or extracted from the message (3) The strategies are loaded from the strategy database (4) The agent is created (5) The id of the agent is communicated back to the beneficiary, and (6) The beneficiary object controls the new agent

4 Conclusions

Bond is a Java written, agent-based, distributed-object system we have developed for the past few years. Bond provides a framework for interoperability based upon (1) metaobjects that provide information about network objects, and (2) software agents capable to use the information about network objects and carry out complex tasks.

Some of the components we propose e.g. the agent framework, the scheduling agents, the monitoring and security frameworks, are generic and we expect that they will be included in other applications like distance learning, or possibly electronic commerce.

The design of a Problem Solving Environment based upon a network of PDE solvers is one of the applications of Bond that illustrates the advantages of a component based architecture versus a monolithic design of a PSE.

A beta version of the Bond system was released in mid March 1999 under an open source license, LPGL, and can be downloaded from our web site, **http://bond.cs.purdue.edu**.

Acknowledgments

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, and by a grant from the Intel Corporation.

References

[1] Bölöni, L., and D.C. Marinescu, *An Object-Oriented Framework for Building Collaborative Network Agents*. Kluever Publishers, 1999 (to appear).

[2] Bölöni, L., R. Hao, K.K. Jun, and D.C. Marinescu, *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System*, Proc. Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems, in LNCS, vol 1586, Springer Verlag, 1999, pp. 275-283.

[3] Bradshaw, J. M., *An Introduction to Software Agents*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 3-46, 1997.

[4] *The Grid, Blueprint for a New Computing Infrastructure*, Foster, I. and C. Kesselman, Eds., Morgan Kaufmann, (1998).

[5] Finn, T., Y. Labrou, and J. Mayfield, *KQML as an Agent Communication Language*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 291-316, 1997.

[6] Franklin, S. and A. Graesser, *Is it an Agent, or just a Program?*, Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer Verlag, 1996.

[7] Genesereth, M. R., *An Agent-Based Framework for Interoperability*, in J. M. Bradshaw Ed. *Software Agents*, MIT Press, pp. 317-345, 1997.

[8] Hao, R., L. Bölöni, K.K. Jun, and D.C. Marinescu, *An Aspect-Oriented Approach to Distributed Object Security*, Proc. 4-th IEEE Symp. on Computers and Communications, IEEE Press, (1999), (in print).

[9] Jennings, N. R., K. Sycara, M. Woolridge, *A Roadmap of Agent Research and Development*, in Autonomous Agents and Multi-Agent Systems, 1, pp. 275-306, 1998.

[10] Marinescu, D.C., and Bölöni L., *A Component-Based Architecture for Problem Solving Environments*, 1999, (in preparation).

[11] Tsompanopoulou, P., L. Bölöni, D.C. Marinescu, and J.R.Rice, *The Design of Software Agents for a Network of PDE Solvers* Proceedings of Workshop on Autonomous Agents, IEEE Press, 1999 (in press).

An Object-Based Metasystem for Distributed High Performance Simulation and Product Realization

Victor P. Holmes, John M. Linebarger,
David J. Miller, and Ruthe L. Vandewart

Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico, USA, 87185
vpholme@sandia.gov
http://www.pdo.sandia.gov/SI.html

Abstract. The Simulation Intranet/Product Database Operator (SI/PDO) is a cadre system which comprises one element of a multi-disciplinary distributed and distance computing initiative known as $DisCom^2$ at Sandia National Laboratories. The SI/PDO is an architecture for satisfying Sandia's long term goal of providing integrated software services for high fidelity full physics simulations in a high performance, distributed, and distance computing environment. This paper presents the initial requirements, design, and implementation of the SI/PDO which is based on a grid-like network computing architecture. The major logical elements of the architecture include the desktop, distributed business objects, and data persistence. ...

1 Introduction

The purpose of the Simulation Intranet/Product Database Operator (SI/PDO) architecture is to satisfy Sandia National Laboratories' long term goal of providing designers and analysts an integrated set of product realization and virtual prototyping services which include high fidelity simulations in a high performance (HP), distributed, and distance computing environment. The initial focus of the architecture is on the development of a distributed object framework which allows users Web-based desktop access to applications that require high performance distributed resources for modeling, simulation, analysis, and visualization. This framework satisfies the following characteristics:

- works within a heterogeneous, distributed computing environment,
- is object-oriented and based on open standards,
- exhibits network transparency, containing components which are decoupled,
- makes new and existing applications appear as distributed object services,
- provides for sequencing, launching, and monitoring of sets of applications,
- coordinates archival and retrieval of information,
- provides a consistent, integrated operator interface.

1.1 Architectural Thrusts

There are three main thrusts addressed by the first implementation of this framework. First and foremost, the framework architecture provides end users transparent desktop access to Sandia's Computational Plant (CPLANT), a massively parallel computing resource constructed of commodity parts. This involves integration of distributed object computing and Web-based computing with HP computing, areas which have in the past taken separate research paths and now are merging to create a component-based architecture for modeling, simulation, and analysis. Technologies being employed include object request brokers (ORBs), object-oriented databases (ODBMS), and Java Beans, coupled with applications using the Message-Passing Interface (MPI) and parallel visualization techniques.

The second aspect of the framework architecture involves integrating visualization with analysis to monitor progress and potentially inject computational steering. HP computing normally involves long running batch-oriented computations followed by complex postprocessing to finally visualize the results. In some cases, the analysis has to be resubmitted with modified component meshes and parameters. In the SI/PDO model, the framework can provide image snapshots of simulation time steps to the desktop such that users can determine if the analysis is on track without having to wait until it is completed. Eventually, the framework will expand upon this *spyglass* concept and allow the user to interrupt an analysis to tweak the mesh or parameters, and ultimately use the power of the CPLANT clusters to perform real-time parallel visualization of data as it is generated.

The third thrust involves knowledge management through a concept known as the Product Database Operator or PDO. This concept includes the ability to capture large quantities of data generated by these analyses in an object repository, classify this data into a taxonomy which is consistent with the Laboratories' business model, and then make relevant information available to the users to solve new problems and achieve new insights, particularly in the area of nuclear component stockpile stewardship.

1.2 Related Work in Distributed Frameworks

Although this area of research is fairly new, numerous framework architectures are being developed which contribute to the advancement of the technology. In addition, with the growing popularity of the Java programming language, many researchers are pursuing Java-based efforts to establish software infrastructures for distributed and distance computing. These include ATLAS [1], Charlotte [2], ParaWeb [3], Popcorn [4], and Javelin [5]. All of these Java-based projects attempt to provide capabilities for parallel applications in heterogeneous environments. The use of Java for building distributed systems will continue to flourish as Internet-based programming becomes more viable. However, the more established approaches incorporate Java-based interfaces but do not rely entirely on this language platform. These include GLOBUS [6], Legion [7], CONDOR [8], and WebOS [9].

2 Component Model Description

The three thrusts discussed above are realized within a grid-based network computing architecture model consisting of desktop clients, distributed business objects, and data persistence. The following sections discuss the general characteristics of each of these elements. A subset of these characteristics has been implemented within the prototype SI/PDO metasystem.

