III International Conference on Particle-based Methods – Fundamentals and Applications
PARTICLES 2013
M. Bischoff, E. Oñate, D.R.J. Owen, E. Ramm & P. Wriggers (Eds)

A DOMAIN-SPECIFIC PROGRAMMING LANGUAGE FOR PARTICLE SIMULATIONS ON DISTRIBUTED-MEMORY PARALLEL COMPUTERS

OMAR AWILE[1,2], MILAN MITROVIĆ[2], SYLVAIN REBOUX[2] AND IVO F. SBALZARINI[1,2]

[1] MOSAIC Group, Center of Systems Biology
Max Planck Institute of Molecular Cell Biology and Genetics
Pfotenhauerstr. 108, 01307 Dresden, Germany.
ivos@mpi-cbg.de, mosaic.mpi-cbg.de

[2] Previously: MOSAIC Group,
ETH Zurich, Department of Computer Science
Universitätstr. 6, 8092 Zurich, Switzerland.

Key words: Parallel Computing, Domain-Specific Language, Software Engineering, Parallel Particle Simulation

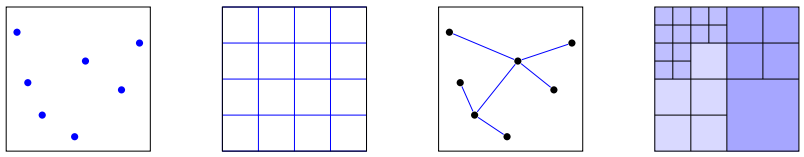
Abstract. We present PPML, a domain-specific programming language for parallel particle and particle-mesh simulations. PPML provides a concise set of high-level abstractions for particle methods that significantly reduce implementation times for parallel particle simulations. The PPML compiler translates PPML code into standard Fortran 2003 code, which can then be compiled and linked against the PPM runtime library using any Fortran 2003 compiler. We describe PPML and the PPML compiler, and provide examples of its use in both continuous and discrete particle methods.

1 INTRODUCTION

Computer simulations are well established as the third pillar of science, alongside theory and experiment. Hardware platforms and high-performance computers are becoming increasingly powerful, enabling unprecedented simulations. Using such hardware, however, requires an increasing amount of specialist knowledge from the programmer.

Many approaches have been taken to render simulation platforms more accessible to a wider audience. Often, the strategy is to provide an intermediate layer of abstraction. The most common approach for introducing abstraction layers is to provide a software library with "canned" building blocks that the user can call upon when building a simulation. An alternative approach is to provide a domain-specific language (DSL).

Particle methods provide a versatile framework for both continuous and discrete simulations. In discrete simulations particles can be used to represent the individual agents of



(a) particles (b) mesh (c) connections (d) topology

Figure 1: PPM provides 4 data abstractions: particles (a), meshes (b), connections (c), toplogies (d). Hybrid particle-mesh simulations can be expressed in terms of particles and (Cartesian) meshes. Connections allow particle-particle associations, representing, e.g., bonds in molecular dynamics simulations. Topologies define domain decompositions and the assignment of subdomains to processes (see also Fig. 2).

a model (e.g., atoms, cars, or animals). Continuous systems can be discretized using particles that represent Lagrangian tracer points (moving finite volumes) and evolve through their pairwise interactions.

DSLs offer unique opportunities for hiding model and hardware complexity while offering the programmer (or user) a simple and expressive interface. An early example of a DSL was BNF[1], a formalism for describing the syntax of programming languages, document formats, communication protocols, and more. Examples of DSLs for numerical simulations include DOLFIN (part of the FEniCS project)[2;3;4] for finite-element simulations, PELLPACK[5;6] for mesh-based PDE solvers, Uintah[7] for massively parallel PDE solvers, and Liszt[8] for mesh-based PDE solvers based on the Scala programming language.

Here, we present a domain-specific language for hybrid particle-mesh simulations on distributed-memory parallel computers. It is derived from parallel particle-mesh abstractions[9] and simplifies code development by introducing domain-specific data types and operations for particle-mesh methods, along with language elements that provide a concise notation. We follow the same philosophy as Liszt and FEniCS in providing a high-level DSL that is compiled to standard Fortran code linking against the PPM library[10;11;12] as a runtime system.

