Diss. ETH No. 20959

A Domain-Specific Language and Scalable Middleware for Particle-Mesh Simulations on Heterogeneous Parallel Computers

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Omar Awile
M.Sc. Computer Science ETH
born on December 4th, 1981
citizen of Kehrsatz BE, Switzerland

Accepted on the recommendation of
Prof. Dr. Ivo F. Sbalzarini, examiner
Prof. Dr. Jens Walther, co-examiner
Prof. Dr. Petros Koumoutsakos, co-examiner
Prof. Dr. Torsten Hoefler, co-examiner

2013

Declaration of Authorship

This thesis is a presentation of my own original research work. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature. The work was done under the guidance of Professor Ivo F. Sbalzarini at ETH Zurich, Switzerland. I hereby declare that this thesis has not been submitted before to any institution for assessment purposes.

Abstract

Alongside theory and experiment, computing has become the third pillar of science. Meeting the increasing demand for computing power, high-performance computer systems are becoming larger and more complex. At the same time, the usability and programmability of these systems has to be maintained for a growing community of scientists that use computational tools.
In computational science, hybrid particle-mesh methods provide a versatile framework for simulating both discrete and continuous models either deterministically or stochastically. The parallel particle mesh (PPM) library is a software middleware providing a transparent interface for particle-mesh methods on distributed-memory computers.
This thesis presents the design and implementation of algorithms, data structures, and software systems that simplify the development of efficient parallel adaptive-resolution particle-mesh simulations on heterogeneous hardware platforms. We propose a new domain-specific language for parallel hybrid particle-mesh methods, the parallel particle mesh language (PPML). This language provides abstract types, operators, and iterators for particle-mesh methods, using the PPM library as a runtime system. We also present a graphical programming environment, called webCG, that allows rapid visual prototyping of PPML programs from any web browser. These developments are accompanied by several extensions to the PPM library itself. We redesign the PPM library core following an object-oriented paradigm. This allows directly representing abstract types and operators in PPM, which greatly simplifies the runtime support for PPML. A number of extensions address the use of PPM on heterogeneous multi- and many-core platforms, and for adaptive-resolution particle simulations. This first includes a Fortran 2003 POSIX threads wrapper library, extending PPM to hybrid multi-processing/multi-threading environments. Second, we present a generic algorithm for 2D and 3D particle-mesh interpolation on streaming multi-processors, and a portable OpenCL implementation thereof. We benchmark this implementation on different general-purpose GPUs and compare its performance with that of sequential and OpenMP-parallel versions. This extends the PPM library to transparently support GPU acceleration. Third, we present a new communication scheduler based on graph vertex-coloring. We assess the asymptotic runtime and perform bench-

marks on graphs of varying sizes and adjacency degrees, outperforming PPM's previous communication scheduler in all cases. Fourth, we present a novel neighbor-finding algorithm for adaptive-resolution particle methods. Adaptive-resolution neighbor lists retain an asymptotic runtime of $O(N)$, albeit at the cost of $O(N \log N)$ for constructing the data structures. This extends the PPM library to support adaptive-resolution particle simulations.

We demonstrate the ease of use, flexibility, and parallel scalability of the new PPM design and of PPML in two example applications. The first one considers a continuum particle method for reaction-diffusion simulations. The second application is a discrete Lennard-Jones molecular dynamics simulation. The first application amounts to a mere 70 lines of PPML code and sustained 75% parallel efficiency on 1936 cores with less than half a second of wall-clock time per time step. The second application consists of 140 lines of PPML code and achieved 77% efficiency on 1728 cores.

Zusammenfassung

Computing hat sich neben Theorie und Experiment zur dritten Säule der Wissenschaft entwickelt. Um der steigenden Nachfrage nach Rechenleistung gerecht zu werden, werden Hochleistungsrechensysteme immer grösser und komplexer. Gleichzeitig muss die Bedienbarkeit und Programmierbarkeit dieser Systeme für eine wachsende Anzahl von Wissenschaftlern erhalten bleiben.

Partikel-Gitter Methoden bieten ein vielseitiges Framework zur deterministischen oder stochastischen Simulation sowohl diskreter wie auch kontinuierlicher Modelle. Die parallel particle mesh (PPM) Softwarebibliothek bietet eine transparente Schnittstelle für Partikel-Gitter Methoden auf verteilten Rechnern an. Diese Dissertation befasst sich mit dem Design und der Implementation von Algorithmen, Datenstrukturen und Software-Systemen, die die Entwicklung von effizienten parallelen adaptive-resolution Partikel-Gitter Simulationen auf heterogenen Hardwareplattformen ermöglichen. Wir stellen eine neue Domain-spezifische Sprache, parallel particle mesh language (PPML), für parallele Partikel-Gitter Methoden vor. Diese Sprache bietet abstrakte Typen, Operatoren und Iteratoren für Partikel-Gitter Methoden an und benutzt die PPM Bibliothek als Runtime-System. Wir präsentieren auch eine grafische Programmierumgebung, genannt webCG, die graphisches rapid-Prototyping von PPML Programme auf jedem Web-Browser ermöglicht.