2.1 Web Top

A web browser is viewed as the user's primary desktop operating environment. The browser provides access to applications and services required to accomplish laboratory missions. The browser component of a grid-based architecture is concerned with aspects of presentation without knowledge of business rules. It provides a user-centered, document-centered, coarse-grained, stateless world view. In the context of the SI/PDO, the browser displays the *web pages* which provide access to modeling, simulation, and analysis capabilities. Java Bean components are used to represent these applications and services. An applet serves as the container or *bean box* for graphically programming a simulation sequence, configuring each simulation component in the sequence (customizers, introspection), invoking the sequence (event model), and monitoring its progress. This approach provides a component-based architecture which transforms legacy applications into application components, or *business objects.*

2.2 Distributed Access to Applications and Services

Transparent connections and communication must be provided between the Web Top and the distributed services. This messaging element of the grid architecture provides the notion of distributed blobs of computing resources which are available to the SI/PDO. These resources may be local to a site or remotely accessed, and the messaging interface makes them appear as if they are on the user's local machine. The initial implementation uses the Java Bean event model coupled with event model *adapters* which can potentially accommodate various protocols. These adapters provide the transparent mechanisms for browser-based beans to connect and communicate with the distributed applications and services. Currently the adapters are implemented using CORBA and IIOP.

2.3 Modeling, Simulation, and Analysis Business Objects

The business objects in the architecture are the legacy codes and new codes provided by Sandia scientists and commercial vendors which implement the services required by the Laboratories' missions and are accessed through the SI/PDO. These applications should be presentation and data storage independent, usually have connections with other business objects, and may need to store and retrieve information from the persistent store. As opposed to the Web client, business objects provide an application-centered, fine-grained, stateful world view.

To achieve the concept of business objects, most legacy codes require wrappers which handle the distributed object aspects of their use. The encapsulation strategy should be a component-first approach which involves performing a domain analysis to generate an object model, identifying public interfaces for this object model, and encapsulating the legacy application to populate the specific functions of this public interface. A complete wrapper should perform connection protocol management, data translation and information processing, error detection and recovery, and environment management, which includes insulation of the users from changes and upgrades.

2.4 Knowledge Management

The final element of the architecture, data persistence, should ultimately take the form of a knowledge management system. The value of Sandia's information assets requires more than just the ability to store and retrieve them in a distributed fashion. It should also be possible to dynamically match information to specific processes or unknown situations and leverage that information to achieve new results and insights into mission-related problems. Some key elements of a knowledge management system include distributed object databases and associated tools for legacy data conversion, knowledge creation analysis, collaboration, web content management, intelligent agent implementation, and visualization. Functions which transform information into knowledge include capturing data in an object repository and organizing it into a taxonomy which reflects the SI/PDO business model, the ability to make information available to a knowledge seeker, and the application of that knowledge to solve new problems.

3 Architectural Component Design

A generalized pictorial representation of the architecture is shown in Figure 1. On the left is the desktop environment and on the right are the distributed computing resources which are available. A Java-based applet running inside the browser provides a prototype desktop environment. The designer uses this common desktop environment to configure, link together, and launch various applications, and each launched application transparently locates, connects to, or acquires the appropriate distributed resources for that application.

For legacy codes, there are wrappers which allow older codes to become network-aware and behave as distributed object services. For new codes, they should be developed as network-based components from the start. The wrappers represent *business* objects. Sandia's business in this context includes applications such as solid modeling, meshing, finite element analysis, and visualization. Some codes execute on workstations and others are launched on HP massively parallel computers, depending upon the code's requirements and the users' needs.

Finally, data persistence and knowledge management are represented as a PDO repository in Figure 1. Both the wrappers and the desktop are capable of communicating with the PDO repository. All distributed communication is performed using CORBA.

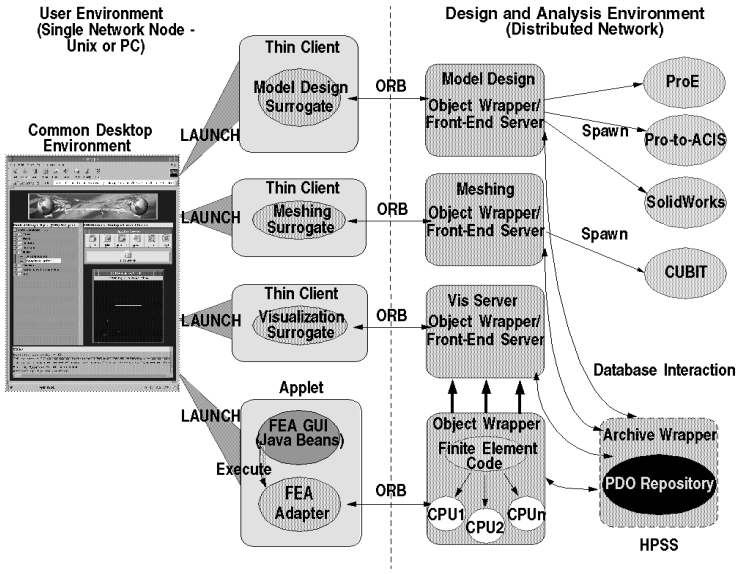

Fig. 1. Generalized SI/PDO Architecture for Modeling and Simulation

3.1 Desktop Components

User Interface Applet. Figure 2. illustrates a prototype display of the Web Top interface. The desktop applet consists of three panels. The panel on the left serves as a navigation panel and contains a tree of products and services available to that user in the database. The panel on the right serves as a workspace panel where the user can interact with the various services. If an application has a GUI associated with it, this GUI can be displayed within the workspace panel. In addition, images being sent from a visualization service may be displayed here as well. The bottom panel serves as a status display for messages from both desktop and distributed services.

Beans for Applications and Services. A PDO bean provides desktop access to the database. It retrieves all of the PDO's from the database and builds the navigation tree. By selecting a product and service from the tree, the user generates an event which causes the PDO bean to instantiate an application or service bean. The names of the beans are available from the database, and therefore, any new application or service can be plugged into the framework by creating a bean for it which adheres to some minimal design patterns, and then providing meta-data for the database about the new application or service. The core framework code itself (applet and PDO bean) does not need to be modified. The creation of application and service beans allows the user to configure,

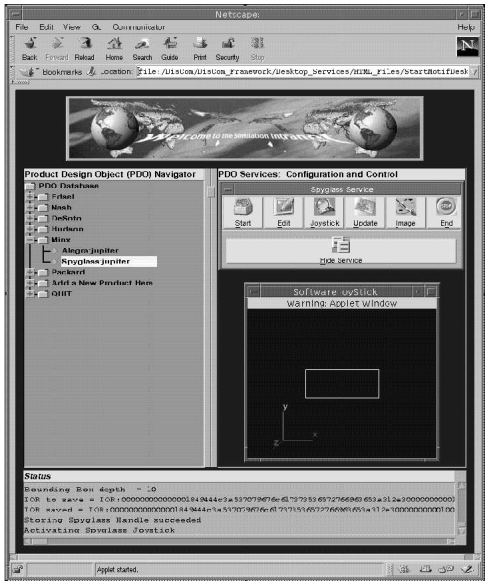

Fig. 2. Browser-Based Prototype User Interface

launch, monitor, and display results for modeling, simulation, and analysis tasks associated with a product.

Bean Adapters for Distributed Communication. As discussed above, each application bean will normally include an adapter class for distributed communication. There are several advantages to using adapters. The adapter provides a nice encapsulation mechanism for communication code and makes the design more object-oriented in nature. This encapsulation subsequently allows for multiple implementations of the adapter without affecting the bean application code. Such multiple implementations allow for different communication technologies and protocols. Another advantage of an adapter is that it can be implemented as a separate thread which prevents bottlenecks from occurring on the desktop. When the user submits a request to the service, it can be carried out by this thread without affecting the user's ability to perform subsequent desktop interactions. Finally, various multiplexing and queuing algorithms can be inserted into adapters, allowing the user to batch requests to a service without having to wait for one to complete before issuing another one.

