Parallelization of an Object-Oriented Particle-in-Cell Simulation*

Simon Pinkenburg, Marcus Ritt, and Wolfgang Rosenstiel

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

Abstract. We describe our experience made in parallelizing a Particle-in-Cell simulation. The project was part of our efforts to apply object-oriented methodologies to the development of parallel physical simulations. Unlike earlier projects, which were developed and parallelized in cooperation with physicists, the goal was to parallelize a sequential simulation code written in C++ without having support from its developers. Our interest was to analyze the structure of the original code and the possibilities ofadding the parallelization after sequential development. Also, since the parallelization was targeted to distributed memory architectures, we wanted to test the deployment of the object-oriented message-passing library developed in our working group.
Based on a static and dynamic analysis, we describe several general parallelization strategies and the implementation of one of them. We give a introduction to our message-passing library and detail its extension to collective communication, which was necessary to implement the parallel algorithm. Runtime measurements made on two different architectures are compared. We conclude with a discussion of the findings made in course of the project.

Keywords: Object-orientation, Message-Passing, Simulation, Particle-in-Cell

1 Introduction

This work is part of a government funded collaboration of physicists, mathematicians and computer scientists to develop large-scale physical simulations for massive parallel computers. Our group is concerned with the development of adequate runtime environments and libraries to parallelize these simulations effectively.

While in industry object-oriented techniques and programming languages are widely used, the scientific computing community still does not employ them in

* This project is funded by the Deutsche Forschungsgemeinschaft as a part of the Collaborative Research Center 382 (Methods and Algorithms for the Simulation of Physical Processes on High Performance Computers)

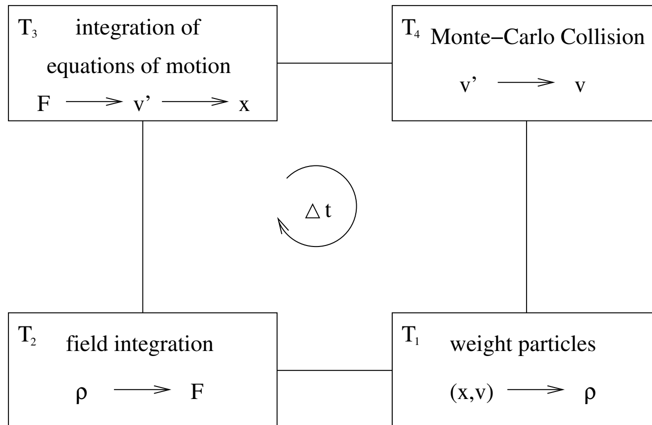

Fig. 1. Steps in a Particle-in-Cell simulation with MCC

the majority of the applications. In recent years, some efforts to enable object-orientation for parallel computing have been made, mainly in the C++ community. Efficient and standard-conforming compilers, mathematical base libraries reduced the performance gap between classical approaches and object-oriented codes [14]. Parallel programming standards like the Message-passing interface provide support for object-oriented languages and efforts to establish object-oriented frameworks for parallel computing are ongoing [15, 8, 11].

In this paper, we investigate how object-oriented methods can help in the parallelization of object-oriented codes. A focus is the reusability of the object-oriented design of a sequential code. In section 2 we give a overview over the Particle-in-Cell method used in this application. We explore parallelization strategies in section 3 and discuss their implementation in the next section. In section 4, we also present an object-oriented message-passing library supporting collective communications, which helped to parallelize the application on distributed memory machine at a reasonable level of abstraction. Measurements of the parallel execution are presented and discussed in section 5. We conclude in section 6.

2 Particle-in-Cell simulation with Monte-Carlo collisions

In a Particle-in-Cell simulation, the medium under consideration is represented by a large number of macro particles, each describing the physics of an ensemble of real particles. The macro particles reside in a simulation space of finite geometric boundaries. In contrast to other particle methods, the particle interactions are calculated using a discrete grid. The grid divides the simulation space into usually regular subregions or cells (hence the name of the method). Characteristic physical properties of the particles are weighted on the grid using a kernel

function. The momentum equations on the grid are solved with some standard method (for example finite differences). Based on these results, the forces are calculated and interpolated to the particles positions to solve the equations of motion.