Diese Entwicklungen werden durch mehrere Erweiterungen der PPM Bibliothek selbst begleitet. Wir überarbeiten die PPM core Bibliothek nach einem objekt-orientierten Paradigma was eine verbesserte Repräsentation von den abstrakten Datentypen und Operatoren in PPM ermöglicht und eine vereinfachte Runtime-Unterstützung für PPML bietet. Eine Reihe von Erweiterungen von PPM befasst sich mit der Unterstützung von PPM für heterogene Multi- und Many-Core-Plattformen und adaptive-resolution Partikel-Simulationen. Erstens, entwickeln wir eine Fortran 2003 POSIX-Threads Wrapper-Bibliothek, die PPM auf Hybrid multi-processing/multi-threading Umgebungen erweitert. Zweitens, stellen wir einen generischen Algorithmus für 2D und 3D Partikel-Gitter Interpolation auf Streaming Multi-Prozessoren und portabler OpenCL Umsetzung vor. Wir testen dessen Implementation auf verschiedenen GPUs und vergleichen dessen Leistung mit den sequentiellen und OpenMP-parallel-Versionen. Drittens, stellen

wir einen neuen auf Graph-Knotenfärbung beruhenden, Kommunikations-Scheduler vor. Wir analysieren die asymptotische Laufzeit und führen Benchmarks auf Graphen verschiedener Größen und Adjazenzgraden aus. Der neue Scheduler übertrifft PPM's vorherigen Kommunikation-Scheduler in allen getesteten Fällen. Viertens, präsentieren wir einen neue Zelllisten-Algorithmus für adaptive-resolution Partikelmethoden. Adaptive-resolution Zelllisten weisen eine asymptotische Laufzeit von $O(N)$ auf, jedoch benötigt der Aufbau der Datenstrukturen $O(N \log N)$ Zeit. Wir erweitern damit PPM's Unterstützung für adaptive-resolution Methoden.
Wir demonstrieren die Benutzung, Flexibilität und die parallele Skalierbarkeit des neuen PPM Designs und der PPML Sprache in zwei Anwendungsbeispielen. Die erste Anwendung, ist eine kontinuierliche Partikelmethode zur Simulation von Reaktions-Diffusions System. Die zweite Anwendung ist eine diskrete Lennard-Jones Molekulardynamik-Simulation. Die erste Anwendung ist in lediglich 70 Zeilen PPML-Code implementiert und erreicht 75% parallele Effizienz auf 1936 Prozessor-Cores mit weniger als einer halben Sekunde Laufeit pro Zeitschritt. Die zweite Anwendung besteht aus 140 Zeilen PPML Code und erreicht 77% paralleler Effizienz auf 1728 Cores.

Acknowledgements

First and foremost, I thank Ivo for his supervision during my PhD studies. His enthusiasm, optimism and perseverance have always been an inspiration to me. I thank Ivo for the trust and friendship he offered to me at the beginning of this journey and ever since. I truly enjoyed working with Ivo and look back to many fond memories of my time in the MOSAIC group.

I feel privileged to have been part of the MOSAIC group, a place that is far more than the sum of its parts. Working with this exceptional group of people gave me the opportunity to learn many much broader skills than my PhD studies. It was an honor and great joy to work together with Sylvain. I admire his math skills and patience! Thanks go also to Rajesh, my first office-mate, who was always there for a quick chat or a good laugh. Birte, Jo, and Christian, the first MOSAICians, I thank them for their collegiality and advice that helped me on many occasions. I learned a lot from Greg's broad knowledge and always enjoyed our discussions, thank you! Thanks also to Ömer with whom I wrote many lines of code in the first year of our PPM adventure :). Ferit deserves special mention for his unresting commitment to our projects and his good spirits - even at 3 in the morning. Finally, I am especially thankful to Janick. He has not only been a great colleague but also a friend who has made these last months of our PhDs, despite the pressure we were facing, always fun. It is thanks to him that I kept my sanity :). I'm glad we could share that office up in the attic.

To my best friends Yves and Fred: Thank you for your friendship, for the good times, and for your many advices. You guys rock in so many ways!

Tamara, thank you for being my companion, friend, love. I cannot begin to express how much I value you. I have learned, through your relentless efforts to make things right and to make the people in your life happy, to become a better human and for this I thank you. I consider myself immensely lucky to be part of that small team that we are and I look forward to conquering the world together with you!

Above all, I would like to thank my family. Thank you for providing me with everything I needed to be able to get to this point and for teaching me the most important lesson of my life: I can achieve anything I dream of, if I just try hard enough. Thank you.

Zurich, December 2012

Omar

Contents

Nomenclature

API Application programming interface

APU Accelerated processing unit

AST Abstract Syntax Tree

CPU Central processing unit

CUDA Compute Unified Device Architecture

DOM Document Object Model

DSL Domain-Specific Language

eDSL embedded domain specific language

FPGAs Field-programmable gate array

GPGPU General purpose graphical processing unit

MPI Message passing interface

NUMA Non uniform memory access

OpenCL Open Computing Language

OpenMP Open Multiprocessing

POSIX Portable Operating System Interface, a set of standards specified by IEEE to allow for operating system compatibility.