3.2 Application and Service Components

Persistent Services. An object-oriented database is a key component of the framework, maintaining information and methods for each product design and analysis. A persistent CORBA interface to the database provides access from the desktop or from applications which are running on workstation clusters or high

performance computers. This interface provides the operations needed to initiate user sessions, launch applications and services, retrieve status, and manage the information generated.

In addition, because some of the ORBs used by the framework may not provide an automatic launching capability of application servers, it is necessary for the framework to provide this so that desktop users do not have to deal with such details. For the short term, this capability will be provided by a simple CORBA-based Launch server which must be persistently active on machines where applications are executed. This server implements a simple IDL interface containing a single launch operation for starting up CORBA-based applications.

Finite Element Application Service. The Alegra application is a structural dynamics finite element code written in C++. It was selected as the first candidate application for the framework. It was made CORBA-aware so that it could be accessed in a distributed fashion from the desktop using the Web Top aspects of the framework. Therefore, an IDL interface was defined for Alegra operations, and the object implementation of this interface, or Alegra wrapper, serves as a front end to the Alegra software. The IDL interface provides operations for starting an execution, retrieving status of the execution, and aborting or terminating an execution.

Visualization Service. A *spyglass* service was developed which enables users to view early results of a computation. It consists of a CORBA-based server which awaits signals from an analysis code indicating that simulation time step data is available. This data is accessed, the geometry of requested objects is extracted and rendered into an off-line frame buffer (using the Mesa graphics library, which is OpenGL-like), and a JPEG image is created from the frame buffer. This image is sent both to the PDO and to the Web Top.

If the Web Top is active, it can perform *remote navigation* and request that an image be regenerated from another perspective (i.e., camera position). A Java3D-based object is present on the Web Top that contains a bounding box of the objects in the image. Six degree-of-freedom navigation is possible within that Java3D object, and the bounding box can be repositioned accordingly. When the desired point-of-view of the bounding box is reached, the viewpoint transformation matrix is extracted and sent to the spyglass service which uses it to reposition the camera and render another JPEG image from that perspective.

3.3 Product Database Operator (PDO)

In the Sandia environment, the maintenance and accessibility of details regarding the development of a complex product is vitally important. A primary reason for this is accountability, but there is also the fact that certain products have a very long projected lifetime and must occasionally be upgraded or have various components replaced. Each modification must undergo simulation testing with the original parts of the product. Another aspect of this problem is the aging

workforce and the loss of expertise that comes with attrition. Often, it is just not possible to return to the original designer or analyst of the product or component for further information.

All of this leads to the necessity of keeping a repository with large amounts of disparate information that should be easily accessible. Currently at Sandia there are multiple established databases that store some of this information. The databases are queried by the person needing the data, and then the query results are manually combined with other data sources to accumulate all the information necessary to complete the task at hand. This is a massive bookkeeping effort.

The Product Database Operator (PDO) is a software system which combines the capabilities of a commercial Object Oriented Database Management System (ODBMS) with a Common Object Request Broker (CORBA) server to support an integrated, functional information repository. The PDO provides persistent storage to establish, maintain, and make available programmatically (to a CORBA client) product information, along with system, configuration, or other details required. All of this information that is needed to execute simulations, alter the design of any components, and incorporate new analysis methods is made available without burdening the user with the task of remembering such details. The need for such automation is accentuated by the potential for maintaining numerous binaries and libraries for particular software due to the use of heterogeneous systems, and by the utilization of remote, unfamiliar platforms as they are made available through network expansion.

The PDO repository provides a valuable archive of product development information and meta-data relating to evolution of a product and its components. Complex persistent information objects capture the design state, recipe, requirements, results, software tools, and access information for a legacy or in-development product. Product information in the PDO includes not only the specifics necessary to locate any information relating to the product, but also includes methods to actually do the access or processing of a particular aspect of the problem, such as starting a program or transferring a file to a needed location. Methods can locate and deliver necessary inputs to the application and receive and retain outputs from that application. Designers, analysts and developers, each from their differing viewpoints, make use of the PDO to record and deliver the detailed and changing information necessary to integrate data, procedures, and locations for executing complex software applications and for providing documentation of decisions made and lessons learned.

Product information can be linked together to form composite products and to do regression testing on software modifications and upgrades. Products may be analyzed with alternative software to not only validate the composition and construction of the product, but also to validate and calibrate the analysis software. Event notification is supported to enable active database functionality and to synchronize steps of a process. The PDO can store documentation and help files and maintain the status of an in-progress analysis. The use of shared objects in the database enables certain information to exist in only one instance, implying that a single database update will modify many information items. This

minimizes the impact of system upgrades and repartitioning, new development, new hardware, and other changes inevitable in a dynamic, developing system.

4 Conclusions

The framework is currently in an initial prototype phase. A focused problem domain for the first implementation involves providing desktop access to the Alegra finite element code executing on a Linux-based CPLANT Miata HP cluster. In addition, the framework provides a visualization capability which feeds back snapshots of data generated by Alegra at each time step. Figure 3. illustrates the capabilities provided by the first framework prototype.

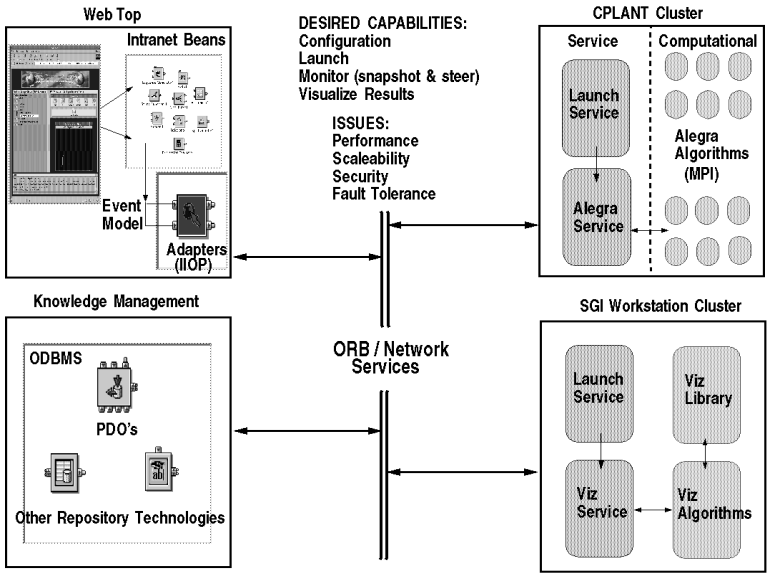

Fig. 3. First Instantiation of the SI/PDO Architecture

On the Web Top are the components to configure, execute, monitor, and display results of an Alegra run. Coupled to the beans through the bean event model are the adapters which handle the IIOP/CORBA protocol used to communicate with application wrappers and the PDO. The PDO repository coordinates all of the activities associated with a particular product development cycle. On the CPLANT side, a launch service automatically launches application wrappers, and wrappers exist for the Alegra and Visualization services. The Alegra service is a front-end to the native Alegra code and has the capability to call CPLANT

services to load the code into the computational nodes, start execution, and monitor status. The Visualization service encapsulates the visualization algorithms for offline rendering and generation of JPEG images. Because the capability does not yet exist to allow direct communication of data in parallel from applications to visualization codes, Alegra generates files on shared file systems, and then a CORBA-based Alegra status task is used to notify the Visualization service when data is available for rendering. When the image is completed, CORBA is again used to ship it back to the desktop where it is displayed.