PiC simulations use grid points to reduce drastically the amount of computation necessary to produce good approximations to the actual behavior of the underlying physical phenomena. To do this they make use of the common physical property that the influence of any two macro particles upon one another quadratically decreases as the geometric distance between them increases. The computational reduction is accomplished by weighting the contributions of the particles on the grid cells, using a kernel function of finite domain. In this way, the computational complexity of simulating n macro particles drops from $O(n^2)$ to $O(n)$.

In this specific application, the PiC method is used to simulate the phenomena in the electrostatic plasma of a direct current glow discharge in a tube. A plasma medium is an ionized gas, which may be regarded as a collection of ions and electrons interacting through their mutual electric and magnetic fields. PiC methods are often used to simulate plasma phenomena by solving Maxwell's Equations in a numeric manner. This involves computing the dynamics of a large number of electrons and ions in the plasma and the influence of the self-consistent electromagnetic fields. Each macro particle represents about $2 \cdot 10^9$ real particles. Since the problem to be solved shows a cylindrical symmetry and the radius components were of no interest, the simulation is effectively one-dimensional. In order to make a model of the direct particle interactions in the plasma, the method is extended by Monte-Carlo collision (MCC) processes. The collision processes prevent regarding the whole physics of the scattering process, but randomly approximate elastic scattering and the stimulation or ionization of neutral gas atoms by electrons.

The sequential PiC code in C++ was the result of an effort creating a framework for PiC simulations called Open Particle Framework (OPAR) [2]. While designed as an extensible framework, it currently implements only the classes required to simulate the direct current glow discharge. The object-oriented CASE tool *together* was used for the static analysis.

The application provides two major abstractions: a task concept and a diagnosis subsystem. Tasks define the execution of the simulation. Each physical entity is encapsulated in a task (by deriving them from class `CTask`) and implements its simulation code. The task classes follow the Composite design pattern and thus a hierarchical execution structure can be defined. Task objects have an external representation for configuring the simulation. This configuration file makes it possible to define the tasks, their parameters and the static dependencies (associations) between them. On start of the simulation, the static object model is reconstructed from this configuration file. Thus, the user is able to construct arbitrary simulation setups based on the set of implemented physical entities without recompiling the application.

The diagnosis subsystem contains classes for producing output from simulation runs. Any diagnosis output (f.ex.particle trajectories or the plasma density distribution) is implemented in its own diagnosis task. This task can be attached as a subtask to the physical entity to observe, and produces the desired output.

Since *together* does not support dynamic analysis by producing an UML sequence diagram from a concrete run, the dynamic behavior was analyzed by manually tracing some executions. In each timestep, the execution engine runs all configured tasks in the order given by the configuration file. The configuration supports one-time tasks, which can be used for additional setup, and tasks running the entire simulation time.

3 Parallelization strategies

After initializing the particles, each simulation timestep executes four substeps: T_1 weights the particles on the grid, T_2 calculates the solution on the grid, T_3 solves the equations of motion of the particles and T_4 applies the Monte-Carlo processes to the particles.

A parallelization has to give a decomposition of the tasks and consider the data dependencies between the parallel computations. Regarding time complexity, tasks T_1, T_3 and T_4 are linear in the number of particles and T_2 is linear in the number of grid points. In T_1, T_3 and T_4, the computation on different particles is independent, but there is a data dependency for T_1 and T_3 to all grid points (an output dependency in case of T_1, and an input dependency in T_3). In T_2, the computations on a grid point depend on its neighbors. Therefore, T_1, T_3 and T_4 can be decomposed on the number of particles, and T_2 on the number of grid points. For the parallel execution of T_1 and T_3, either the data dependencies on the grid have to be resolved, or concurrent accesses to the grid have to be synchronized. The same holds for T_2 in respect to neighboring grid points.

3.1 Decomposition

A parallel Particle-in-Cell algorithm has the choice of divide up the work done on the particles, the grid or both. The next sections explore and compare these different approaches.

Domain decomposition The first possibility, dividing up the grid into subgrids of the same size where each of them remains on one processor, is also the most common way used for particle methods. Each processor thereby keeps one subgrid and all particles in its local memory.

In step T_1 – computing the density – the processors weight the particles on their part of the grid. Next, the electric fields, potentials and force fields are locally solved in parallel under the consideration of data dependencies between the boundary values of each subgrid. Now, the equations of motion can be solved setting the new positions and velocities of all particles. A synchronization follows, updating the positions and velocities of the particles on all processors to

regain redundancy. Thereby, only the moved particles are exchanged by a scatter function in order to reduce the communication overhead. The used function enables every processor to have all new positions and velocities. Finally, the Monte-Carlo processes are executed in parallel.