PPML Parallel Particle-Mesh Language

SDK Source development kit

SIMD Single instruction, multiple data

SPMD Single program multiple data, parallelization technique for example employed in MPI

SVG Scalable Vector Graphics

TCP Transmission control protocoll

VTK Visualization Toolkit

Introduction

In the past decades, computational science has advanced from a supporting role in engineering and physics to an important pillar of scientific innovation. Today, computing is an integral tool complementing scientific theory and experiment. Computing enables us to explore systems that are difficult or impossible to study experimentally. Examples include star formation in astrophysics, global climate models in geophysics, and biomolecule folding and interactions in biology. Computer simulations are by now ubiquitous in the sciences, including the social sciences [Michel et al., 2011]. In engineering, computation in general and simulations in particular have become essential parts of product design and testing.

As a result, the demand for larger, more powerful and more capable computer systems has been steadily increasing. Modern computer systems, however, are increasingly complex with several levels of parallelism and memory hierarchy. This is particularly true in high-performance computing. The peak performance of the fastest supercomputer has increased from

41 TFlop/s in November 2002[1] to 27 PFlop/s in November 2012[2] and is expected to reach the exascale by 2018.

The introduction of multicore processors in 2005, and more recently the use of graphics processing units (GPUs) for general purpose computing, have enabled a continuous increase in computing power [Asanovic et al., 2009, Geer, 2005]. This, however, comes at the cost of reduced programmability. Moreover, memory capacity is growing faster than memory bandwidth. This is further aggravated by the fact that several processor cores often share the same memory bus. To harness the full power of today's fastest computers, intimate knowledge of distributed- and shared-memory parallelism and heterogeneous computer architectures is required. Sbalzarini [2010] summarized these developments with the terms *performance gap*, and *knowledge gap*. The performance gap is the decreasing fraction of sustained performance of simulation codes over the theoretical peak-performance of the hardware. The knowledge gap is the increasingly specialized knowledge required for efficient use of high-performance computers.

Much work has been done to address the performance and knowledge gaps [Asanovic et al., 2009]. A common approach to improving the sustained performance of computer simulation codes, and simplifying the use of heterogeneous parallel hardware platforms, is to introduce abstraction layers. These abstraction layers may be implemented as software libraries or as domain-specific languages (DSL).

Numerous libraries provide abstractions for parallel infrastructure, parallel algorithms, and parallel numerical methods. Examples of low-level infrastructure libraries include AMPI [Chao et al., 2007] providing dynamic load balancing and multithreading within the message passing interface [MPI Forum 2012], pthreads [POSIX, 2004], a standard library for shared-memory parallelism, and the parallel utilities library (PUL) [Chapple and Clarke, 1994]. Mid-level libraries such as the parallel particle mesh (PPM) library [Sbalzarini et al., 2006a, Awile et al., 2010], a portable middleware for the parallelization of hybrid particle-mesh methods, provide abstractions to a class of numerical methods, hiding the parallel implementation behind a transparent API, while allowing for flexibility within the framework of

[1] Earth-Simulator, `http://www.top500.org/system/167148`
[2] Titan, `http://www.top500.org/system/177975`

the numerical method. Other examples include PETSc [Balay et al., 2004] providing algorithms and data structures for solving partial differential equations on parallel hardware platforms, Trilinos [Heroux et al., 2005], and POOMA [Reynders et al., 1996, Karmesin et al., 1998], a parallel C++ template library for particle and mesh methods. High-level abstraction libraries provide implementations of entire numerical solvers. Examples include FFTW, PARTI [Moon and Saltz, 1994], a library for Monte-Carlo simulations, and pCMAlib [Mueller et al., 2009], a numerical optimization library.

Another approach to providing abstractions is by means of a domain-specific language (DSL) [Mernik et al., 2005]. For instance, OpenMP and OpenCL are domain-specific extensions to the C programming language that, along with their runtime libraries, allow the user to write portable shared-memory-parallel code. Distributed-memory parallelism is provided by C/C++-like languages and DSLs such as unified parallel C [UPC, 2005], Charm++ [Laxmikant and Gengbin, 2009], an object-based parallel programming system, Linda [Carriero and Gelernter, 1989], and X10 [Charles et al., 2005], a parallel programming language using the partitioned global address space (PGAS) model.

These DSLs provide tools for writing parallel applications with little or no restrictions on the numerical method. The DSLs provided through FEniCS [Logg, 2007, Logg and Wells, 2010], Liszt [DeVito et al., 2011], and DUNE [Dedner et al., 2010], however, limit the programmer to a certain class of numerical methods. In return, they offer a higher level of abstraction, providing a simpler parallel programming model, or entirely hiding parallelism.