2 ABSTRACTIONS

The PPM library[10;11;12] is a middleware implementing a number of abstractions for parallel hybrid particle-mesh simulations without the burden of dealing with the specifics of heterogeneous parallel architectures, and without losing generality with respect to the models that can be simulated using particle-mesh methods. The PPM library provides three groups of abstractions:

1. **Data abstractions** provide the programmer with the basic building blocks of hybrid particle-mesh simulations. This allows one to reason in terms of particles, meshes, and connections, rather than arrays and pointers (see Fig. 1).
2. **Communication abstractions** provide transparent inter-process communication for the data abstractions (see Fig. 2). They make the incurred communication overhead explicit, while hiding the intricacies of programming a parallel application.

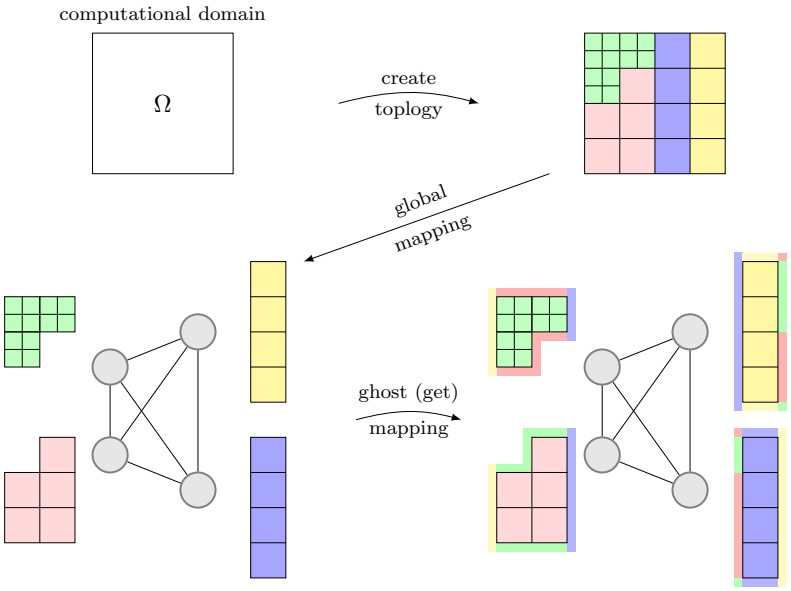


Figure 2: In the *topology* abstraction, the computational domain Ω is decomposed and the subdomains are assigned to processors (upper panels). A *global mapping* distributes particles, mesh data, and connections across the processors (represented by the nodes in the graph) as specified by to the topology (lower-left panel). The *ghost-get mapping* creates ghost layers around subdomains according to the topology and the boundary conditions (in the example shown here, periodic boundary conditions).

Making the communication overhead explicit to the programmer helps assess the expected parallel scalability of a PPM program.

3. **Compute abstractions** encapsulate computations performed on the data abstractions. For particle–particle interactions they internally use cell lists[13], Verlet lists[14], or adaptive-resolution neighbor lists[15] to efficiently find interaction partners.

3 The PPML Language

The PPML language implements the above abstractions in an embedded DSL (eDSL), embedded in Fortran 2003. The PPML compiler translates PPML code to standard Fortran 2003 code that links against the PPM library as its runtime system.

3.1 PPML syntax and features

Besides particles, meshes, and connections, PPML provides the following types:

- A *field* is an abstract type that represents a *continuous* mathematical function that may be scalar or vector-valued (e.g., temperature, velocity, ...). Fields can be discretized onto particles or meshes.
- *Operators* are abstract objects that represent mathematical operators that can be applied to fields. The rationale behind equipping PPML with operators and fields is to allow the user to express the governing equations of a model directly in PPML. These types can also be used to provide the user with contextual feedback during execution, and with annotated error messages.
- *Operator discretizations* are defined by discretizing an operator over particles or a mesh. They amount to implementations of numerical schemes for the respective operator.

```
foreach p in particles(pset) with fields(f) ! [options]
   ! iterator body
   f_p =    ! set value of field f for particle p
end foreach
```

Particle position and property data can directly be accessed using the `foreach` particle iterator. Individual particle positions and properties are accessed using a LaTeX-like subscript notation.

```
foreach n in equi_mesh(M) with sca_fields(f,df) indices(i,j) stencil_width(1,1)
   for real
      df_n = (f_n[-1,] + f_n[+1,] +  f_n[,-1] + f_n[,+1] - 4.0_mk*f_n)/h2
   for north
      ! set boundary condition for north boundary
end foreach
```

Mesh iterators allow the programmer to loop through all nodes of a mesh, irrespective of its geometry. The basic `foreach` control-flow structure can be extended with user-defined options and clauses. Array index offsets can be used as a notational shortcut when writing mesh operator stencils.