The whole communication overhead, neglecting communication of boundary values, depends on the number of particles, making this strategy only useful for problems with a large computation on the grid and a small amount of particles.

Particle decomposition The second strategy divides up the particles on each processor and keeps the grid redundant in memory to resolve data dependencies. After weighting the particles on the local grid, a global reduction operation, which sums up all local grids and distributes the result back to all processors, leaves the application with a redundant global grid. The use of a reduction operation is legal because of the additivity of the physical quantities weighted on the grid. Now, the processors can locally compute the electric field, the potential and the forces on the grid in parallel and subsequently advance their particles by setting their new positions and velocities. After simulating the Monte-Carlo collisions the cycle can be repeated.

This strategy parallelizes only the operations on the particles. The computation on the grid, while executed in parallel, is the same on all processors and thus adds to the sequential overhead. The additional communication overhead depends on the number of grid points transmitted in the reduction operation. This approach is efficient for problems with small grids and a great large number of particles.

Domain and particle decomposition The third possibility is to combine both alternatives: divide up the particles and the grid points. Steps T_1, T_3 and T_4 – dealing with particles – should be divided by particles and step T_2 by grid points.

If the memory is large enough, an implementation could keep the grid redundant and simply merge the steps of the particle and domain decomposition. In step T_1, each processor weights its particles on the local grid. After the grid is subdivided into equal parts, each processor sums up the local contributions of the subgrid now is responsible for from all other processors. This can be done in a single reduction operation. In the next step, the processors compute the fields on each subgrid in parallel, considering the data dependencies between boundary values. Afterwards each processor broadcasts its local grid, bringing all local grids up-to-date, which makes in turn possible to proceed locally with the computation of new positions and velocities of the particles and the Monte-Carlo collisions.

This implementation involves a lot of communication overhead. If tight memory resources prohibit the redundancy of some data structures, like in the algorithm above, this overhead gets even worse. Such an approach is only efficient for problems with large grids and a large number of particles.

Conclusion The choice of a parallelization strategy for an application depends on the percentages of computation spent in updating the particles and the grid. As a requirement of the PiC method, the number of particles must exceed the number of grid points. Usually the computation on the grid contributes less than 10% to the total running time, depending on the complexity of the algorithm. In our case, the number of particles exceeds the number of grid points by a magnitude of 2 to 4, therefore the computation on the grid contributes less than 1%. Thus we decided to decompose the computation only by the number of particles. If the simulation would be extended to three dimensions, which increases the number of grid points and requires more complex algorithms for the solution of the field equations, a combined domain and particle decomposition approach would be most promising.

3.2 Load balancing

Another important point was to add a load balancing component. Initially, the particles are distributed equally to all processors. Due to the Monte-Carlo collisions and ionizations done in the last step of the algorithm, the distribution of the particles changes in course of the simulation. The resulting load imbalance can reduce the speedup significantly. The load balancing component checks the particles distribution periodically and averages the number of particles on the processors. Since the running time of a full timestep is very small, the load balancer must be carefully tuned to add little overhead. This was achieved by checking the balance only every few steps and executing the averaging process, which implies the communication of the particles, only if the imbalance exceeds a configurable limit. Further, the averaging process is stopped, if a sufficient balance is reached. For this application, the load balancer starts rebalancing at 20% imbalance and stops at 5% imbalance. This can be usually done in a single communication step, which keeps the overhead small.

4 Implementation

A goal of the implementation was to parallelize the application without substantially rewriting the sequential code. The implementation had to modify three parts of the code: At initialization the particles must be distributed equally to all processors, between T_1 and T_3, the redundant grid must be updated, and in after some number of timesteps, the particles must be load balanced. Where it is possible, the implementation follows the task model. Initialization of the particles is done in tasks of class `CGeneration`. Since the particles are not initialized from external values but created on startup, the initial distribution of particles to processors could be done by modifying this task (by inheritance) to assign newly created particles to the processors in a round-robin fashion. This step requires no communication.

The summation of the local grids could be implemented transparently as a subtask of class `CDensity`, which is responsible for weighting the particles to the

grid. The subtask executes the reduction operation after the local updates are done.

Since the load balancer depends on the particles, it has been implemented as a subtask of class `CSpecies`, the representation of the particles. After the integration of the equations of motion, this subtask is responsible for checking and, if necessary, averaging the particle distribution.