In this thesis we present a domain-specific language for parallel hybrid particle-mesh methods using the PPM library as a runtime system [Sbalzarini et al., 2006a, Sbalzarini, 2010]. Furthermore, we extend the PPM library in several ways to include support for multi- and many-core hardware platforms, as well as adaptive-resolution particle methods. The thesis is structured as follows:

Part I: Preliminaries

We start by briefly recapitulating the key concepts of hybrid particle-mesh methods and of the PPM library. Focusing on continuum particle methods, we summarize function and operator approximation using particles, and briefly describe hybrid particle-mesh methods.

The PPM library provides abstractions for parallel particle-mesh methods. We discuss the library's overall design and describe *particles*, *meshes*, *topologies*, and *mappings*, the main abstraction types and operations implemented in PPM. We conclude Part I by surveying some previous simulations and applications that have used the PPM library.

Part II: Extending PPM

In Part II we describe the several extensions to the PPM library that we have designed and implemented as part of this thesis.

One possible approach to simplifying the design of software is by encapsulation, particularly by using abstract data types. In chapter 3 we describe a new design of the PPM library using Fortran 2003 object-oriented language features. We implement the PPM abstractions [Sbalzarini, 2010] as derived types. Operation abstractions, such as PPM mappings, are implemented as type-bound procedures. Furthermore, we extend the PPM library with several new utilities. This includes VTK file output for particles and meshes and a configuration-file and runtime-argument processing framework that allows easy creation and processing of runtime parameters for PPM client applications. A third new utility implements Peano-Hilbert curve sorting for improved memory-locality of particles. We present benchmarks comparing a client application written for the original version of PPM with an updated client using the new PPM design.

The inherent adaptivity of particle methods is particularly appealing when simulating multiscale models or systems that develop a wide spectrum of length scales. Evaluating particle-particle interactions using neighbor-finding algorithms such as cell lists [Hockney and Eastwood, 1988] or Verlet lists [Verlet, 1967], however, quickly becomes inefficient in adaptive-

resolution simulations where the interaction cutoff radius is a function of space. Chapter 4 presents a novel adaptive-resolution cell list algorithm and the associated data structures that provide efficient access to the interaction partners of a particle, independent of the (potentially continuous) spectrum of cutoff radii present in a simulation. We characterize the computational cost of the proposed algorithm for a wide range of resolution spans and particle numbers, showing that the present algorithm outperforms conventional uniform-resolution cell lists in most adaptive-resolution settings.

The problem of finding an optimal communication schedule between processors can be abstracted as a graph-coloring problem. In chapter 5 we propose a new communication scheduler based on the DSATUR vertex-coloring algorithm [Brélaz, 1979] and present its implementation. We benchmark the implementation on a number of graphs of different sizes and adjacency degrees, and compare it with Vizing's algorithm, which was previously used in PPM. The new implementation shows an overall improved runtime performance and lower computational complexity than Vizing's algorithm.

In section 6.1 we present the design and use of a Fortran 2003 wrapper for POSIX threads, called forthreads. Forthreads is complete in the sense that it provides native Fortran 2003 interfaces to all pthreads routines where this is possible. We demonstrate the usability and efficiency of forthreads for SIMD parallelism and task parallelism. We present forthreads/MPI implementations that enable hybrid shared- and distributed-memory parallelism in Fortran 2003. Our benchmarks show that forthreads offers performance comparable to that of OpenMP, but better thread control and more freedom. We demonstrate the latter by presenting a multi-threaded Fortran 2003 library for POSIX internet sockets, enabling interactive PPM simulations with run-time control.

A key component of hybrid particle-mesh methods is the forward and backward interpolation of particle data to mesh nodes. These interpolations typically account for a significant portion of the computational cost of a simulation. Due to its regular compute structure, interpolation admits SIMD parallelism, and several GPU-accelerated implementations have been presented in the literature. In section 6.2 we build on these works to develop a streaming-parallel algorithm for interpolation in hybrid particle-mesh

methods that is applicable to both 2D and 3D and is free of assumptions about the particle density, the number of particle properties to be interpolated, and the particle indexing scheme. We provide a portable OpenCL implementation of the algorithm in PPM and benchmark its accuracy and performance. We show that with such a generic algorithm speedups of up to 15× over an 8-core multi-thread CPU implementation are possible if the data are already available on the GPU. The maximum speedup reduces to about 7× if the data first have to be transferred to the GPU. The benchmarks also expose several limitations of GPU acceleration, in particular for low-order and 2D interpolation schemes.

Part III: A domain-specific language for particle methods

Based on the redesigned PPM library introduced in chapter 3, and the abstractions described by Sbalzarini [2010], we develop an embedded domain-specific language (eDSL) for parallel hybrid particle-mesh methods. This domain-specific language is embedded in Fortran code and provides types, operators, iterators, and type templating for particle-mesh methods. The present parallel particle mesh language (PPML) is implemented using a flexible macro system that allows the user to define custom operations and iterators.

