Toward an Object-Oriented Core of the PPM Library

Omar Awile, Ömer Demirel and Ivo F. Sbalzarini

Institute of Theoretical Computer Science and Swiss Institute of Bioinformatics,
ETH Zurich, Universitätstr. 6, CH-8092 Zurich, Switzerland

Abstract. As high-performance computing (HPC) machines become increasingly complex, middleware-based programming paradigms have been particularly successful in reducing code development time and increasing simulation efficiency. The parallel particle-mesh (PPM) library is a state-of-the-art HPC middleware for parallel particle-mesh simulations. It is based on a concise set of six data and operation abstractions. The present paper describes the architecture of the new PPM library core. This new core architecture enables several simplifications in the library's user interface and supports for the first time the implementation of multi-resolution simulations using PPM. We further demonstrate the competitive performance of the new core architecture compared to the previous version of the PPM library.

Keywords: High-Performance Computing, Parallel Computing, Scientific Computing, Middleware, Abstractions
PACS: 02.60.Cb, 02.70.-c, 02.70.Ns

INTRODUCTION

While the use and need for high-performance computing (HPC) are proliferating, programming large parallel computers is still prohibitively intricate and time-consuming. As a result, the efficiency of simulations (in Flops/Watt) is decreasing as hardware platforms become more complex (performance gap). Moreover, only a limited group of researchers has the skills to fully harness the power of the ever larger HPC systems (knowledge gap). Both issues can be addressed by providing an additional layer of abstraction between the HPC system and the scientific programmer. These abstractions can be implemented as a middleware, hiding much of the complexity of the underlying system. The benefits of such HPC middleware have already been demonstrated in projects such as POOMA [1], TRILINOS [2], PETSc [3], and ASTRID [4]. The simplicity provided by a middleware, however, comes at the price of reduced versatility: it limits the user to a specific class of numerical methods.

Here, we consider an HPC middleware for hybrid particle-mesh simulations. Particle-mesh methods provide a unifying framework for numerical simulations. They can simulate models of all four classes (discrete/continuous; deterministic/stochastic). Continuous deterministic models as formulated by partial differential equations can be solved on meshes (using, e.g., finite differences) or particles (using, e.g., particle strength exchange [5]). In discrete models, the particles often have a direct real-world correspondence, such as the atoms in molecular dynamics simulations.

The PPM library [6, 7] defines the state of the art in middleware for distributed-memory particle-mesh simulations. It hides MPI from the application programmer by introducing an additional, transparent layer beneath the user's simulation programs (called "PPM clients"). Since PPM reduces the knowledge gap, the resulting simulations often outperform hand-parallelized codes [6, 8]. The PPM library is independent of specific applications, provided the simulation is phrased in terms of particles, meshes, or a combination of the two. PPM implements modules for adaptive domain decomposition, communication through halo layers, load balancing, particle-mesh interpolation, and communication scheduling. All of this is done transparently without participation of the user program. For further details about the PPM library, we refer to the original publications [6, 7].

Recently, the abstractions upon which PPM is based have been explicitly formalized and extended to a simulation-specification language [8]. This triggered a major re-design of the PPM library's core architecture. The goal of this re-design was to more cleanly reflect the underlying abstractions and to provide encapsulated derived types. This has enabled significant simplifications in the interfaces of most PPM core routines. In addition, we have split the PPM library into two parts: the PPM core and a PPM numerics library. The numerics library uses the routines provided by the core to implement frequently used numerical solvers, such as multi-grid and FFT Poisson solvers, multi-stage ODE integrators, fast multipole methods, and redistancing methods for level sets.

In this paper, we present the new core architecture of the PPM library and compare its performance and parallel scalability with that of the previous version of PPM.