4.1 TPO++

The implementation of the communication was done in TPO++ [3], an object-oriented message-passing library developed in our group. In this section, we give an overview of point-to-point communication in TPO++. For this application, TPO++ has been extended to collective communication, which is discussed in more detail in the next subsection.

TPO++ implements an object-oriented interface for the functionality of the well-known MPI 1.2 message-passing standard [11]. It is intended to fill the semantic gap between current the message-passing standard MPI and the object-oriented programming paradigm. This includes a type-safe interface with a data-centric rather than a memory-block oriented view and concepts for inheritance of communication code for classes. Other goals were to provide a light-weight, efficient and thread-safe implementation, and, since TPO++ is targeted to C++, the extensive use of all language features that help to simplify the interface. A distinguishing feature compared to other approaches [6, 7, 1, 12] is the tight integration of the Standard Template Library (STL). TPO++ is able to communicate STL containers and adheres to STL interface conventions. All communication methods provide the same orthogonal interface for specifying the data objects to communicate. A sender has two options: provide a single datatype (basic or object) or a range of data elements by using a pair of STL iterators. A receiver has the third option to provide a special back inserter (in analogy to the STL back inserters) that allocates the memory on the receiving side automatically.

The following code examples illustrate some of the features of TPO++. Figure 2 shows two classes enabled for transmission in TPO++. For types with a trivial copy constructor like `Point`, a single declaration is sufficient to achieve this. For more complex types like `Circle`, the user has to provide two marshalling methods named `serialize` and `deserialize`. Figure 3 shows the communication of a single object and a collection of objects of class `Circle` from process 0 to process 1.

For further details on TPO++ and a comparison with other object-oriented message-passing systems see [3].

4.2 Collective communication in TPO++

The implementation of collective communication in TPO++ covers the functionality of MPI 1.2. MPI provides three groups of collective primitives:

```
class Point {
public:
  Point() : x(0), y(0) {}
private:
  int x, y;
};
TPO_TRIVIAL(Point);
```

```
class Circle {
public:
  Circle() : radius(0.0) {
    center = new Point(0.0);
  }
  ~Circle() { delete center; }
  void
  serialize(Message_data& m) const {
    m.insert(*center);
    m.insert(radius);
  }
  void
  deserialize(Message_data& m) {
    m.extract(*center);
    m.extract(radius);
  }
private:
  Point* center;
  double radius;
};
TPO_MARSHALL(Circle);
```

Fig. 2. Examples of two classes enabled for transmission in TPO++.

```
using namespace TPO;
if (CommWorld.rank() == 0 ) { // sender
  Circle c;
  CommWorld.send(c, 1);

  vector<Circle> vc(20);
  CommWorld.send(vc.begin(), vc.end(), 1);
} else { // receiver
  Circle c;
  CommWorld.receive(c);

  vector<Circle> vc(20);
  CommWorld.receive(vc.begin(), vc.end());
}
```

Fig. 3. Transmission of user-defined objects.

```
vector<double> vd(10);
TPO::CommWorld.bcast(vd.begin(), vd.end(), 2);
```

Fig. 4. Broadcast of a container of floating-point values in TPO++ rooted at process 2.

- The basic collective primitives `broadcast` and `barrier`.
- Four data-exchange primitives (`scatter`, `gather`, `allgather` and `alltoall`).
- Four combination primitives (`reduce`, `allreduce`, `reduce_scatter` and `scan`).

The goal of the TPO++ implementation of collective communication was to provide a interface consistent with its point-to-point interface and the STL conventions. Of course, the implementation should show a reasonable performance compared to MPI. The existing interfaces suggest an implementation of collective operations as methods of class `TPO::Communicator`. The methods should accept the different kind of data structures discussed in section 4.1. For the details of the interface, several characteristics different from the point-to-point communication are relevant:

Five primitives (`broadcast` and the combination primitives) are *rooted operations*, i.e. they are asymmetrical in respect to one process. The root must be somehow given for these operations. Since it is likely to change in subsequent calls, we chose to add it to the methods parameters. Figure 4 gives an example of broadcasting a vector of floating point values.

The data-exchange and combination primitives are operations on different input and output data structures, given in the same call. With two variants to send and three variants to receive date, we have a maximum of six possible overloaded methods for these operations. Looking more closely, not all possibilities are reasonable, for example a scatter operation, which sends only one element, is impossible[3]. Table 1 summarizes the overloaded methods of these operations.