There is obviously much work remaining in the development of the framework and the overall SI/PDO architecture. The framework needs to be extended and productionized for actual mission-based use. To do this more easily, reuseable design patterns are being documented which can be applied to extrapolating the framework for other applications, services, or problem domains. In addition, further research and development will involve incorporating fault tolerant features, investigating impacts of distance computing on some of the framework's architectural components such as the adapters and network protocols, looking at performance, scalability, security, and other distributed resource management (DRM) issues, and continuing the work on the PDO concept.

References

1. Baldeschwieler, J. E., R. D. Blumofe, and E. A. Brewer, ATLAS: An Infrastructure for Global Computing, Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, 1996.
2. Baratloo, A., M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte: Metacomputing on the Web, Proceedings of the 9th Conference on Parallel and Distributed Computing Systems, 1996.
3. Brecht, T., H. Sandhu, M. Shan, and J. Talbot, ParaWeb: Towards World-Wide Supercomputing, Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, 1996.
4. Camiel, N., S. London, N. Nisan, and O. Regev, The POPCORN Project: Distributed Computation over the Internet in Java, 6th International World Wide Web Conference, April 1997.
5. Christiansen, B. O., P. Cappello, M. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, Javelin: Internet-Based Parallel Computing Using Java, Department of Computer Science, University of California, Santa Barbara, 1998.
6. Foster, I., and C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications, 1997.
7. Grimshaw, A. S., W. A. Wulf, and the Legion team, The Legion Vision of a Worldwide Virtual Computer, Communications of the ACM, 40 (1), January 1997.
8. Litzkow, M., M. Livny, and M. W. Mutka, Condor - A Hunter of Idle Workstations, Proceedings of the 8th International Conference of Distributed Computing Systems, June 1988.
9. Vahdat, A., P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin, WebOS: Operating System Services For Wide Area Applications, Technical Report CSD-97-938, UC Berkeley, 1997.

Molecular Dynamics with C++. An object oriented approach.

Matthias Müller

[1] Institute for Computer Applications I
Pfaffenwaldring 27,
D-70569 Stuttgart, Germany
matthias@ica1.uni-stuttgart.de
[2] High Performance Computing Center (HLRS)
Allmandring 30
D-70550 Stuttgart, Germany

Abstract. The complexity of parallel computing hardware calls for software that eases the development of numerical models for researchers that do not have a deep knowledge of the platforms' communication strategies and that do not have the time to write efficient code themselves. We show how "*container*" data structures can address the needs of many potential users of parallel machines that have so far been deterred by the complexity of parallelizing code. These example presented here is a molecular dynamics simulation.
Our approach uses several ideas from the Standard Template Library (STL). Particles are stored in a Template Container that also distributes them among the processors in the parallel version. For the integration applicators are applied to the particles. The force calculation is done by a method "for_each_pair" similar to "for_each" that takes an extended applicator as an argument. This extended applicator takes two particles as its arguments and is applied to every potential interaction pair.

1 Principles of Molecular Dynamics

The principle of a Molecular Dynamics (MD) program is: find the interaction partners for every particle, calculate the forces between the particles and integrate the equations of motion. A real MD program can be rather complex, i.e. by using optimization techniques like linked cell algorithms for short range interaction. Different MD programs need different types of particles with different degrees of freedom and forces. Parallel computers with distributed memory add another level of complexity: particles have to be distributed among the processors and interactions across processor boundaries have to be calculated.

With this in mind we developed a MD package for short range interaction. It allows the easy modification of particle type, interaction force, and integration scheme and allows to switch between the serial and parallel implementation without changing the main program. We decided to use C++ and the Message Passing Interface (MPI) for implementation because both are available on most

supercomputers as well as on the workstations where the code is developed. In addition C++ was designed to have no or only a minimal runtime overhead while still offering the advantages of an object oriented language.

The parallel paradigm applied here is domain decomposition (see Fig. 1), therefore every CPU is responsible for a part of the domain and has to exchange informations about the border of its domain with its adjacent neighbors. Instead of exchanging them directly with all 8 or 26 neighbors in two or three dimensions respectively the Plimpton scheme[1] is applied here (see Fig. 2).

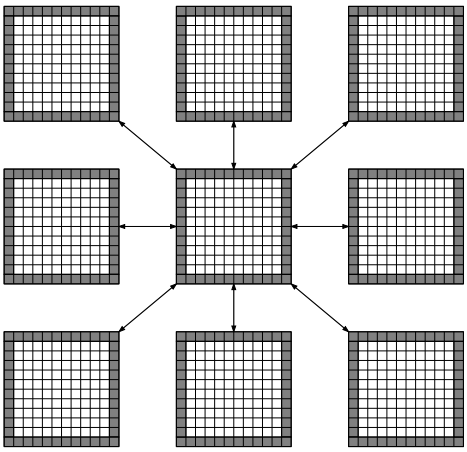

Fig. 1. Domain decomposition. The cells of the linked cell algorithm containing data that has to be shared with the neighbors (*shadow cells*) are in gray.

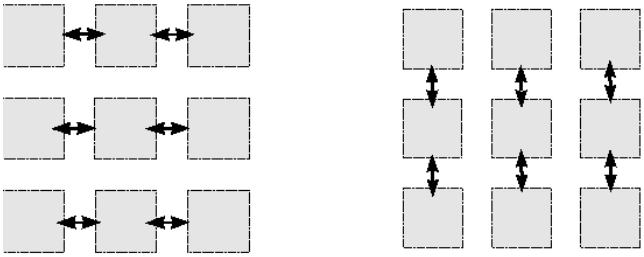

Fig. 2. Plimpton scheme to exchange *shadow cells*.

2 Design of the particle container

To free the ordinary programmer from the details of bookkeeping and still provide him with the freedom to implement his own particle types, forces and integration scheme we decided to provide a template container for particles similar to the containers that are provided by the standard template library (STL). In addition to the normal members provided by a STL container a particle container provides a template member `for_each_pair` to allow for iterations over pairs of particles. Here the function object concept of STL is applied, the force class has to supply an operator() for two particles:

```
template<class PARTICLE>
class Force{
public:
  inline void operator()(PARTICLE & p1 , PARTICLE & p2){
     Vector force;
     // calculation of f goes here
     p1.force += force;
     p2.force -= force;  // actio = reactio
  }
};
```

This example also demonstrates the possibility to formulate the algorithm in a natural, dimension independent way. To avoid possible performance problems due to the many small vectors used, we employ the template expression technique [2] to perform loop unrolling, inlining and other optimizations at compile time.

The main loop of a MD simulation will look similar to this example.

```
PartContLC<Particle,3> box(ll,ur,bound_cond,cut_off);
double dt=0.1;                              // time step
MD_int_vv_start<Particle>  int_start(dt);   // integration
MD_int_vv_finish<Particle> int_finish(dt);
while( t<maxT ){
  // velocity verlet integration first part:
  for_each_particle(box.begin(),box.end(),int_start);
  // force calculation:
  box.update();                             // communication
  box.for_each_pair(myForce);
  // velocity verlet integration second part:
  for_each_particle(box.begin(),box.end(),int_finish);
  // advance time:
  t+=dt;
}
```

In the STL algorithms and containers are separated as much as possible. However, in order to have an efficient search algorithm for interaction pairs we

had to incorporate it into the container, much like the sort algorithm for lists in the STL. The member function `for_each_pair` performs a pair search with the well known linked cell algorithm. For debugging purposes there exists a `for_each_pair(iterator,iterator,func_object)` that performs a brute force N^2 loop across all pairs.

