SPH2000: A Parallel Object-Oriented Framework for Particle Simulations with SPH*

Sven Ganzenmüller, Simon Pinkenburg, and Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik
Department of Computer Engineering
University of Tübingen
Sand 13, 72076 Tübingen, Germany
{ganzenmu,pinkenbu,rosen}@informatik.uni-tuebingen.de

Abstract. A widespread method in parallel scientific computing is SPH, a grid-free method for particle simulations. Lots of libraries implementing this method evolved in the past. Since most of them are written in FORTRAN or C, there is a lack of integration of object-oriented concepts for scientific applications. These libraries are therefore hard to maintain and to extend. In this paper, we describe the design and implementation of sph2000, a parallel object-oriented framework for particle simulations written in C++. Its key features are easy configurability and good extensibility for the users to support their ongoing development of the SPH method. The use of design patterns lead to an efficient and clear design and the implementation of parallel I/O improved the performance significantly. A sample application was implemented to test the framework.

1 Introduction

Within the Collaborative Research Center (CRC) 382 physicists, mathematicians and computer scientists work together to research new aspects of astrophysics and the motion of multiphase flows, evolve them to models and run parameter studies to verify these models. Several particle codes are used to simulate these physical problems.

A well known particle simulation method is Smoothed Particle Hydrodynamics (SPH). SPH is a grid-free Lagrangian method for solving the hydrodynamic equations for compressible and viscous fluids. It was introduced in 1977 by [4] and [7] and has become a widely used numerical method for astrophysical problems. Nowadays the SPH approach is also used in fields of material sciences, for modeling multiphase flows [9] and the simulation of brittle solids [2].

Resolution and accuracy of a simulation depend on the number of used particles and interaction partners. Actual physical problems need large numbers to achieve reasonable results. Thus, high-performance computers are indispensable.

* This project is funded by the DFG within CRC 382: *Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern* (Methods and algorithms to simulate physical processes on supercomputers).

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 1275–1284, 2005.

As a result, most users in the domain of physical simulations developed self-made parallel scientific libraries. Although object-oriented programming became most recently state-of-the-art in parallel programming, almost all libraries are still implemented in FORTRAN or C, leading to a lack of integration of object-oriented concepts for parallel scientific applications.

Our group has a strong effort to develop fast parallel particle libraries, which are portable to all important parallel platforms. Therefore, we developed a parallel object-oriented SPH framework which is clearly structured, easy to configure, maintain and extendable by the advantages of object-oriented programming and design patterns. The main goals were to provide a general framework for SPH-like particle simulations and to reduce the parallel overhead and serial parts, which limit the overall speedup. To apply object-orientation to parallelization, we use TPO++, an object-oriented communication library set up on top of MPI [5]. It was developed in our working group and provides the same functionality and efficiency as MPI 1.2. Recently, we extended it by an object-oriented interface for parallel I/O. A sample application points up the usability of our approach.

This paper is organized as follows. In Section 2, we present related work. Section 3 contains the design and implementation of the framework. In Section 4 we present a sample application and in Section 5 we conclude.

2 Related Work

Cactus [1] is a large framework for parallel physical simulations with many modules, called thorns, which offers interfaces for different languages, e.g. for C++. POOMA [11] is as well a wide-spread framework for parallel physical computations. Both libraries, like most others, are laid out for grid methods. Although POOMA supports moving particles, they have to be arranged in arrays. The focus is on interactions and transformations between the particle arrays and the grid fields.

Besides other SPH libraries [6], Gadget [12] is a large SPH library which offers standard algorithms for astrophysics with self-gravitation. However, like most SPH libraries, it is not object-oriented.

The embedding of pure particle methods in object-oriented libraries is not very common. To use the given object-oriented libraries, the developer of particle methods must take along the overhead of grid methods, deal with a higher complexity and learn the concepts of the library, although not every concept is needed. Using the procedural SPH libraries means to abandon the advantages of object-oriented programming.

3 Design and Implementation

3.1 Design Goals

Our main goal was to develop a parallel object-oriented SPH-framework with extensibility, maintainability and reusability of the code. A main concern in the

design was the strict decoupling of parallelization and physics. Another goal was to prove the feasibility of the object-oriented approach in the performance critical domain of particle simulations without loosing efficiency. The result is a parallel object-oriented particle simulation framework written in C++, called sph2000. Classes modeling the elements of the problem domain generate a well structured design. The use of several design patterns [3] helped to organize the classes clearly and efficient. They are introduced to structure the class library, to separate and group the application elements, as well as to define uniform interfaces. Additionally, they allow to insert extensions more simply because of decoupled elements. Table 1 shows the application elements and its corresponding design patterns.