Table 1: Examples of PPML control-flow structures.

- The *ODE* type encapsulates time stepping methods that can be used to evolve the particle positions and properties over time.

In addition to types and operators, PPML also provides `foreach` control-flow structures (Table 1) for intuitively iterating through particles or mesh nodes. These iterators also provide special clauses for accessing only parts of a mesh (i.e., its bulk or its halo layers) or subsets of particles (i.e., only real particles or only ghost particles). This is particularly useful when writing operator stencils. Naturally, `foreach` loops can be nested and composed, providing great flexibility. PPML extends Fortran's array index operator `()` with an array index offset operator `[]`. This operator simplifies writing finite-difference stencils on meshes by only specifying relative index offsets.

PPML also extends Fortran by simple type templating. Modules, subroutines, functions, and interfaces can be templated. Multiple type parameters can be specified and their combinations chosen.

3.2 Implementation

The PPML framework is composed of three parts: a *parser*, a *macro collection,* and a *generator*. The *parser* reads PPML/Fortran code and generates an abstract syntax tree (AST) as an intermediate representation. The parser largely ignores Fortran, but is aware of scope declarations, such as modules, subroutines, functions, and derived types. The *macro collection* consists of a set of macros that contain template PPML operations and iterator code. The *generator* performs the actual source-to-source compilation. Whenever a PPML instruction is encountered, the generator looks up the appropriate macro and

```
macro create_mesh(topology,offset)
% scope.var({result.to_sym => "type(ppm_mesh),pointer::#{result}"},
%          :ppm_mesh)
allocate(<%= result %>,stat=info) or_fail_alloc("Could not allocate <%= result %>")
call <%= result %>%create(<%= topology %>,<%= offset %>,info)
end macro
```

Listing 1: A PPML macro implementing a mesh creation operation. This macro declares a variable of type `ppm_mesh` in the current scope. The name of the variable is determined by the PPML parser from the left-hand side of the assignment within which this macro is invoked. The success of the allocation is checked using the `or_fail_alloc` macro. Finally, the type-bound procedure `create` is called on the newly allocated `ppm_mesh` instance (see Ref.[12] for details).

injects the resulting Fortran code. The structure of the PPML framework is shown in Fig. 3. It is implemented using the Ruby programming language, which provides a number of libraries and tools for parsing and code generation.

The PPML parser uses the ANTLR parser generator. This parser and lexer generator allows us to define PPML's syntax and grammar using an extended BNF-like grammar[12]. Since PPML is an embedded language, it is sufficient for the parser to recognize those aspects of Fortran that are used as part of PPML code, or where an ambiguity between PPML and Fortran would otherwise arise.

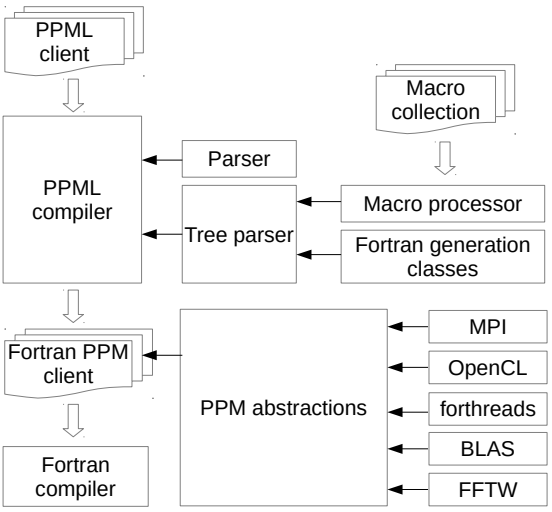


Figure 3: The PPML framework. Line arrows represent "used by" relationships, hollow arrows show the processing order. The PPML compiler is supported by the parser and the tree parser, which in turn uses the PPML macro collection to expand PPML code to Fortran 2003.