The integration in the code example above needs two function objects for the velocity verlet integration:

```
template<class T>
class MD_int_vv_start{
public:
  MD_int_vv_start(double adt=1.){dt=adt;};
  inline void operator()(T& p){
    p.x += p.v*dt+ p.f*(0.5*dt*dt); // update position
    p.v += p.f*(0.5*dt);            // update velocity
    p.f=0.;                         // clear force
  }
private:
  double dt;                        // time step
};


template<class T>
class MD_int_vv_finish {
public:
  MD_int_vv_finish(double adt=1.){dt=adt;};
  inline void operator()(P& p){
    p.v += p.f*(.5*dt);             // update velocity
  }
private:
  double dt;                        // time step
};
```

2.1 Latency hiding

To achieve latency hiding it was decided to group the communication in two sections. First the communication for the first dimension is initiated by calls to `MPI_Isend`. In a second stage this communication is completed and the calls for all remaining dimensions are performed, allowing a partial overlap of communication with computation. In the basic version of the template container all communication is done in `update`. Now this call is split into `update_init` and `update_wait` representing the two stages of communication. In this application the force calculation for particles that interact only with particles of the core domain is performed between the first and the second stage. The optimized container provides the calls `for_each_inner_pair` and `for_each_bound_pair` to calculate the pairwise interaction for the corresponding domains.

3 Performance

3.1 Comparison between C and C++

While recent benchmarks have shown that C++ can compete with C or Fortran [3] it is not clear whether this holds for a particular application. Because performance depends on the abstraction techniques used [4, 5] it is a non trivial task to find a suitable balance between certain techniques and performance. Because the force calculation is the most time consuming part in an MD simulation, the danger of an abstraction penalty at this point is high.

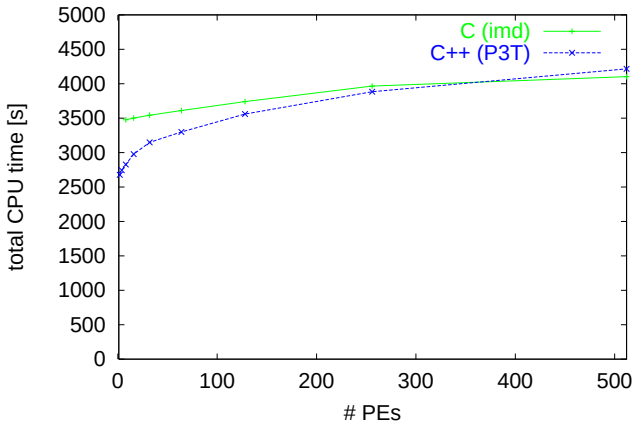

Fig. 3. Total runtime of two different Molecular Dynamics programs depending on the number of PEs.

To get an estimate of the performance penalty of using C++ and its abstraction we made a comparison with a program (imd) written in C that was developed independently [6]. The basic linked cell algorithm is the same. For a simulation of a Lennard-Jones fcc crystal with 442368 atoms the C++ version is between 3 percent slower and 20 percent faster. There are several reasons why the speed up of the C++ program is worse. First, latency hiding is applied and its advantage is lost rapidly with decreasing numbers of particles. Second, Newton's third law is not applied across PE boundaries. This increases the workload on a single PE and results in larger overall runtimes when the cost of the additional communication for exchanging forces is negligible. For large particle numbers the overlapping of communication and computation will also overcome performance problems due to possible congestion of the T3Es network [7].

3.2 Performance between different compilers

While C++ was designed to have only minimum runtime overhead, the actual abstraction penalty does not only depend on the abstraction level used, but also

on the optimization technique of the compiler. Fig. 4 gives an impression of the differences we observed between different compilers. There is a factor of more than three between the best and worst optimizing compiler.

But using the best compiler is not enough. We observed a similar performance penalty when upgrading from KCC 3.2 to KCC 3.3. The reason was that with KCC 3.3 exception handling was turned on by default. Using a compiler switch to disable exception handling we were back to the old performance. This can not be intrinsic to exception handling because `cxx` does show good performance while still providing exception handling. Obviously some important optimizations are affected.

Compiler	flags
SUN CC 4.2	-fast
DEC cxx 6.x	-O5 -tune host -ieee -assume noaccuracy_sensitive
IBM xlC 3.1.4	-O3
Cray CC 3.0.1.3	-O3
KAI KCC 3.2	+K3 -O3
GNU gcc 2.7.2	-O3

Table 1. Optimization flags used for the different compilers.

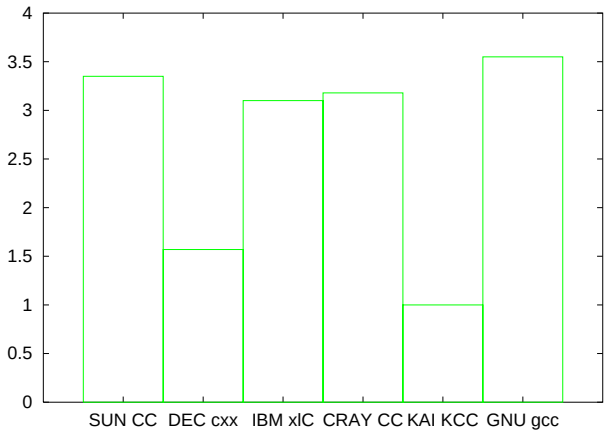

Fig. 4. Runtime of a MD program with different compilers relative to runtime using KAI KCC. The runtime with KAI KCC 3.2x is set to one, because it is available on all platform. The value for GNU gcc is the average from two platforms (Digital Unix (3.4) and SUN Solaris (3.7)).

4 Conclusion

We have shown that object-oriented concepts help to design flexible, portable and easy-to-use tools for molecular dynamics. In particular, judicious application of the template and inlining mechanisms of C++ *can* lead to program performance at par or only slightly worse than that of classical procedural languages. In several cases, the container abstractions described above have proved to yield programs easily portable to parallel platforms and between 2 and 3 dimensions.

As an technical aside we see that the use of optimizing C++ compilers is often severely hampered by the available memory and file transfer rates. In addition, on the software side, compiler vendors only recently support the important new features of C++ specified in the ANSI/ISO standard.

I acknowledge financial support of the SFB 382 and the institute of computer application 1 (ICA1) where this software has been developed. I also would like to thank several colleagues at the ICA1 who take an active part in developing and improving the described software, among others Kai Höfler, Oliver Kitt, Christian Manwart, Reinmar Mück, Gerd Sauermann, Stefan Schwarzer.

References

1. Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.
2. Todd Veldhuizen. Expression templates. *C++ Report*, pages 26–31, June 1995.
3. Todd Veldhuizen. Scientific computing: C++ vs. Fortran. *Dr. Dobb's Journal*, November 1997.
4. Scott Haney. Is C++ fast enough for scientific computing? *Computers in Physics*, 8(6):690–694, Nov/Dec 1994.
5. Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10:458–462, 1996.
6. J. Stadler, R. Mikulla, and H.-R. Trebin. IMD: A software package for molecular dynamics studies on parallel computers. *Int. J. Mod. Phys. C*, 8:1131–1140, 1997.
7. Matthias Müller and Michael Resch. Pe mapping and the congestion problem on the T3E. In Hermann Lederer and Friedrich Hertweck, editors, *Proceedings of the Fourth European Cray-SGI MPP Workshop*, pages 20–28. IPP, Garching, Germany, September 1998. see http://www.rzg.mpg.de/mpp-workshop/proceedings.html.