CP1281, *ICNAAM, Numerical Analysis and Applied Mathematics, International Conference 2010*
edited by T. E. Simos, G. Psihoyios, and Ch. Tsitouras

ABSTRACTIONS FOR PARALLEL PARTICLE-MESH SIMULATIONS

The new PPM core implements encapsulated modules of six entities that allow describing parallel particle-mesh simulations on a high level of abstraction [8]:

- *Topologies* are adaptive domain decompositions that divide the physical space into subdomains of varying size and assign those subdomains to processors. Multiple topologies can be defined at the same time.
- *Particles* are zero-dimensional computational elements defined by their position and a vector of properties.
- *Meshes* are regular Cartesian grids defined by their resolution. Several meshes of different resolutions can be associated with the same topology.
- *Connections* link particles to graphs or unstructured grids (or model bonds in discrete simulations).
- *Mappings* perform communication between processors. There are four mapping types: global, local, ghost-get, and ghost-put. Global mappings distribute particles, meshes, or connections according to a certain topology. Local mappings exchange particles between neighboring processors. The ghost-get and ghost-put mappings populate the halo layers or send halo information back to the corresponding source processor, respectively. All mappings are implemented as stacks and transparently determine a near-optimal communication schedule.
- *Interactions* perform local (per subdomain) computations between particles, meshes, connections, or any combination of these. Due to the presence of halo layers, this requires no communication.

These abstractions provide an intermediate level of granularity, between the fine-grained communication abstractions of MPI and the coarse-grained abstractions of FFT libraries or finite-element libraries. Moreover, the present set of abstractions separates communication from computation. This allows assessing and optimizing the communication overhead of a simulation already in its abstract specification. Using these abstractions, a simple particle simulation can be specified in only a few lines:

```
initialize positions and properties of particles
topo ← makeTopo(particles,domain,decomposition)
globalMapping(particles,topo)
for t = 0 to end do
   ghost-get(particles,topo)
   interaction(particles,topo,kernel)
   localMapping(particles,topo)
end for
```

IMPLEMENTATION IN PPM

The new PPM core implements the above abstractions as encapsulated data structures. This consequently reflects the layer paradigm of software engineering and prevents the user from exposure to lower abstraction levels and internal library information. The implementation relies on Fortran derived types and the capability of declaring data members private. These encapsulated objects provide a first step toward a fully object-oriented core of the PPM library. Other concepts, such as inheritance and polymorphism, are also supported in Fortran 90/95 [9, 10], and Fortran 2003 adds compiler support for object-oriented programming.

Consider the encapsulated data structure for the topology abstraction. A topology object necessarily contains all data describing the decomposition and distribution of the particles and meshes on the different processors. In the new architecture, all of these data are collected in a derived type `ppm_t_topo`. When a new topology is created, an object of this type is instantiated and the user is given a handle to this object, without access to its internal data. The number of concurrently defined topology objects in a simulation is unlimited. The client keeps track of all handles, whereas the PPM library internally stores the corresponding opaque objects. This architecture dispenses with the previous need of designating a particular topology as "current". Instead, the topology handle is passed as an argument to all mapping and interaction routines. Relaxing the limitation of having a single "current" topology allows for multi-resolution simulations where the data can be simultaneously distributed across several topologies.

PPM client design is assisted by the one-to-one correspondence between the above-described abstractions and the modules of the PPM library. Translating an abstract specification of a simulation to source code is straightforward. The code example in Figure 1 illustrates how the second and third lines of the above example (topology creation and global mapping) would be implemented using the new PPM core.

```
! create a topology; the parameters decomp and assign are symbolic
! constants that select the decomposition and assignment strategies
CALL ppm_mktopo(topoid,xp,N,decomp,assign,minphys,maxphys,bc,ghostsize,cost,info)
! map all particle positions and their weights to the new toplogy
CALL ppm_map_part_global(topoid,xp,Npart,info)    ! positions
CALL ppm_map_part_push(wp,1,Npart,info)           ! weights
CALL ppm_map_part_send(Npart,Mpart,info)          ! send/receive all data
CALL ppm_map_part_pop(wp,1,Npart,Mpart,info)      ! weights
CALL ppm_map_part_pop(xp,ndim,Npart,Mpart,info)   ! positions
```

FIGURE 1. Initializing a particle simulation using PPM takes only a few lines. This code section corresponds to lines 2 and 3 of the above pseudo code. We assume that the user has already initialized the particles (`xp`,`wp`). After executing, all particle positions `xp` and their weights `wp` are distributed according to the topology with decomposition and assignment (`decomp`, `assign`).