PPML macros are eRuby Fortran templates that are evaluated at code-generation time. eRuby is a templating system that allows embedding Ruby code into text documents that are expanded to clear text at execution-time. Macro calls within a macro are processed recursively. Listing 1 shows an example PPML macro.

Besides data abstractions and iterators, PPML provides a number of convenience macros for frequently used tasks ranging from error handling and reporting to command-line argument and configuration-file handling.

PPML's flexible infrastructure and the use of macros encourage the programmer to extend the language capabilities wherever required. An existing macro can be overwritten by a user-defined version. Furthermore, new macros can easily be defined and are imported into the macro collection.

4 EXAMPLES AND BENCHMARKS

We illustrate the use of PPML and benchmark its performance and parallel efficiency in both a continuous and a discrete example. The continuous example considers simulating a reaction-diffusion system using a DC-PSE particle method[16]. The discrete example considers a simple Lennard-Jones[17] molecular dynamics simulation.

4.1 PPML examples

We present two minimal examples simulating diffusion in the domain $(0,1) \times (0,1)$ using either a mesh-based discretization or a particle discretization. Diffusion of a scalar quantity $U(\mathbf{x}, t)$ is governed by $\partial U/\partial t = \Delta U$. Listing 2 shows the PPML code using a Cartesian mesh discretization and second-order central finite-difference stencils to discretize the Laplace operator. Lines 1–10 are variable and parameter declarations. Note that PPML objects do not need to (but can) be explicitly declared. Lines 12–17 initialize PPM, create a field, a topology, a mesh, and discretize the field onto the mesh. Lines 18–23 specify the initial condition. In this example this is done by sampling a Gaussian on the mesh using a mesh-`foreach` loop. Line 24 creates an `ode` object, passing to it the right-hand-side callback function, the fields and discretizations used inside the right-hand-side function, the fields to be updated by the time integrator, and the integration scheme to be used. In lines 25–31 we implement the main time loop of the simulation. Time start, end, and step size can be hard-coded, but are usually provided at runtime through command-line arguments or parameter files. Line 32 finalizes the client by calling the macro for PPM finalization. Lines 35–45 implement the right-hand side of the simulation using a mesh-`foreach` loop over all real (i.e., non-ghost) mesh nodes.

```
client mini
  integer, dimension(4) :: bcdef = ppm_param_bcdef_periodic
  integer               :: istage = 1
  ! mesh parameters
  real(mk), dimension(2), parameter :: offset = (/0.0_mk, 0.0_mk/)
  integer, dimension(2)             :: gl = (/1,1/)
  real(mk), dimension(2), parameter :: h = (/0.01_mk, 0.01_mk/)
  real(mk), dimension(2)            :: sigma = (/0.1_mk,0.1_mk/)
  real(mk), parameter               :: pi = acos(-1.0_mk)
  global_var(step,<#integer#>,0)

  ppm_init()
  U = create_field(1, "U")
  topo = create_topology(bcdef)
  mesh = create_mesh(topo,offset,h=h,ghost_size=gl)
  add_patch(mesh,[<#0.0_mk#>,<#0.0_mk#>,<#1.0_mk#>,<#1.0_mk#>])
  discretize(U,mesh)
  foreach n in equi_mesh(mesh) with sca_fields(U) indices(i,j) stencil_width(1,1)
    for real
      U_n = 1.0_mk/(2.0_mk*pi*sigma(1)*sigma(2)) *                &
      &     exp(-0.5_mk*((((i-1)*h(1)-0.5_mk)**2/sigma(1)**2) + &
      &                  (((j-1)*h(2)-0.5_mk)**2/sigma(2)**2)))
  end foreach
  o, nstages = create_ode([U], mini_rhs, [U=>mesh], eulerf)
  step = step + 1
  t = timeloop()
```

```
    do istage=1,nstages
      map_ghost_get(mesh)
      ode_step(o, t, time_step, istage)
    end do
  end timeloop
  ppm_finalize()
end client

rhs mini_rhs(U=>mesh)
  real(mk)                  :: h2
  get_fields(dU)
  h2 = product(mesh%h)
  ! calculate Laplacian
  foreach n in equi_mesh(mesh) with sca_fields(U,dU) indices(i,j) stencil_width(1,1)
    for real
      dU_n = (U_n[-1,] + U_n[+1,] + U_n[,-1] + U_n[,+1] - 4.0_mk*U_n)/h2
  end foreach
  step = step + 1
end rhs
```

Listing 2: PPML program to simulate diffusion on a Cartesian mesh using finite differences.