Simulating and Modeling in Java

Augustin Prodan[1], Florin Gorunescu[2], and Radu Prodan[3]

[1] Iuliu Haţieganu University, Cluj-Napoca, aprodan@umfcluj.ro
[2] University of Medicine and Pharmacy, Craiova, gorun@medinf.comp-craiova.ro
[3] University of Basel, prodan@ifi.unibas.ch

Abstract. The purpose of this paper is to present preliminary results of an experimental work concerning the possibilities offered by Java language for simulating and modeling applications. It describes an object-oriented approach to the simulation of the random variables by means of clasical distributions. The theoretical fundamentals for the things implemented are also given. An application which shows a model for the flow of patients around departments of geriatric medicine is presented. Future research effort will be oriented towards model analysis by use of simulation study, exploratory data analysis, statistical data mining, queuing models and visualizations.

1 Introduction

Previous research has shown that stochastic models are advantageous tools for representation of the real world [6]. Based on theoretical fundamentals in stochastic modeling [5], we intend to make an incremental development of an object-oriented Java framework containing the main elements for building and implementing stochastic models. This is an experimental method developed in the Department of Mathematics and Informatics, Iuliu Haţieganu University Cluj-Napoca, in collaboration with the University of Medicine and Pharmacy Craiova. The models are fitted to data from actual medical and pharmaceutical settings. We incorporated methods for simulating and modeling the flow of patients in the departments of geriatric medicine. The population of geriatrics in a given hospital district is relatively stable and therefore we may model the movement of geriatric patients by considering both their entrance into the hospital and their stay in the geriatric unit. The need for a realistic model to represent the length of stay of geriatric patients in a unit, is recognized by physicians, health care and hospital administrators. However, early models did not consider the difference between acute and chronic care patients and thus the mean time a patient spent in the department was used as a performance indicator. The difference between acute and long-stay care patients led to the development of compartmental deterministic discrete-time model, considering acute and long-stay care as compartments in the geriatric department [3]. While the deterministic approach enables the calculation of means of geriatric patients requiring hospital care, the stochastic model also enables one to extract variances by taking into account patient variability hence allowing the variations inherent in individual

behaviour to be quantified. Once the model has been shown accurately to represent the movement of geriatric patients, given expert knowledge of feasible changes to the system and their associated costs, the theoretical effects of modifications to the system may be calculated and assessed without actually having to implement the changes. Furthermore, the implementation of the respective changes could also prove to be quite costly. It is therefore possible to maximize the efficiency of the geriatric unit, thus allowing for the optimization of the use of hospital resources which can, in turn, improve health care in the hospital.

2 The Fundamentals of Simulation in Java

2.1 The Simulation of Random Numbers in Java

The basis of a simulation study are the random numbers. Most computer languages have a built-in random number generator. The numbers generated this way are not random as they are deterministically computed via some mathematical formulae. This is the reason they are sometimes called pseudo-random numbers. However, such numbers can be used to simulate random numbers, because they have the appearance of being random values uniformly distributed on interval (0,1). Moreover, these numbers can be used to generate random numbers on any interval. For this reason, in this paper we refer to them as random numbers.
Java provides the following facilities to generate random numbers [4]:

The Class java.util.Random – An instance of this class can be used to generate a stream of random numbers based on a linear congruential formula [2].

The Method java.lang.Math.random() – This method is a simpler way of using the one above, by making use of the default constructor. When this method is first called, it creates a single new random number generator, which is then used for all calls to this method and nowhere else. It returns a random number uniformly distributed on interval $[0.0, 1.0)$, in format double precision.

2.2 The Simulation of Classical Distributions

The classical random variables are the most simple stochastic models, so called distributional models, which enter into the composition of other complex models. It stands to reason that it is necessary the development of some techniques to simulate the values of random variables with various distributions. This is likely because they frequently appear in applications.
A hierarchy of classes which models the clasical distributions is proposed. The hierarchy is shown in Fig. 1 using an Inverted Tree Diagram. Each distribution is determined by a set parameters and a distribution function [5]. Based on these elements, a polymorfic method called **simValue()** is specified and specialised by each distribution class in turn, in order to simulate a specific value. An instance

of a particular class can be used to simulate a set of values for the corresponding random variable, by calling the **simValue()** method as many times as needed.

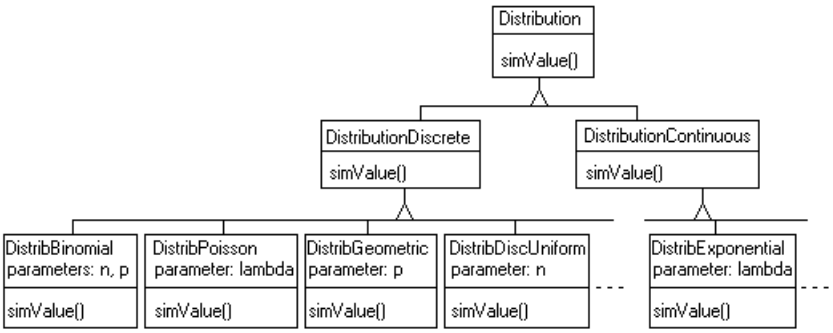

Fig. 1. The Hierarchy of Classes for Clasical Distributions

The particular implementation for each class is based on one or more of the following techniques: the Inverse Transform Technique, the Acceptance-Rejection Technique and the Composition Technique [5].

The Inverse Transform Technique – This technique is based on the following theorem [5]:

Theorem 1. *If U is a uniform (0, 1) random variable, then for any distribution function F, the random variable X defined by $X = F^{-1}(U)$ has distribution F.*

This theorem shows that one can simulate a random variable X with distribution function F by generating a random number U and then setting $X = F^{-1}(U)$. The Inverse Transform Technique is used to simulate a discrete random variable (binomial $BIN(n,p)$), as shown in subsection 3.1, and a continuous random variable (exponential $EXP(\lambda)$), as presented in the subsection 4.1.

The Acceptance-Rejection Technique – An efficient method for simulating a random variable with density function $g(x)$ is important in that it can be used as a basis for simulating another random variable with density function $f(x)$. If c is a constant for which we have $\frac{f(x)}{g(x)} \leq c$, $\forall x$ such that $g(x) \neq 0$, the acceptance-rejection algorithm for simulating the random variable with the density function $f(x)$ can be expressed as follows [5]:

1. Simulate a value for random variable Y having density function $g(x)$;
2. Simulate a random number U;
3. If $U \leq \frac{f(Y)}{cg(Y)}$, set $X = Y$, otherwise go to step 1.

The following theorem asserts that the previous algorithm works well [5]:

Theorem 2. *The random variable X simulated by acceptance-rejection technique has the density function $f(x)$. The number of iterations that are needed to simulate a value is a geometric random variable with mean c.*

The subsection 4.2 shows how acceptance-rejection technique is used to simulate the normal random variable.

The Composition Technique – If F_i, $i = 1, 2, ..., n$ are distribution functions and ρ_i, $i = 1, 2, ..., n$ are nonnegative numbers so that $\sum_{i=1}^{k} \rho_i = 1$ then the distribution function F given by $F(x) = \sum_{i=1}^{n} \rho_i F_i(x)$ is said to be a *composition*, or *mixture*, of the distribution functions F_i, $i = 1, 2, ..., n$. A method to simulate from F is to simulate a random variable I, so that $p_i = P\{I = i\} = \rho_i$, $i = 1, 2, ..., n$ and then to simulate from F_I. That means, if the simulated value of I is $I = k$, then the second simulation is from F_k. The composition technique is presented in subsection 5.3.

3 The Simulation of Discrete Random Variables

This section describes the implementation of the Inverse Transform Technique to simulate a random variable using the binomial distribution. This is represented by the class **DistribBinomial**. In a similar manner, classes for the Poisson, geometric and discrete uniform distributions can be designed and implemented.