The data-exchange primitives also come in a vector-variant, which does not require the data structures of the participating processes to be of the same size. MPI allows the clients for p processes to pass p memory blocks of different size and offset relative to a common base address. An obvious generalization in C++ is to allow the clients to pass a number of element ranges, each given by a begin and end iterator. Several different realizations of such an interface are imaginable. TPO++ favors a container of pairs of begin and end iterators. This simplifies the interface, since only one additional parameters is needed, and enhances readability and safety due the explicit pairing of corresponding iterators. For convenience in the common case of adjacent element ranges of different sizes, TPO++ provides another interface, where only a single container of $p+1$ iterators defining the p segments of different length is required.

[3] Except in the boundary case of one process, in which no communication is needed.

The four combination operations require another parameters, the operation to be applied to the elements given by the processes. Standard operations defined in MPI include for example the sum or the minimum of the values. In conformance with the STL, TPO++ allows the client to pass an arbitrary function object. Unlike the STL, MPI further requires the client to state the commutativity of the operation, which is realized by a boolean class attribute of the function operators type. Figure 5 gives an example of an user-defined function operator and a reduction in TPO++.
The implementation of function operators cannot hide entirely the MPI layer and some deviations of the STL semantics. Unlike STL function operators, the function operators in TPO++ obviously cannot be state-dependent, since the operations are executed on different hosts in different instances. Moreover, TPO++ requires the data objects used in reduction operators to provide a default constructor and to be of constant size. These restrictions result from the constraints of MPI user-defined combination operations.

	Receiver		
Sender	Single	Collection	Back-inserter
Single	Broadcast, Reduce, Scan, Allreduce	(All)Gather	(All)Gather
Collection	Scatter	All operations	All operations

Table 1. Overload collective communication methods. The sender can pass *single* objects or arbitrary *collections* of elements in STL containers. The receiver can get *single* objects, *collections* of elements in STL containers or allocate the space automatically using a *back-inserter.*

```
// user-defined operator
// for combination operations
template <class T>
class sum {
  public:
  static bool commute;
  void operator()(T& inout,
    const T& in) {
    inout += in;
  }
};

template <class T>
bool sum<T>::commute = true;
```

```
vector<double> source(20);
vector<double> result(20);

// reduction
CommWorld.reduce(source.begin(),
                 source.end(),
                 result.begin(),
                 result.end(),
                 sum<double>(),
                 0);
```

Fig. 5. Example of an user-defined reduction operation `sum` and its application in the reduction of a container of floating-point values rooted at process 0.

Related work To our knowledge, three object-oriented message-passing systems, mpi++ [6, 7], para++ [1] and OOMPI [12] implement collective communication primitives, which significantly differ from the MPI C++ bindings [9, 10].

Para++ implements the concept of C++ IO streams for message-passing and provides only a broadcast and multicast primitive for collective communication[4]. Obviously, having only broadcast is not sufficient for most applications.

mpi++ implements the full set of MPI 1.2 collective communication primitives. In mpi++ collective communication primitives are implemented in two template classes, `Collective` and `Reduction`. `Collective` is parameterized with the type information about the data to send and receive, its methods implement all basic and data-exchange primitives, and its attributes hold the communicator to use and possibly the root of the collective communication. The derived class `Reduction` implements the combination primitives and encapsulates additional information about the type of operation to apply. Different to TPO++, mpi++ reintroduces, in analogy to MPI, its own type system, but does not support the STL. Also, the interface is built around the operations, which are reified as classes, and avoids method parameters in favor of attributes.

OOMPI implements the full set of MPI 1.2 collective communications. One of basic abstractions in OOMPI, class `Port` is used to specify the master for rooted operations. All other operations are implemented in the communicator class `Intra_comm`. Class `OOMPI_Op` is a simple wrapper for MPI operators. The constructor also accepts user-defined MPI operators. Unfortunately, at this point, the underlying MPI layer is visible for the user.