PERFORMANCE RESULTS

Enforcing data hiding and encapsulation allowed simplifying many of the subroutine interfaces in PPM. Furthermore, we added for each mapping type a separate routine, eliminating the need for a "mapping type specifier". This further simplified and shortened the routine interfaces and reduced the number of globally stored state variables. In order to test whether the new data structures and interfaces imply any performance toll, we migrated an existing PPM client to the new library. This client solves the diffusion equation in complex, three-dimension geometries using the method of particle strength exchange [5]. The client has previously been extensively used with the original version of PPM [11, 6]. In order to migrate it to the new library core, about 50 lines of code needed to be changed in the client. The client based on the new PPM core and the original one produced the exact same numerical results on all digits.

We measured the wall-clock time per time step and the parallel efficiency by running the test client on an increasing number of processors using the original PPM and the new implementation. All tests were done on a cluster of quad-core AMD Opteron 8380 CPUs running Linux and connected through an InfiniBand QDR network. Each cluster node has four CPUs and 32 GB of RAM, shared between its 16 cores. The library and the client were compiled with the Intel Fortran compiler v11.1 using the `-O3` optimization flag and linked against OpenMPI 1.4.2. The results are shown in Figure 2. The new architecture does not seem to have any adverse effects on performance, at least in this application. The new PPM core shows slightly reduced wall-clock times on all numbers of cores tested. The parallel efficiency is mostly higher than that of the old PPM, except on 8 cores where it is 2% lower.

CONCLUSIONS

We have presented the architecture of the new core of the PPM library and compared its performance to that of the previous version of PPM. The new core implements encapsulated abstractions [8] as derived types. The use of derived types and clean encapsulation has led to a leaner interface of PPM and dispensed with the need of singling out a certain topology as "current". This simplifies the development of PPM clients and allows for the implementation of multi-resolution simulations. Moreover, we have split the PPM library into a core part and a numerics part. This speeds up compilation of the library and simplifies development and debugging of numerical modules. Benchmark simulations using a test client have shown that the new architecture does not lead to a reduction in performance.

The presented code re-design constitutes a first step toward a scalable, object-oriented middleware for high-performance parallel particle-mesh simulations. Present and future work is concerned with testing and including features of object orientation as introduced in Fortran 2003. The new release of the PPM library is available on http://www.ppm-library.org, along with a new and updated documentation.

ACKNOWLEDGMENTS

We thank Prof. Jens Walther (DTU Copenhagen) and Prof. Petros Koumoutsakos (ETH Zurich) for many fruitful discussions and for having started the original PPM project. We gratefully acknowledge operational support from the ETH Zurich cluster team of Olivier Byrde. This project was supported with a grant from the Hasler Foundation (Switzerland) and a grant from the Swiss SystemsX.ch initiative, grant No. LipidX-2008/011.