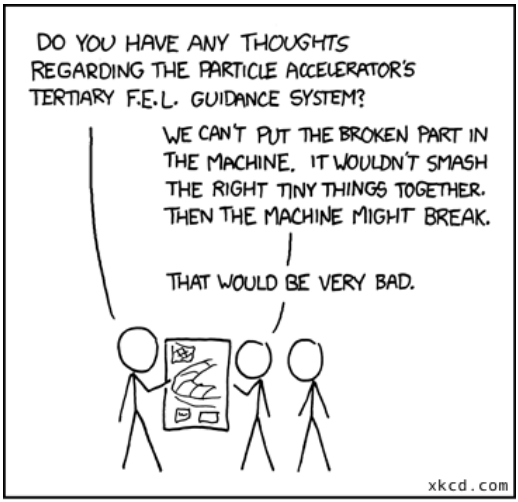
We also present a visual programming environment for PPML. This programming environment is web-browser-based and uses an application server for generating and executing PPML clients. It also provides an interface to monitor running simulations.

We demonstrate the usefulness and scalability of PPML in two example applications. The first application considers a continuum reaction-diffusion simulation. The second application is a discrete Lennard-Jones molecular dynamics simulation. For both clients we perform parallel benchmarks on up to 1936 cores and report parallel efficiencies >75%. Each client can be implemented in just a few hours without requiring special knowledge of parallel programming.

Part I.

Preliminaries

Supported Features



I SPENT ALL NIGHT READING SIMPLE.WIKIPEDIA.ORG, AND NOW I CAN'T STOP TALKING LIKE THIS.

CHAPTER 1

A brief introduction to hybrid particle-mesh methods

Simulations using particles are ubiquitous in computational science and beyond. Particle methods are able to seamlessly treat both discrete and continuous systems either stochastically or deterministically. In discrete particle methods, particles frequently correspond to real-world entities, such as atoms in molecular dynamics simulations or cars in road traffic simulations. In simulations of continuous systems, particles constitute the material points (Lagrangian tracer points) of the system, which evolve according to their pairwise interactions. Examples include the vortex elements in incompressible fluid mechanics simulations [Koumoutsakos, 2005]. Particle methods are intuitively easy to understand and applicable also in situations that cannot be described by (differential) equations, e.g., image segmentation [Cardinale et al., 2012, Bernard et al., 2009].

The dynamics of the simulated system can be described by a set of ordinary differential equations that determine the evolution of the particle positions

and strengths

$$\frac{d\mathbf{x}_p}{dt} = \sum_{q=1}^{N} \mathbf{K}(\mathbf{x}_p, \mathbf{x}_q; \omega_p, \omega_q) \qquad p = 1, ..., N \tag{1.1}$$

$$\frac{d\omega_p}{dt} = \sum_{q=1}^{N} \mathbf{F}(\mathbf{x}_p, \mathbf{x}_q; \omega_p, \omega_q) \qquad p = 1, ..., N. \tag{1.2}$$

K and **F** can be either deterministic (e.g., integral representations of a differential operator, force fields, or generic interaction potentials) or stochastic (e.g., probability density functions).

Hybrid particle-mesh methods combine the strengths of both particle and mesh discretizations. This allows each operation to be done in the more suitable discretization in terms of precision, efficiency, and simplicity. In order to translate between the two discretizations mesh-to-particle and particle-to-mesh interpolations are available.

1.1. Continuum particle methods

In *continuum particle methods* smooth functions are approximated by integrals that are discretized by a set of particles. A particle p is a discrete computational element that is located at a position $\mathbf{x}_p$ and carries strengths ω_p. The particles' *attributes* (location and strengths) evolve so as to satisfy the governing equations of the simulated system in a Lagrangian frame of reference [Koumoutsakos, 2005]. In *pure particle methods*, the functions **K** and **F** arise from the integral approximation of differential operators. In *hybrid particle-mesh methods* some of the differential operators are evaluated on a superimposed regular Cartesian mesh. Particle-to-mesh and mesh-to-particle interpolations are used to translate between the two representations.

1.1.1. Particle function approximation

A smooth function $f(\mathbf{x}) : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ can be approximated using particles by first rewriting it in an integral representation using the Dirac δ-function, then mollifying this integral, and finally discretizing the mollified integral using a quadrature rule.

We use the Dirac δ-function to derive an integral representation of the function f

$$f(\mathbf{x}) = \int f(\mathbf{y})\delta(\mathbf{y}-\mathbf{x})d\mathbf{y} \qquad \mathbf{x},\mathbf{y} \in \Omega. \tag{1.3}$$

The convolution with the δ-function can be interpreted as an evaluation at the precise particle location of $\mathbf{y}$, as done in *point particle methods*. However, in order to approximate smooth functions it is necessary to be able to recover approximations of f at off-particle locations. In order to obtain a smooth approximation for all $\mathbf{y} \in \Omega$, we replace the point particles defined by the Dirac functions with blobs of width ϵ (Fig. 1.1). This is formally done by regularizing the Dirac δ-function with a smooth *mollification kernel* ζ_ϵ of characteristic width ϵ, thus

$$f_\epsilon(\mathbf{x}) = \int f(\mathbf{y})\zeta_\epsilon(\mathbf{y}-\mathbf{x})d\mathbf{y}. \tag{1.4}$$

The mollification kernel is defined as $\zeta_\epsilon = \epsilon^{-d}\zeta(\frac{\mathbf{x}}{\epsilon})$ such that $\lim_{\epsilon\to 0}\zeta_\epsilon = \delta$. The characteristic width ϵ defines the spatial resolution of the method.