5 Performance measurements

The performance of the Particle-in-Cell code has been measured on two different architectures, the Cray T3E, a conventional supercomputer installed at the German supercomputer center in Stuttgart [4] and Kepler, a self-made clustered supercomputer[6] based on commodity hardware [13]. The T3E has 512 nodes, each equipped with a DEC Alpha EV5 21164 processor running at 450 Mhz and 128 MB of RAM. The interconnect organizes the nodes in a three-dimensional torus ($8 \times 8 \times 8$), and every connection to the 6 neighbors provides a nominal bidirectional bandwidth of about 500 MB/s. Measurements of MPI application to application performance gives a bandwidth of about 300 MB/s and 15μs latency. Keplers 96 nodes are running two Pentium III processors at 650 Mhz and have 1 GB of total memory, or 512 MB per processor. Kepler has two interconnects, a fast ethernet for booting the nodes and administration purposes and a Myrinet network for parallel applications. The latter has a multi-staged hierarchical switched topology organized as a fat tree. The nominal bandwidth of 133

[4] In MPI, the multicast is realized by creating a new communicator containing only a subset of all processes.

[6] As of July 2001 Kepler occupies rank 290 at the TOP 500 list of supercomputers.

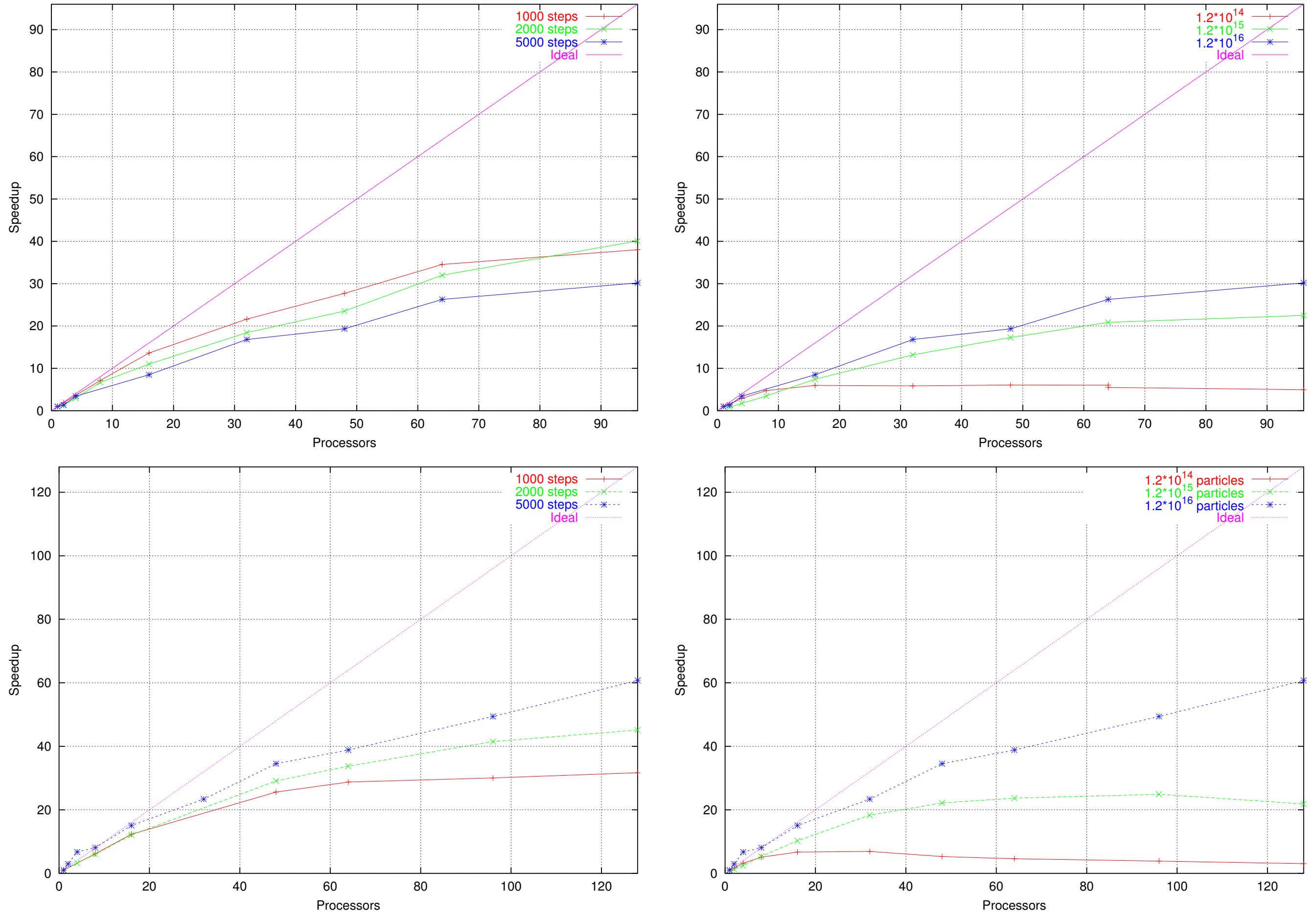

Fig. 6. Measurements on Kepler cluster. Upper row: Load balancing disabled. Lower row: Load balancing enabled. Left column: $1.2 * 10^{16}$ particles for different numbers simulation steps. Right column: 5000 simulation steps for different number of particles.