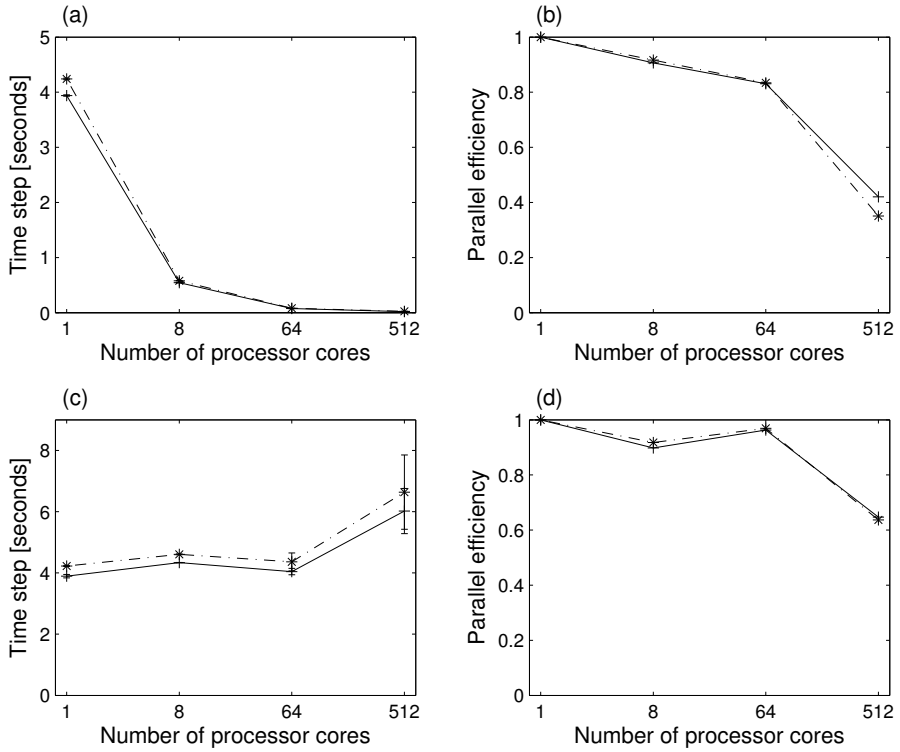

FIGURE 2. Average (symbols) and standard deviation (error bars) of the maximum (over all processor cores) wall-clock time per time step, sampled over 100 steps, and parallel efficiency for the old (dashed lines) and new (solid lines) PPM implementation. (a) and (b) show the results for a fixed-size problem with 2.1 million particles (strong scaling); (c) and (d) for a scaled-size problem starting with 2.1 million particles on 1 processor and going to 1.1 billion particles on 512 processors (weak scaling). All tests were done in a cubic domain using equi-sized and perfectly load-balanced subdomains in order to test the performance of the PPM implementation rather than that of the implemented decomposition algorithms.

REFERENCES

1. J. Reynders, J. Cummings, M. Tholburn, P. Hinker, S. Atlas, S. Banerjee, M. Srikant, W. Humphrey, S. Karmesin, and K. Keahey, "POOMA: a framework for scientific simulation on parallel architectures," in *Proc. 1st Intl. Workshop on High-Level Programming Models and Supportive Environments*, 1996, pp. 41–49.
2. M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, *ACM Trans. Math. Softw.* **31**, 397–423 (2005).
3. S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 – Revision 2.1.5, Argonne National Laboratory (2004).
4. E. Bonomi, M. Flück, R. Gruber, R. Herbin, S. Merazzi, T. Richner, V. Schmid, and C. T. Tran, "ASTRID: a programming environment for scientific applications on parallel vectorcomputers," in *Scientific Computing on Supercomputers II*, edited by J. T. De Vreese, and P. E. van Kamp, Plenum Press, New York, 1990, pp. 51–82.
5. P. Degond, and S. Mas-Gallic, *Math. Comput.* **53**, 485–507 (1989).
6. I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos, *J. Comput. Phys.* **215**, 566–588 (2006).
7. I. F. Sbalzarini, J. H. Walther, B. Polasek, P. Chatelain, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos, *Lect. Notes Comput. Sc.* **4128**, 730–739 (2006).
8. I. F. Sbalzarini, *Intl. J. Distr. Systems & Technol.* **1(2)**, 40–56 (2010).
9. L. Machiels, and M. O. Deville, *ACM Trans. Math. Softw.* **23**, 32–49 (1997).
10. V. Decyk, C. Norton, and B. Szymanski, *Comput. Phys. Commun.* **115**, 9–17 (1998).
11. I. F. Sbalzarini, A. Mezzacasa, A. Helenius, and P. Koumoutsakos, *Biophys. J.* **89**, 1482–1492 (2005).