Table 1. Application elements and corresponding design patterns.

Element	Responsibility	Design Pattern
Initialization	Configuration, Object Creation	Builder, Configuration Table
Mathematics	Time Integration	Strategy, Iterator, Index Table
Physics	Particles, Interactions, Right Hand Side (RHS)	Strategy, Compositum, Iterator, Index Table
Parallelization	Communication, Decomposition, Load Balancing	Strategy, Mediator, Proxy
Geometry	Simulation Domain, Subdomains	Strategy, Decorator

The independent elements can be extended, causing no changes in other classes. The classes within an element can be easily exchanged because of uniform interfaces. Once implemented this enables the user to form new classes by simply copying and adapting the existing ones. Thus, extensions supplement the code instead of changing it.

3.2 Configuration of Simulation Runs

Another goal was to simplify the configuration of simulations, which mainly means to configure a simulation run after the compilation at runtime by reading a parameter file. To avoid conditional compilation with preprocessor directives, as it is often seen in C libraries, the Strategy pattern, which is based on the object-oriented concept of polymorphism is used. With this pattern the program instantiates objects at runtime due to the configuration parameters.

The complete configuration with all parameters of a simulation is stored in an object of the class ParameterMap (Configuration Table pattern). Every object which needs parameters owns a reference to the ParameterMap object. Thus all objects can access the configuration uncomplicatedly. Mainly the initialization objects (Builder pattern) access the ParameterMap to realize the exchangeability of the components. Every Strategy offers an accordant parameter to determine which concrete implementation must be used in the simulation. In every simulation run, the Builders create and initialize only the needed objects.

The concept of the configuration table makes the configuration data available for all auxiliary programs which work with simulation data. Such supplementary tools adopt the existing ParameterMap class without changes.

3.3 A Quantity Index Store

The flexibility of the framework is mainly up to the used physics. Because of enhancing the framework simultaneously with the physical method, the integration of additional physical quantities and the exchangeability of different calculation methods has to be guaranteed.

An SPH particle is a sampling point of the differential equations which moves with the flow and represents a volume element of the moving fluid. It contains all physical quantities of a fluid element, their interactions between each other have to be evaluated and they have to be communicated among the subdomains.

To protect the user from adapting the Particle class for every application, we introduced the class IdStore as an index table. The Particle class contains only the administrative structure for the communication by message passing and two STL containers for the scalar and the vector variables. The initialization determines the fixed number of needed variables within a simulation and writes them into the index table. Thus the particles can adapt themselves dynamically at start time to the respective simulation.

The basis for this design are two classes, the QuantityBuilder and the IdStore. The QuantityBuilder implements references to the used physics and initializes the IdStore object. It evaluates the parameters from the ParameterMap and reserves an index in the IdStore for every physical variable (e.g. position, speed and density). The physical variables are stored inside the particle in the order of these indices. Since the particle itself has no idea about the contained variables, classes needing a particles' variable have to get the information through the IdStore, which represents the interface to the variables.

3.4 Object-Oriented View of the Right Hand Side

The QuantityBuilder class is designed according to the Builder pattern. Besides the initialization of the IdStore it establishes the QuantityList, an STL container of calculation objects for the physical quantities. Since there are no general SPH formulas for the equations of motion, many different approaches evolved in the past. To achieve a high flexibility, the calculation objects are defined as a Strategy with a Quantity base class.

The QuantityBuilder knows all possible quantities and their dependencies. By reading the physical parameters from the ParameterMap, quantities are selected and stored in the QuantityList in respect to the physical dependencies (see Fig. 1). In each time step an RhsCalculator object iterates through the QuantityList to compute the right hand side (RHS) of the differential equations.