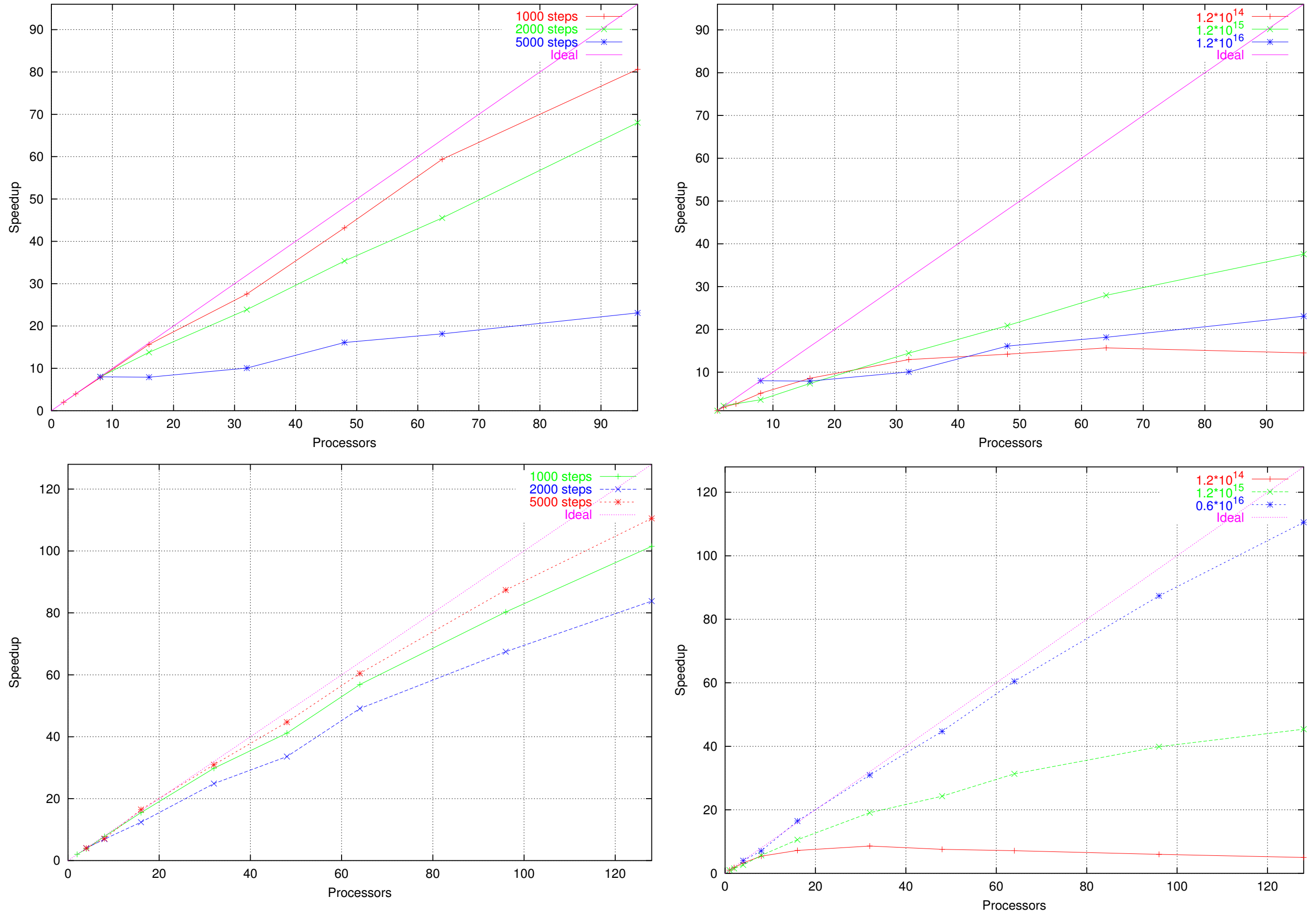

Fig. 7. Measurements on Cray T3E. Upper row: Load balancing disabled. Lower row: Load balancing enabled. Left column: $1.2 * 10^{16}$ particles for different numbers simulation steps. Right column: 5000 simulation steps for different number of particles.