The order of accuracy $O(\epsilon^r)$ of the mollification can be conveniently described by the number of moments of the δ-function that the mollification kernel conserves. The kernel is of order r if

$$\int \mathbf{x}^k\zeta(\mathbf{x})d\mathbf{x} \stackrel{!}{=} \int \mathbf{x}^k\delta(\mathbf{x})dx \qquad \forall k \in \{0,...,r-1\}. \tag{1.5}$$

This implies that non-negative, symmetric kernels, such as a Gaussian, can only produce discretizations of order up to $r = 2$ since for moments $k \geq 1$ only $k = 1$ can vanish (for a detailed description see [Cottet and Koumoutsakos, 2000, Koumoutsakos, 2005]).

Finally, the mollified integral representation of $f(\mathbf{x})$ is discretized over N

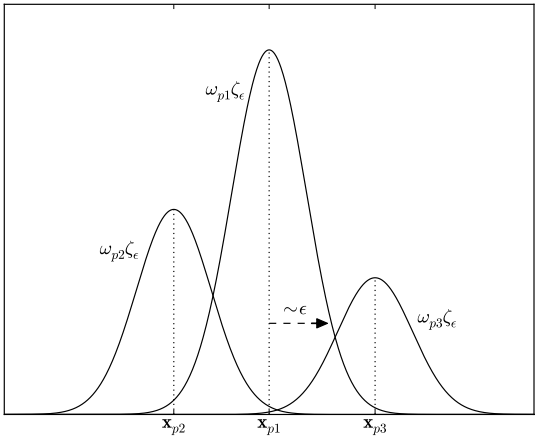


Figure 1.1. Three particles $\mathbf{x}_{p1}$, $\mathbf{x}_{p2}$, and $\mathbf{x}_{p3}$ carrying Gaussian mollification kernels ζ_ϵ with strengths ω_{p1}, ω_{p2}, and ω_{p3}.

particles using the rectangular quadrature rule

$$f_\epsilon^h(\mathbf{x}) = \sum_{p=1}^{N} \omega_p^h \zeta_\epsilon(\mathbf{x}_p^h - \mathbf{x}), \tag{1.6}$$

where h is the particle distance and $\mathbf{x}_p^h$ and ω_p^h are the numerical solutions of particle positions and strengths obtained from discretizing Eqs. 1.1 and 1.2. Since $\omega_p^h = f(y)dy$, it is an extensive quantity and depends on the quadrature rule employed. Here, we use the rectangular rule yielding $\omega_p^h = f(\mathbf{x}_p^h)V_p$ where V_p is the particle's volume. The overall accuracy of the discretization is then

$$f_\epsilon(\mathbf{x}) = f(\mathbf{x}) + O(\epsilon^r) + O\left(\frac{h}{\epsilon}\right)^s, \tag{1.7}$$

where s depends on the number of continuous derivatives of the mollification kernel ζ [Cottet and Koumoutsakos, 2000, Koumoutsakos, 2005]. In the

case of a Gaussian kernel $s \to \infty$. From the approximation error in Eq. 1.7 we see that h/ϵ must be < 1 for the discretization error to remain bounded. In other words the particle spacing must be such that particle kernels always overlap (Fig. 1.1).

1.1.2. Operator approximation

In *pure particle methods* discretized differential operators are constructed either by differentiating the smooth particle function approximation as used in the *smoothed particle hydrodynamics* (SPH) method [Gingold and Monaghan, 1977] or by finding integral operators that are equivalent to the corresponding differential operators. In *hybrid particle-mesh methods* some, but not all, differential operators are evaluated on a regular Cartesian mesh carrying the corresponding intensive quantities of the discretized function.

1.1.2.1. Pure particle methods

As an example of a pure particle method for discretizing a differential operator, we detail the *particle strength exchange* (PSE) method, that was first introduced by Degond and Mas-Gallic [1989a]. PSE can be used to construct a discretized integral operator from a differential operator. For example the PSE operator for the Laplacian (∇^2) can be derived starting from the Taylor expansion of f around $\mathbf{x}$:

$$f(\mathbf{y}) = f(\mathbf{x}) + (\mathbf{y}-\mathbf{x})\cdot\nabla f(\mathbf{x}) + \frac{1}{2}\sum_{i,j}(x_i - y_i)(x_j - y_j)\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} + \cdots . \quad (1.8)$$