In listing 3 we solve the same governing equation with the same initial and boundary conditions, but using a particle discretization instead of a mesh discretization. In lines 27/28 we use a discretization-corrected particle strength exchange (DC-PSE) operator[16] as a particle approximation to the Laplacian. The main differences with the mesh-based implementation are in the setup, where we create a particle set instead of a mesh (line 12), and in the implementation of the right-hand side. PPM provides routines for defining and applying DC-PSE operators. We hence only need to call these routines.

```
client mini
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  real(ppm_kind_double), dimension(:,:), pointer :: displace
  integer                                        :: istage = 1
  real(mk), dimension(2) :: sigma = (/0.1_mk,0.1_mk/)
  real(mk), parameter    :: pi = acos(-1.0_mk)
  global_var(st,integer,0)

  ppm_init()
  U = create_field(1, "U")
  topo = create_topology(bcdef)
  c = create_particles(topo)
  allocate(displace(ppm_dim,c%Npart))
  call random_number(displace)
  displace = (displace - 0.5_mk) * c%h_avg * 0.15_mk
  call c%move(displace, info)
  call c%apply_bc(info)
  deallocate(displace)
  global_mapping(c, topo)
  discretize(U,c)
  foreach p in particles(c) with positions(x) sca_fields(U)
    U_p = 1.0_mk/(2.0_mk*pi*sigma(1)*sigma(2)) *          &
    &   exp(-0.5_mk*(((x_p(1)-0.5_mk)**2/sigma(1)**2)+((x_p(2)-0.5_mk)**2/sigma(2)**2)))
  end foreach
  map_ghost_get(c)
  n = create_neighlist(c,cutoff=<#2.5_mk * c%h_avg#>)
  Lap = define_op(2, [2,0, 0,2], [1.0_mk, 1.0_mk], "Laplacian")
  L = discretize_op(Lap, c, ppm_param_op_dcpse,[order=>2,c=>1.0_mk])
  o, nstages = create_ode([U], mini_rhs, [U=>c], eulerf)
```

```
  t = timeloop()
    do istage=1,nstages
      map_ghost_get(c,psp=true)
      ode_step(o, t, time_step, istage)
    end do
    st = st + 1
  end timeloop
  ppm_finalize()
end client

rhs mini_rhs(U=>parts)
  get_fields(dU)
  dU = apply_op(L, U)
end rhs
```

Listing 3: PPML program to simulate diffusion using particles and the DC-PSE method[16].

4.2 Benchmarks

We benchmark the PPML language using two example PPML simulations. The first benchmark considers a continuum Gray-Scott reaction-diffusion system[18]. The second simulation implements a molecular dynamics simulation of a Lennard-Jones gas[17] as an example of a discrete particle method. We benchmark both clients on 256 four-way quad-core AMD Opteron 8380 nodes (4096 cores in total) and measure the runtimes, and parallel efficiencies. The nodes are connected with an InfiniBand 4X QDR network. We compile the PPM library and the benchmark clients using the Intel Fortran compiler 13.0.0 with the `-O3` flag. We use OpenMPI 1.6.2, which has native support for InfiniBand.

4.2.1 Simulating a continuous reaction-diffusion model using PPML

Continuous deterministic reaction-diffusion systems model the time and space evolution of the concentration fields of several species reacting with each other and diffusing. Nonlinear reaction-diffusion systems can give rise to steady-state patterns[19].