MB/s is the maximum PCI transfer rate, measurements give about 115 MB/s bandwidth and 7μs latency.

The Particle-in-Cell application has been run with couple of different parameters. Three different numbers of particles have been used to observe the effects of a varying computation to communication ratio. Three different numbers of simulation steps show the growing imbalance without load balancing and the improvements after load balancing. All measurements have been made on up to 128 processors.

Figure 6 shows the results for the Kepler cluster, figure 7 for the Cray T3E. Note that the speedups of the large runs on Cray T3E with $1.2 * 10^{16}$ particles and 5000 simulation steps are based on the runtimes for 8 processors, which improves these speedups artificially. The problem did not fit in a smaller number of processors. For the same reason, the largest runs on Cray T3E use only $0.6 * 10^{16}$ simulation particles.

The different number of simulation steps show the effect of a growing imbalance if load balancing is disabled, which results in a decreasing performance with increasing number of simulation steps. With load balancing enabled, the picture is reversed. The speedups improve and the application can profit from the increasing number of particles created in course of the simulation. For smaller number of simulation particles, the load balancing is not able to improve the speedups, in case of $1.2 * 10^{14}$ particles they even decrease. Regarding the initial number of simulation particles, the results clearly show a break-even between $1.2 * 10^{15}$ particles, showing moderate speedups, and $1.2 * 10^{16}$ particles, with very good performance. For $1.2 * 10^{14}$ particles, the application can gain only limited runtime improvement.

6 Conclusions

While the task model allows flexible combination of physical entities, the configuration is restricted to a sequential execution model. In this particular domain, an extension could provide parallel execution primitives. A shortcoming of this implementation is the lack of a consistent organization beyond the task abstraction. For example, the objects defining the geometry and representing the particles are used globally, and therefore are tightly coupled to all other classes. As a consequence of this, it is impossible to configure the application without knowing the classes in detail. The lack of documentation makes it even more complicated.

The use of a design tool simplified the reverse-analysis of the application substantially as well as the understanding of the class dependencies, which otherwise would have not been possible. Regarding the parallelization, static and dynamic analysis helped to select from the theoretical approaches the most reusable strategy, which is not necessarily the most efficient one. In this case, both strategies were the same, which simplified the parallelization substantially. The subsequent addition of a parallelization without modification of the sequential code

was possible due to the flexibility of the task model. The tool also helped a lot documenting the code.

References

1. O. Coulaud and E. Dillon. Para++: C++ bindings for message-passing libraries. In *EuroPVM Users Meeting*, September 1995.
2. T. Daube and H. Schmitz. OPAR: Open architecture C++ plasma simulation code. Ruhr-Universität Bonn, 1998.
3. T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An object-oriented message-passing library in C++. pages 43–50. IEEE Computer society, 2000.
4. High performance computing center Stuttgart. *Cray T3E-900/512*. Online. URL: http://www.hlrs.de/hw-access/platforms/crayt3e (January 2001).
5. M. Hipp, S. Hüttemann, M. Konold, M. Klingler, P. Leinen, M. Ritt, W. Rosenstiel, H. Ruder, R. Speith, and H. Yserentant. A parallel object-oriented framework for particle methods. In E. Krause and W. Jäger, editors, *High Performance Computing in Science and Engineering '99*, pages 483–495. Springer-Verlag, 1999.
6. D. Kafure and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995.
7. D. Kafure and L. Huang. Collective communication and communicators in mpi++. Technical report, Department of Computer Science Virginia Tech, 1996.
8. Los Alamos National Laboratory. *POOMA*, 2000. Online: http://www.acl.lanl.gov/PoomaFramework.
9. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.
10. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.
11. M. Snir and W. Gropp. *MPI: The complete reference.* MIT Press, 1998.
12. J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of the POOMA conference*, 1996.
13. University of Tübingen. *Kepler cluster website.* Online. URL: http://kepler.sfb382-zdv.uni-tuebingen.de.
14. T. Veldhuizen. Arrays in blitz++. In D. Caromel, R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, pages 223–231, 1998.
15. G. V. Wilson and P. Lu, editors. *Parallel Programming using C++.* The MIT Press, Cambridge, 1996.