Like most elements of the framework, the Integrator class is also implemented as a Strategy and thus configurable and extendable like the quantity classes. The several Runge-Kutta and adaptive integrators of the framework are based on an

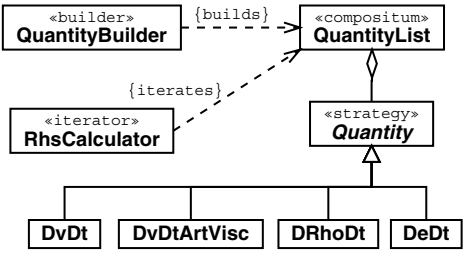

Fig. 1. Simplified class diagram of the sph2000 calculation classes.

abstract Integrator class. It defines interfaces to iterate through all particles' differential variables to integrate the right hand side and to store intermediate integration steps. The Strategy pattern is very easy to apply, since the user only has to write a configuration file, e.g. which Integrator should be utilized, and the Builders only create the needed objects of each Strategy.

3.5 Parallelization and Domain Decomposition

Geometrical Point of View. For an efficient parallelization we implemented a domain decomposition, dividing up the simulation area into several rectangular subdomains. These are equally spaced at the beginning of the simulation but dynamically change their size during runtime to keep the load balanced between all processors. The size is thereby given by the number of interaction partners, since this linear effects the calculation time.

A class SubSimulation was implemented, which defines the base methods of a subdomain like administrating the geometry, the adjacent domains and the particles within the subdomain. It is based on the Strategy pattern to be able to differentiate between subdomains with different tasks. The framework knows two specialized types of SubSimulations. The BoundarySimulation extends the SubSimulation by methods for reflecting and absorbing particles at the boundaries. The InletSimulation, based on the Decorator pattern, can decorate the latter with additional methods for inserting new particles to the simulation through an inlet. The ParameterMap includes the initial and behavioral values of these three types.

Domain Decomposition by Grouping Objects. In every time step each subdomain has to process the same tasks, like preparing the calculation, communicating particles to other subdomains, computing lists which contain the interaction partners, calculating the right hand side, or integrating the equations of motion. Each subdomain therefore contains classes and objects respectively for these tasks. From an object-oriented view each subdomain is a group of objects, which represent this area geometrically. The communication and application flow within a group is implemented using an *intra node* Mediator, which

knows all contained objects, coordinates the chronological processing of the tasks and uncouples all objects within a group from each other.

To communicate particles and other information to and from the adjacent subdomains, an *inter node* Mediator is needed. This Communicator encapsulates the information about the whole domain and communication structure. The implementation follows the design patterns Strategy and Mediator. The class BaseCommunicator defines the interface between intra node and inter node communication (see Fig. 2).

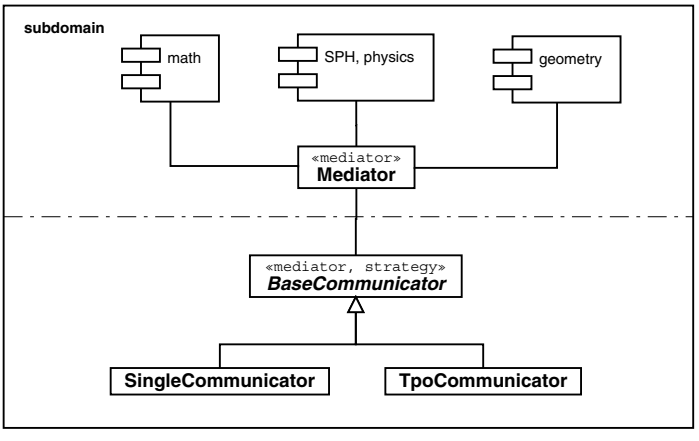

Fig. 2. Simplified diagram of a subdomain in sph2000. The upper part shows the modules with the SPH classes and the intra node mediator. The lower part shows the class diagram for the communication strategy. The Communicator classes encapsulate the whole inter node communication. The calculation objects are completely decoupled from the parallelization.

The major advantage of this concept is, that it enables the user to easily divide up the simulation domain into as many subdomains as processors are available. For communication between the processors the message passing object TpoCommunicator is generated, which uses TPO++. In case of a single processor simulation this communicator only has to be replaced by a single-node communicator. The user only has to exchange the communicators in the configuration file, leaving the code unchanged.

To coordinate all subdomains a special master subdomain is implemented, which is extended by several administration objects. These are objects for time-keeping and administrating all particles as well as particle-I/O objects for saving and restoring particle allocations.