3.1 Binomial Random Variable

A binomial random variable X with parameters n and p has the probability mass function: $p_i = P\{X = i\} = C_n^i p^i (1-p)^{n-i}$, $i = 0, 1, 2, ..., n$. The key of using the inverse transform technique to simulate such a random variable is the following recursive identity:

$$P\{X = i+1\} = \frac{n-i}{i+1} \frac{p}{1-p} P\{X = i\}$$

With i denoting the value currently under consideration, $pp = P\{X = i\}$ and $F = F(i) = P\{X \leq i\}$, the algorithm can be expressed as follows:

1. Simulate a random number U;
2. Initializations: $rap = \frac{p}{1-p}$, $i = 0$, $pp = (1-p)^n$, $F = pp$;
3. If $U < F$, set $X = i$ and stop;
4. Recursion: $pp = rap \frac{n-i}{i+1} pp$, $F = F + pp$, $i = i + 1$, then go to step 3.

The Java algorithm which implements the method **simValue()** inside the class **DistribBinomial**, is shown in Fig. 2.

The algorithm can be improved upon when the mean np is large. Since a binomial random variable with mean np is most likely to take on one of the values closest to np, a more efficient algorithm would first check one of these values, then search downward in the case where $X \leq np$, or upward otherwise.

```
public class DistribBinomial extends DistributionDiscrete {
    int n;    // number of trials.
    double p; // probability of success.
    . . .
    public double simValue() { // Simulate a value
        double U, rap, pp, F, i=0;
        U = Math.random();
        rap = p/(1-p);
        pp = Math.exp(n*Math.log(1-p)); // P{X=0}
        F =  pp; // F(0) = P{X=0}
        while (U > F) {
            pp = (rap*(n-i)/(i+1))*pp; // recursion
            F = F + pp; // F(i)=P{X<=i} distribution function
            i++;
        }
        return i;// Return the simulated value
    } // simValue()
} //
```

Fig. 2. The Class DistribBinomial

3.2 Simulated versus Theoretical Values

The way to verify the accuracy of a particular **simValue()** method is to compare the simulated results with theoretical ones. A Java applet provides this facility in the case of discrete distributions. An instance of a discrete distribution class is used to generate a sequence of N independent values for the corresponding random variable. The **doSimulation()** method, shown in Fig. 3, serves this purpose. In Java all method bindings happen polymorphically via late binding,

```
void doSimulation(DistributionDiscrete dis) {
    for (int k=0; k<N; k++) {d[(int)dis.simValue()]++;}
}
```

Fig. 3. The Simulation and Counting of N Values for a Discrete Distribution

so the method **simValue()** can talk to the base class **DistributionDiscrete**, and all derived-class cases will work correctly using the same code. The generated values for a random variable X are stored in the array $d[X]$. With a visualization method, the elements of this array are expressed graphically in a column format, as shown in the left graphs of the Fig. 4, 5, 6 and 7. The right graphs of the same figures are columns representing the corresponding theoretical values for the same distributions, proportionally with probabilities p_i, $i > 0$. By selecting various discrete distributions while the applet is running, one can compare the simulated results with the theoretical values.

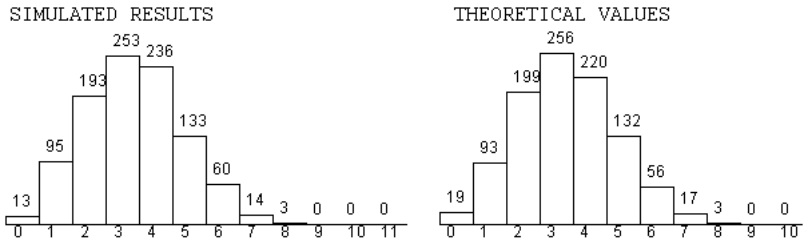

Fig. 4. Binomial Distribution BIN(11, 0.3), for N=1000 Values

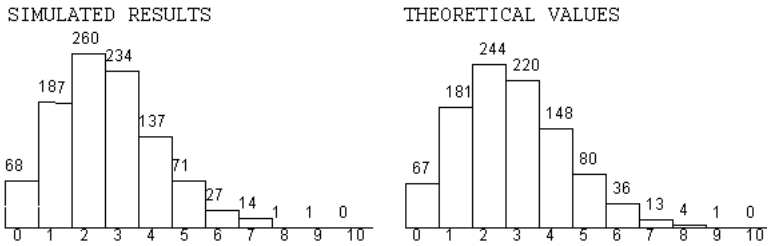

Fig. 5. Poisson Distribution POI(2.7), for N=1000 Values

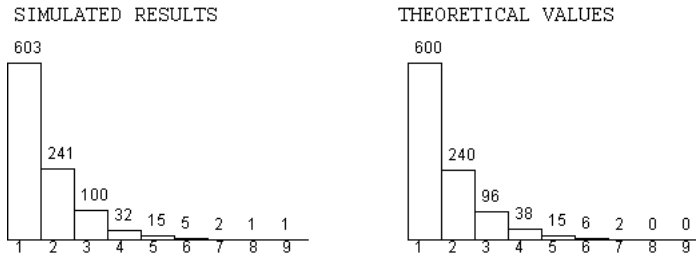

Fig. 6. Geometric Distribution GEO(0.6), for N=1000 Values

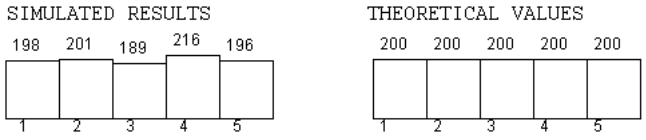

Fig. 7. Discrete Uniform Distribution DU(5), for N=1000 Values

4 The Simulation of Continuous Random Variables

4.1 Exponential Random Variable

An exponential random variable with rate $\lambda > 0$ (mean $\frac{1}{\lambda}$) has the probability density function $f(x) = \lambda e^{-x}$, and the distribution function $F(x) = 1 - e^{-x}$, for $0 < x < \infty$. If we let $u = F(x) = 1 - e^{-x}$, it is easy to verify that $x = F^{-1}(u) = -\frac{1}{\lambda}\ln(1-u)$. Noting that if U is uniform on (0, 1), then $1 - U$ is also uniform on (0, 1), and the inverse transform algorithm for simulating an exponential random variable with parameter λ can be expressed as follows:

1. Simulate a random number U;
2. Set $X = -\frac{1}{\lambda}\ln(U)$.

This algorithm is implemented by the method **simValue()** inside the class **DistribExponential** (Fig. 8).

```
public class DistribExponential extends DistributionContinuous {
    double lambda; // parameter for exponential distribution(rate)
    . . .
    public double simValue() { // Simulate a value
        double U = Math.random();
        return  -1/lambda*Math.log(U);
    } // simValue()
} ///;
```

Fig. 8. The Class DistribExponential for Exponential Random Variable

4.2 Normal Random Variable

A unit normal random variable Z(with mean 0 and variance 1) has the probability density function $f(x) = \sqrt{\frac{2}{\pi}}e^{-\frac{x^2}{2}}$, $0 < x < \infty$. To simulate a value for Z we apply the acceptance-rejection technique using as $g(x)$ the exponential density function with $\lambda = 1$, that is $g(x) = e^{-x}$, $0 < x < \infty$. It is easy to verify that maximum value of $\frac{f(x)}{g(x)}$ occurs when $x = 1$, so we can take $c = \max\frac{f(x)}{g(x)} = \frac{f(1)}{g(1)} = \sqrt{2e\pi}$. Because $\frac{f(x)}{cg(x)} = e^{x-\frac{x^2}{2}-\frac{1}{2}} = e^{-\frac{(x-1)^2}{2}}$, the acceptance-rejection algorithm for simulating $X = |Z|$ (the absolute value of a unit normal) can be expressed as follows [5]:

1. Simulate Y, an exponential random variable with parameter $\lambda = 1$;
2. Simulate a random number U;
3. If $U \leq e^{-\frac{(Y-1)^2}{2}}$ set $X = Y$, otherwise go to step 1.