This expression is then convolved with a kernel function η_ϵ

$$\begin{aligned}\int (f(\mathbf{y}) - f(\mathbf{x}))\,\eta_\epsilon(\mathbf{y}-\mathbf{x})d\mathbf{y} = &\int (\mathbf{y}-\mathbf{x})\nabla f(\mathbf{x})\eta_\epsilon(\mathbf{y}-\mathbf{x})d\mathbf{y} \\ &+\frac{1}{2}\int \sum_{i,j}(x_i - y_i)(x_j - y_j)\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\eta_\epsilon(\mathbf{x}-\mathbf{y}) \\ &+\cdots\end{aligned} \quad (1.9)$$

that is designed to match the desired differential operator. The moments of η_ϵ must be such that the differential operator is isolated from the Taylor expansion. In case of the Laplacian we must hence choose a kernel function that fulfills:

- η_ϵ is even, canceling out all odd terms in Eq. 1.9
- $\eta_\epsilon(\mathbf{x}) = \epsilon^{-d}\eta\left(\frac{x}{\epsilon}\right)$ and $\int z_i^2\eta(\mathbf{z})d\mathbf{z} \stackrel{!}{=} 2 \quad \forall i \in \{1, ..., d\}$, normalizing the remaining second derivative in Eq. 1.9
- $\int \mathbf{z}^\mathbf{k}\eta(\mathbf{z})d\mathbf{z} \stackrel{!}{=} 0 \quad \forall k \in \{3, ..., r+1\}$, canceling out all remaining higher order terms up to order $r + 1$

Applying such an η_ϵ, and solving for $\frac{\partial^2 f(\mathbf{x})}{\partial x_i^2}$ leads to the desired approximation of the Laplacian operator

$$\nabla_\epsilon^2 f(\mathbf{x}) = \epsilon^{-d} \int \left(f(\mathbf{y}) - f(\mathbf{x})\right) \eta_\epsilon(\mathbf{y} - \mathbf{x})d\mathbf{y}. \tag{1.10}$$

Finally, the approximated integral operator can be discretized over the particles using a quadrature rule. The PSE method has recently been extended by Schrader et al. [2010] with a discretization correction framework (DC PSE) that allows for high-resolution discretizations with full convergence rate. DC PSE operators hence provide the required convergence rate for irregular particle distributions and over the entire spectrum of resolutions.

Evaluating operators on particles requires $O(N^2)$ operations. However, if the operator have local support, auxiliary data structures such as cell lists [Hockney and Eastwood, 1988] or Verlet lists [Verlet, 1967] can be used to reduce the runtime to $O(N)$. We present in chapter 4 an extension of cell lists that remains efficient also for adaptive-resolution particle methods where the cutoff radii may range over several orders of magnitude. For long-range interactions, approximation algorithms such as Barnes-Hut [Barnes and Hut, 1986] or multipole expansions [Greengard and Rokhlin, 1988] may be used to reduce the time complexity to $O(N \log N)$ or $O(N)$ respectively. Hybrid particle-mesh methods with FFT or multigrid based mesh solvers can be used. Such methods offer runtime complexities of $O(M \log M)$ and $O(M)$ (for M mesh nodes) respectively, however, often with lower prefactors than pure-particle methods.

1.1.2.2. Hybrid particle-mesh methods

Hybrid particle-mesh methods use regular Cartesian meshes to efficiently evaluate long-range differential operators. This is for example done in Particle in Cell (PIC) methods as proposed by Harlow [1964] and in P3M (particle-particle-particle-mesh) algorithms [Hockney and Eastwood, 1988]. Non-uniform and unstructured meshes are not considered in hybrid particle-mesh methods, as sub-grid scales are represented on the particles. Therefore, "translations" between particle and mesh discretizations are needed. This is usually done using moment preserving particle-mesh interpolations [Hockney and Eastwood, 1988, Monaghan, 1985] provide the required accuracy and can be implemented efficiently.

1.1.3. Particle-mesh interpolation

Hybrid particle-mesh methods rely on accurate and efficient interpolation of particle properties or *strengths* to the nodes of a *(block-wise) uniform Cartesian* mesh, and back [Hockney and Eastwood, 1988]. We refer to the former as "particle-to-mesh interpolation" and the latter as "mesh-to-particle interpolation". In particle-to-mesh interpolation, one aims at exact conservation of the first few moments of the interpolated function. This ensures conservation of mass, momentum, angular momentum, etc., depending on the order of the interpolation scheme. In mesh-to-particle interpolation, conservation of moments is generally not possible due to the non-uniform spacing of the target particles. In both cases however, the interpolation error decreases as a power of the mesh spacing h. This power is called the *order of convergence* of the interpolation scheme.