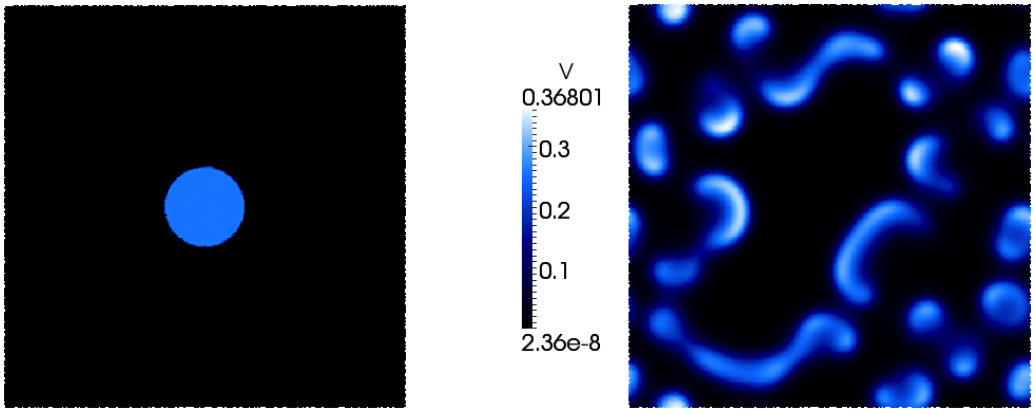
We implement a PPML simulation the Gray-Scott reaction-diffusion model

$$\frac{\partial U}{\partial t} = D_U \nabla^2 U - UV^2 + f(1-U)\,, \qquad \frac{\partial V}{\partial t} = D_V \nabla^2 V + UV^2 - (f+k)V, \tag{1}$$

where D_U and D_V are the diffusion constants of species U and V, and f and k are reaction rates. We simulate the system using a second-order DC-PSE scheme[16].

Figure 4 shows the concentration field of V at time zero and after $4 \cdot 10^4$ time steps of size $\Delta t = 0.05$ for $k = 0.051$, $f = 0.015$, $D_U = 2 \cdot 10^{-5}$, $D_V = 10^{-5}$, and 10^5 particles. As expected for this set of parameters, patterns appear in the concentration of V. The simulation was run on 16 quad-core AMD Opteron 8380 processors, using one core per processor, and took 10 minutes to complete.

We benchmark the PPML reaction-diffusion client on up to 1936 cores. All timings are measured per iteration and averaged over 100 iterations. For a weak scaling (i.e., fixed problem size per processor) the parallel efficiency is defined as $E_p = T_1/T_p$, where



(a) (b)

Figure 4: A PPML Gray-Scott reaction-diffusion particle simulation using the discretization-corrected PSE method[16] with 10^5 particles. (a) Initial condition. (b) Concentration field V after $4 \cdot 10^4$ time steps.

T_1 is the runtime using a single process, and T_p is the time to run a p-fold larger problem using p parallel processes. Figure 5a shows the parallel efficiency of a weak scaling when using all cores of each node. When increasing the number of processes from 4 to 16, the efficiency drops by more than 40%. This drop in efficiency is likely due to a bottleneck in memory access. Therefore, we also test the client's parallel efficiency using only one core per processor (Fig. 5b).

4.2.2 Simulating molecular dynamics using PPML

We use PPML to simulate a discrete particle model of the dynamics of atoms in a gas. The interactions between the atoms are approximated by the Lennard-Jones potential[17]

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{2}$$

where r is the distance between the two atoms, ϵ the potential well depth, and σ is the distance at which the potential is zero. We truncate the Lennard-Jones potential at $r_c = 2.5\sigma$ for computational efficiency and use Verlet lists[14] with a cutoff radius r_c. In order not to recompute the Verlet lists at every time step, we add a 10% "safety-margin" (skin) to the cutoff radius for the Verlet lists. The Verlet lists then only need to be recomputed once any particle has moved farther than half the skin thickness.

The PPML program simulates the forces, potential and kinetic energies, and motions of all atoms (particles). Time integration is done using the velocity Verlet algorithm[14;20]. We validate the implementation by simulating a Lennard-Jones gas with 1 million particles. The particle positions are initialized on a Cartesian mesh with $h \approx 1.5\sigma$. We simulate 14,000 time steps, allowing the gas to equilibrate, and monitor the total and potential energies (E_{tot}, E_{pot}) of the system. Figure 6a shows the results. The total energy is conserved, while the potential energy stabilizes at the equilibrium of the system.