As the density $f(x)$ is an even function, the previous algorithm simulate the absolute value of a unit normal distribution. To obtain a unit normal distribution we set Z be equally likely to be either X or $-X$.

5 A Model for Geriatric Medicine

5.1 Generalities

The population of geriatrics in a given hospital district is relatively stable and therefore we may model the movement of geriatric patients by considering both their entrance into the hospital and their stay in the geriatric unit. Admissions are modeled as a Poisson process with parameter λ (the arrival rate), estimated by using the observed inter-arrival times. Previous research has shown that the flow of patients around compartments of geriatric medicine may be modeled by the application of a mixed-exponential distribution, where the number of terms in the mixture corresponds to the number of stages of patient care. A common scenario is that there are two stages for in-patient care: *acute* and *long-stay*. The difference between acute and long-stay patients has led to the development of a compartmental deterministic discrete-time model, considering acute and long-stay care as compartments in the geriatric unit. The use of stochastic compartmental analysis [6], which assumes probabilistic behaviour of the patients around the system, is considered a more realistic representation of an actual situation rather than the simpler deterministic model. In order to simulate the model, we have split it into two parts: the Poisson arrivals and the in-patient care.

5.2 Simulating Admission as a Poisson Stream in Continuous Time

The arrival of patients is modeled by a Poisson process at rate λ= 2.75 patients per month with the corresponding density $f(x) = \lambda e^{-\lambda x}$, $x \geq 0$. Incorporating the results provides one way in which to simulate the inter-arrival times. This is accomplished by examining the times between events for such a process. These times are independent exponential random variables, each with rate λ. For this purpose we use instances of the class **DistribExponential**, described in subsection 4.1. To simulate the first T time units of a Poisson process, it is necessary to simulate the successive inter-arrival times, stopping when their sum exceeds T. A Java applet called **arrivalsPoisson**, which incorporates the statements presented in Fig. 9, performs a graphical simulation of monthly arrivals is used the. The number of arrivals occurring in each month are counted in corresponding elements of the array $P[\,]$. Running this applet with parameter $T = 12$ (one year), one can visualise monthly arrivals for a period of one year, the results bearing the semblance of those in Fig. 10.

5.3 Simulating the In-Patient Care Time

The in-patient care – *acute* and *long-stay* – is modeled as a mixed-exponential phase-type distribution, using the composition technique (see subsection 2.2) with the density $f(x) = \rho\alpha_1 e^{-\alpha_1 x} + (1-\rho)\alpha_2 e^{-\alpha_2 x}$, $x \geq 0$, where $\rho = 0.93$, $\alpha_1 = 0.04$ and $\alpha_2 = 0.001$, which implies a mean care time of $\frac{\rho}{\alpha_1} + \frac{1-\rho}{\alpha_2} \approx 93.25$ days per patient. Fig. 11 shows how the *composition of classes* paradigm [1] is

```
double t=0, T=12, lambda=2.75;
DistribExponential de = new DistribExponential(lambda);
...
for (int k=0; t<(double)T; k++) {
    t = t + de.simValue(); // increment with inter-arrival time
    P[(int)t]++; // Increment patients number for corresp. month.
}
```

Fig. 9. The Simulation of a Poisson Process with Rate $\lambda = 2.75$

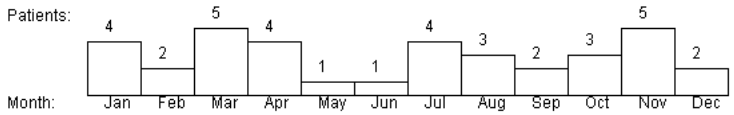

Fig. 10. The Poisson Arrivals at Rate $\lambda = 2.75$ per Month, for One Year

used to implement the composition algorithm for simulating the in-patient care time.

```
double t=0, U, alfa1=0.04, alfa2=0.001, ro=0.93;
DistribExponential e1 = new DistribExponential(alfa1);
DistribExponential e2 = new DistribExponential(alfa2);
...
for (int k=0; k<n; k++) {
    U = Math.random();
    if (U<ro) {t = e1.simValue();}
         else {t = e2.simValue();}
    S[k] = (int)t; // care time for patient k (in days)
}
```

Fig. 11. The Simulation of Care Time as a Mixed Exponential Distribution

The n variable stores the number of patients. The array $S[\,]$ is used to register the care time in days for patients. The care time for each patient is counted in the corresponding element of the array $S[\,]$. Running the sequence presented in Fig. 11 with parameter $n = 15$, the results appear as in Fig. 12.
Once the model has been shown to accurately represent the movement of geriatric patients, given expert knowledge of feasible changes to the system and the costs associated, the theoretical effects of modifications to the system may be calculated. Furthermore, the costs can be assessed without having to implement the changes in a real setting, which could be costly. Therefore, it is possible to maximize the efficiency of the geriatric department, thus optimizing the use of hospital resources, in order to improve hospital care.

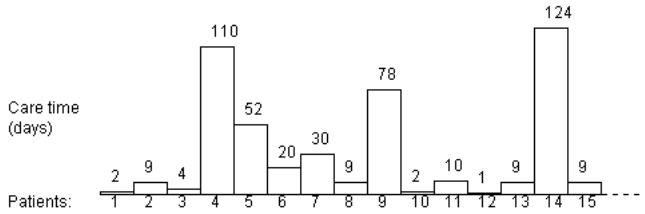

Fig. 12. The Simulated Results for In-Patient Geriatric Care Time

6 Conclusions

The paper describes some results obtained in research concerning the possibilities offered by Java language for simulating and modeling applications. A set of base line classes for simulation of classical distributions were introduced. Based on their design and implementation, a model was created which accurately represents the movement of geriatric patients. Both geriatricians and hospital administrators agreed that such a model can be used to maximize the efficiency of the geriatric department and to optimize the use of hospital resources in order to improve hospital care. The two simulations presented in section 5 need to be integrated within two threads inside the same application process. We intend to continue the incremental development with new classes for implementing simulation and model techniques. A model for estimating tumor sizes by using the Hit or Miss Monte Carlo integration will be created. Future research effort will be also oriented towards the development of new models, model analysis, statistical data mining, and visualisations.

References

1. Bruce Eckel. *Thinking in Java.* President, MindView Inc., Prentice Hall PTR, 1998.
2. Donald E. Knuth. *The Art of Computer Programming, vol 2.* Addison-Wesley Publishing Company, 1973.
3. G. Harrison, P. Millard *Balancing acute and long-term care: the maths of throughput in departments of geriatric medicine,* Math. Inform. Med., 30, 221-228, 1991.
4. Augustin Prodan, Mihai Prodan. *Mediul Java pentru Internet.* Ed. ProMedia-plus, Cluj-Napoca, 1997.
5. Sheldon M. Ross. *A Course in Simulation.* Macmillan Publishing Company, New York, 1990.
6. G. Taylor, S. McClean, P. Millard. *Continuous-time Markov models for geriatric patient behaviour.* Appl. Stochastic Models Data Anal, 13, 315-323, 1998.