3.6 Parallel I/O

The first results of sph2000 showed a significant lack in performance due to sequential I/O. Since current standards like MPI 2 [8] only support procedural

interfaces for parallel I/O, we extended TPO++ by an object-oriented interface for parallel I/O [10].

The initial version of sph2000 implements an I/O strategy with one master process gathering the part results from all other processes and saving the whole data in ASCII format to disk. The new strategy using the parallel I/O interface was to implement collective I/O. Thereby, all processes can access the same file in parallel, which improves the performance significantly, since the communication to the master process is needless and the whole parallel I/O bandwidth can be used for transferring the data to and from disk. In addition, the library internally calculates the correct offsets within the file where each process has to place its part, avoiding any extra implementation by the user.

To provide sph2000 with parallel I/O, the particle-I/O class of the framework had to be adapted. The following listing shows the adapted method `saveDataFile` and represents the simple usage of the interface:

```
#include<tpo++.H>

void ParticleIO::saveDataFile(const ParticleContainer& particles, string name)
{
    TPO::File fh;
    int code = fh.open(TPO::CommWorld, name, TPO_MODE_CREATE);
    fh.write_all(particles.begin(), particles.end());
    fh.close();
}
```

This implementation of using a single collective call (`fh.write_all`) reduces the size of the original code by about 100 lines of code. The call is needed to save the containers of particle objects of each processor to disk simultaneously. Two iterators `begin()` and `end()` thereby define the beginning and ending of the container. This syntax also enables the user to store only a sub-set of particle objects to disk.

The performance improvement through the usage of parallel I/O within the framework depends on the application as well as on the ratio between the proportion of computation and I/O within the application. To determine the real performance gain we inserted it in our sample application, which is presented in the next section.

4 Sample Application

A first sample application was implemented to test the whole framework. It simulates the injection of diesel into an air filled chamber. Diesel engine manufactures are interested in an optimal injection of the diesel into the combustion chamber. A perfect mixing of diesel and air means an efficient use of the fuel and therefore reduces emission. For this reason the breakup of the diesel jet must be examined and understood. When injected into the cylinder of an engine, the diesel jet undergoes two stages of breakup. In the primary breakup large drops and filaments split off the compact jet. These turn into a spray of droplets during the secondary breakup. This secondary process is well known and can be modeled as a spray, but the understanding of the primary breakup is only in the

initial stages. The physical effects that might influence the primary breakup are the pressure forces in the interface region of diesel and air, instabilities of the jet induced by cavitation inside the injection nozzle, surface tension and turbulence. In this area SPH simulations are not very common. There are several problems concerning the physics of multiphase flows and the requirements in terms of compute power are very high. Due to its extensibility sph2000 is very applicable for this area and enables the user to easily and fast implement new physical concepts. So far the framework provides 5 kernel functions, 6 integrators, and 20 quantities to calculate the state equations and the equations of motion for the air and diesel particles.

Fig. 3. 3D simulation of diesel injection with 2.5 million particles. The picture shows the injected diesel. First drops are already split off the jet. The injection causes a density wave traveling in front of the liquid stream.

2D and 3D simulations with up to 2.5 million particles reveal a broadening and breakup of the diesel jet leading to turbulences behind the dispartment. After a while single drops are separated from the compact jet, see Fig. 3.

The performance of the sample application was conducted on Kepler [13], a self-made clustered supercomputer based on commodity hardware. It consists of two parts: An older part with 96 dual Pentium III (650 MHz) nodes with 1 GB of memory, and a newer part with 32 dual AMD Athlon (1.667 GHz) nodes, each sharing 2 GB of memory. We measured two different simulation setups: The first running on the Pentium nodes with disabled I/O and the second on the Athlons with I/O enabled in every second time step, to compare the performance of sequential and parallel I/O. We always used only one processor per node.

Table 2 shows the performance results of the first setup. Due to memory shortage of the Pentium nodes, the measurements were made starting with 2 nodes. The parallelization scales very good up to 64 processors leading to a remarkable speedup of 44.10.

Table 2. Results of the first simulation setup with 1 million particles on Pentium III.