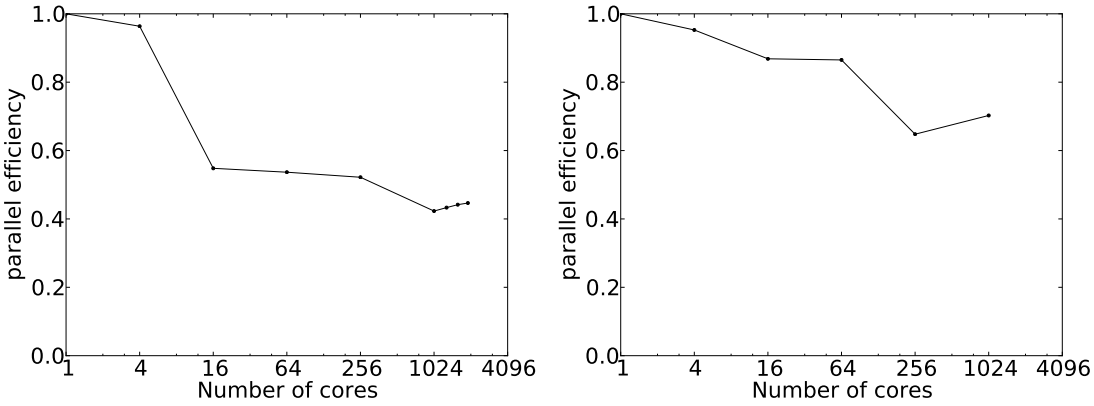


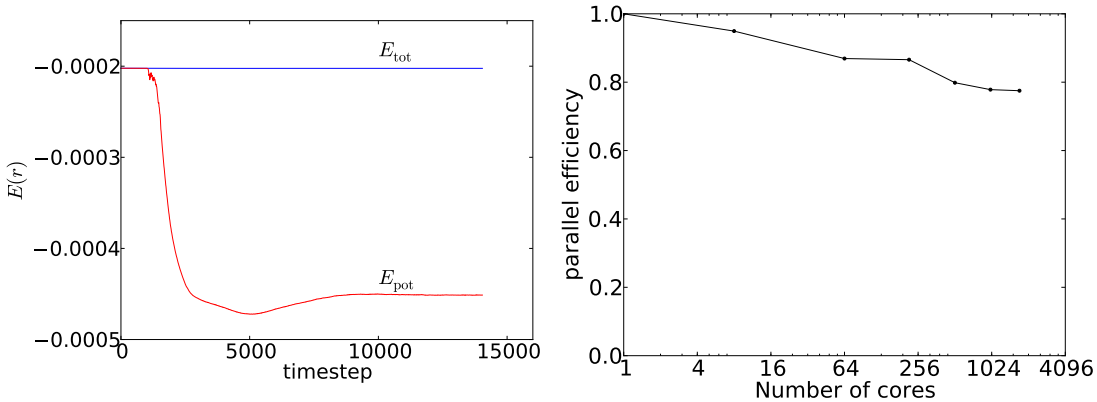
Figure 5: Parallel efficiency for a weak scaling of the PPML reaction-diffusion client (wall-clock time T_1=0.228s). The largest number of cores used was 1936. (a) Using all cores per node leads to a performance drop at 16 cores. (b) The parallel efficiency when using only one core per processor.

Figure 6b shows the results of a weak scaling with 1 million particles per process. All timings are measured per iteration and averaged over 100 iterations. The minimum and maximum times per process were identical, indicating perfect load balance. We use two cores of each processor. Using all cores leads to a loss of performance, similar to what we observed for the reaction-diffusion benchmark.

5 CONCLUSIONS

We have presented PPML, a domain-specific language (DSL) for parallel hybrid particle-mesh simulations. PPML is based on a set of concise abstractions for parallel particle methods[9]. These abstractions are of intermediate granularity and make the communication overhead of a simulation explicit. The PPML language formalizes these abstractions and offers iterators for particles and meshes, a templating mechanism, and program scopes. PPML types, operations, and iterators are implemented using macros that can be modified and extended. The PPML compiler translates PPML code to standard Fortran 2003, which can then be compiled and linked against the PPM library[10;11;12].

We have illustrated PPML in two examples simulating diffusion on both a mesh and a particle discretization. We have benchmarked the efficiency of PPML-generated parallel simulations for both a continuous and a discrete particle method. A DC-PSE reaction-diffusion simulation was written in 70 lines and a Lennard-Jones molecular dynamics simulation in 140 lines. Both simulations natively support shared- and distributed-memory parallelism. Our benchmarks have shown parallel efficiencies of 70% to 80% for both simulations if not all cores per node are used (memory bottlenecks). This shows that it is possible to rapidly implement fully featured parallel simulation codes that scale up to 1000 cores and beyond. PPML allows users with little experience in parallel programming to quickly develop particle-mesh simulations for parallel hardware platforms.



(a) (b)

Figure 6: (a) Total (blue) and potential (red) energy of 1 million particles over 14,000 time steps in a PPML Lennard-Jones simulation. (b) Parallel efficiency of the PPML Lennard-Jones simulation (wall-clock time T_1=1.43s, parallel efficiency 77.5% for 1728 MPI processes simulating 1.7 billion particles).

6 ACKNOWLEDGMENTS

This work was supported by grant #200021-132064 from the Swiss National Science Foundation and by a grant from the Swiss SystemsX.ch initiative, grant LipidX-2008/011.

References

[1] Backus, J.W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. *Proc. Int. Conf. Inf. Process.* (1959).

[2] Logg, A. Automating the finite element method. *Arch. Comput. Method. E.* (2007) **14**:93–138.

[3] Logg, A., Langtangen, H.P. and Cai, X. Past and future perspectives on scientific software. In A. Tveito, A.M. Bruaset and O. Lysne, eds., *Simula Research Laboratory*, pp. 321–362. Springer Berlin Heidelberg (2010).

[4] Logg, A. and Wells, G.N. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.* (2010) **37**.

[5] Houstis, E.N., Gallopoulos, E., Bramley, R. and Rice, J. Problem-solving environments for computational science. *IEEE Comput. Sci. Eng.* (1997) **4**:18 –21.

[6] Houstis, E.N., Rice, J.R., Weerawarana, S., Catlin, A.C., Papachiou, P., Wang, K.Y. and Gaitatzes, M. PELLPACK: a problem-solving environment for PDE-based applications on multicomputer platforms. *ACM Trans. Math. Softw.* (1998) **24**:30–73.

[7] de St. Germain, D.J., McCorquodale, J., Parker, S.G. and Johnson, C.R. Uintah: A massively parallel problem solving environment. In *Proc. HPDC'00: Ninth IEEE*

Intl. Symp. on High Performance and Distributed Computing. IEEE (2000).

[8] DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J. and Hanrahan, P. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proc. 2011 Intl. Conf. Supercomputing*, SC'11. ACM (2011) pp. 9:1–9:12.

[9] Sbalzarini, I.F. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Systems & Technol.* (2010) **1**:40–56.

[10] Sbalzarini, I.F., Walther, J.H., Bergdorf, M., Hieber, S.E., Kotsalis, E.M. and Koumoutsakos, P. PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys.* (2006) **215**:566–588.

[11] Awile, O., Demirel, O. and Sbalzarini, I.F. Toward an object-oriented core of the PPM library. In *Proc. ICNAAM*. AIP (2010) pp. 1313–1316.

[12] Awile, O. *A Domain-Specific Language and Scalable Middleware for Particle-Mesh Simulations on Heterogeneous Parallel Computers*. PhD thesis, Diss. ETH No. 20959, ETH Zürich (2013).

[13] Hockney, R.W. and Eastwood, J.W. *Computer Simulation using Particles*. Institute of Physics Publishing (1988).

[14] Verlet, L. Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys. Rev.* (1967) **159**:98–103.

[15] Awile, O., Büyükkeçeci, F., Reboux, S. and Sbalzarini, I.F. Fast neighbor lists for adaptive-resolution particle simulations. *Comput. Phys. Commun.* (2012) **183**:1073–1081.

[16] Schrader, B., Reboux, S. and Sbalzarini, I.F. Discretization correction of general integral PSE operators in particle methods. *J. Comput. Phys.* (2010) **229**:4159–4182.

[17] Jones, J.E. On the determination of molecular fields. II. from the equation of state of a gas. *Proc. Roy. Soc. Lond. Series A* (1924) **106**:463–477.

[18] Gray, P. and Scott, S.K. Autocatalytic reactions in the isothermal, continuous stirred tank reactor. oscillations and instabilities in the system A + 2B $\rightarrow$ 3B; B $\rightarrow$ C. *Chem. Eng. Sci.* (1984) **39**:1087–1097.

[19] Turing, A.M. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc. London* (1952) **B237**:37–72.

[20] Swope, W.C., Andersen, H.C., Berens, P.H. and Wilson, K.B. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *J. Comput. Phys.* (1982) **76**:637–649.