Processors	1	2	4	8	16	24	32	64
Time per step (in s)	-	116.9	58.8	33.8	16.4	11.3	9.7	5.3
Speedup	-	2.00	3.98	6.90	14.24	20.68	24.10	44.10

The results of the second simulation setup show the significant effect of parallel I/O on the performance (see Table 3). The I/O part could be improved by a factor of 20 when using 32 processors working on a parallel file system with 32 distributed disks. Since I/O is only a small proportion of the whole simulation, the overall gain using parallel I/O reduces to a - still remarkable - factor of 3.5 (64 processors). Note that even the sequential simulation with parallel I/O is faster than without parallel I/O, due to changing from ASCII file format to binary format and less code overhead for saving the particles.

Table 3. Results of the second simulation setup with 1 million particles on Athlon.

Processors	1	2	4	8	16	24	32	64
Execution time (in s)								
- sequential I/O	2090.2	1050.9	530.4	407.0	349.4	305.1	272.4	251.6
- parallel I/O	1223.5	617.2	312.5	191.4	130.9	96.5	79.8	71.3
Speedup								
- sequential I/O	1	1.98	3.94	5.13	5.98	6.85	7.67	8.30
- parallel I/O	1	1.98	3.91	6.39	9.34	12.67	15.33	17.15

5 Conclusion and Future Work

The application of object-oriented development methods has improved the quality of our simulation codes. The implementation is very easy to maintain and extend, e.g. to add the physics of surface tension or turbulence. The result of object-oriented techniques with design patterns is a framework, in which classes have clear and strictly separated responsibilities. Different methods can be interchanged without influencing other code. The use of our parallel I/O library and optimizations for communication reduced the sequential parts of the framework to a minimum. These lead to a well scaling parallel performance.

In the future, we focus on the development of models for simulating surface tension and turbulence. Due to an increased number of particles which is needed to simulate these effects meaningful, we are investigating solutions to decrease the amount of calculated interactions without increasing the runtime of the simulations. Since optimizing the communication is not sufficient, we furthermore try to exclude less important calculations: First, we try to separate the density wave of the injected diesel from the outer air, and second, the air particles shall be generated during runtime according to the motion of the jet. The idea is to calculate only air regions, which are affected by the diesel jet. Both leads to notedly less calculation overhead.

References

1. G. Allen, T. Goodale, E. Seidel. The Cactus Computational Collaboratory: Enabling technologies for relativistic astrophysics, and a toolkit for solving pdes by communities in science and engineering. In *7th Symposium on the Frontiers of Massively Parallel Computation-Frontiers 99*, New York, 1999. IEEE
2. W. Benz, E. Asphaug. Catastrophic Disruptions Revisited. In *Icarus*, 142: 5–20, 1999
3. Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
4. R. A. Gingold, J. J. Monaghan. Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars. In *Monthly Notices of the Royal Astronomical Society*, 181: 375–389, 1977
5. T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An object-oriented message-passing library in C++. In *Proceedings of the 2000 International Conference on Parallel Processing*, pages 43–50. IEEE Computer society, 2000.
6. S. Kunze, E. Schnetter, R. Speith. Development and Astrophysical Applications of a Parallel Smoothed Particle Hydrodynamics Code with MPI. In *High Performance Computing in Science and Engineering '99*, E. Krause, W. Jäger (ed.), Springer, p. 52 – 61, 2000.
7. Leon B. Lucy. A Numerical Approach to Testing the Fission Hypothesis. In *The Astronomical Journal*, 82(12): 1013–1024, 1977
8. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. *Online. URL: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*, July 1997.
9. F. Ott, E. Schnetter. A modified SPH approach for fluids with large density differences. In *ArXiv Physics e-prints*, 3112-+, 2003
10. S. Pinkenburg and W. Rosenstiel. Parallel I/O in an Object-Oriented Message-Passing Library. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, 2004.
11. J. V. W. Reynders, J. C. Cummings, P. J. Hinker, M. Tholburn, M. S. S. Banerjee, S. Karmesin, S. Atlas, K. Keahey, W. F. Humphrey. Pooma: A framework for scientific computing applications on parallel architectures. In *Parallel Programming using C++*, MIT Press, 1996; http://acts.nersc.gov/pooma/
12. V. Springel, N. Yoshida, S. D. M. White. GADGET: A Code for Collisionless and Gasdynamical Cosmological Simulations. In *New Astronomy*, 6(3): 51, 2001.
13. University of Tübingen. Kepler Cluster website. *Online, URL: http://kepler.sfb382-zdv.uni-tuebingen.de/kepler/start.shtml*, 2001.