We consider moment-conserving interpolation schemes in two and three dimensions that are Cartesian products of one-dimensional interpolants. The weights W of the one-dimensional interpolants are computed independently for each dimension i using the *mesh distance* $\mathbf{s}$ between the mesh

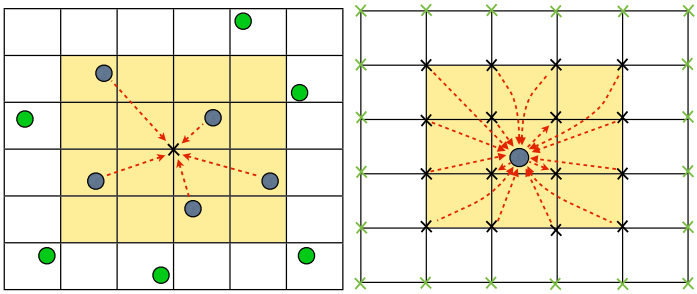


Figure 1.2. Schematic of (a) particle-to-mesh and (b) mesh-to-particle interpolation in 2D using an interpolation function with support $\pm 2\mathbf{h}$ (support region is shaded in yellow). Blue particles ($\mathcal{P}(m)$) and mesh nodes ($\mathcal{M}(p)$) are within the support region of the center node/particle and hence assign onto it. Green particles and nodes lie outside the support and are not considered.

node m and the particle p, $\mathbf{s}(m,p)$:

$$W_i(m,p) = \phi(s_i(m,p)) \tag{1.11}$$

$$= \phi\left(\frac{|x_i(m) - x_i(p)|}{h_i}\right), \tag{1.12}$$

where ϕ is the one-dimensional interpolation function, $x_i(m)$ is the i-th component of the position of mesh node m, and $x_i(p)$ the i-th component of the position of particle p.

The final interpolation weight in d dimensions is computed as:

$$W(m,p) = \prod_{i=1}^{d} W_i(m,p). \tag{1.13}$$

Particle-to-mesh interpolation is then formulated as follows:

$$\vec{\omega}(m) = \sum_{p \in \mathcal{P}(m)} W(m,p)\vec{\omega}(p)\,, \tag{1.14}$$

where $\mathcal{P}(m)$ is the set of particles that contribute to mesh node m, i.e., are within the support of the interpolation weights W from the mesh node m (see Fig. 1.2(a)), and $\vec{\omega}$ is the value (particle property) that is being interpolated.

Likewise, mesh-to-particle interpolation is formulated as:

$$\vec{\omega}(p) = \sum_{m \in \mathcal{M}(p)} W(m,p)\vec{\omega}(m)\,, \tag{1.15}$$

where $\mathcal{M}(p)$ is the set of mesh nodes that contribute to particle p (see Fig. 1.2(b)).

The choice of interpolation function ϕ determines the order of convergence and the number of conserved moments. Here, we consider the M_4' (introduced by Monaghan as the W_4 function [Monaghan, 1985]) and the linear interpolation schemes:

$$\phi_{M_4'}(s) = \begin{cases} \frac{3}{2}s^3 - \frac{5}{2}s^2 + 1 & ,0 \leq s \leq 1 \\ -\frac{1}{2}s^3 + \frac{5}{2}s^2 - 4s + 2 & ,1 < s \leq 2 \\ 0 & ,\text{else.} \end{cases} \tag{1.16}$$

$$\phi_{\text{linear}}(s) = \begin{cases} 1 - s & ,0 \leq s \leq 1 \\ 0 & ,\text{else.} \end{cases} \tag{1.17}$$

The order of convergence is 3 for M_4' and 2 for linear interpolation. M_4' conserves moments up to and including the second moment, whereas the linear interpolation scheme conserves moments up to and including the first moment. Linear interpolation is sometimes also termed "Cloud-in-Cell" (CIC) interpolation [Hockney and Eastwood, 1988]. An SIMD-efficient evaluation of the interpolation polynomials can be done using the Horner scheme [Rossinelli et al., 2011].

CHAPTER 2

The Parallel Particle Mesh Library

Applications of particle methods are numerous, covering simulations of continuous systems using methods such as vortex methods (VM) [Koumoutsakos, 2005] or smooth particle hydrodynamics (SPH) [Gingold and Monaghan, 1977] and simulations of inherently discrete systems such as molecular dynamics (MD) or gravitational bodies in astrophysical simulations. Particle-mesh methods additionally employ one or several meshes, allowing the use of efficient algorithms for long-range interactions. This includes P3M (particle-particle-particle-mesh) [Hockney and Eastwood, 1988] or Particle in Cell (PIC) methods [Harlow, 1964]. Despite their intuitive formulation and versatility, particle-mesh methods give rise to several difficulties in their efficient parallel implementation. First, the computational domain including the particles and meshes must be decomposed and assigned to processors such that the computational load on and communication between processors is well balanced. Furthermore, the simultaneous presence of particles and meshes presents a single optimal topology. Second, the

processors must communicate particle and mesh updates with their neighbors through a consistent and efficient protocol. Third, particle movement and inhomogeneity may invalidate the domain decomposition and require redecomposing and redistributing particles and/or meshes.

The Parallel Particle Mesh (PPM) library [Sbalzarini et al., 2006a] provides a physics-independent, transparent, and portable interface for implementing particle-mesh methods on parallel distributed-memory computers. The library provides subroutines for adaptive domain decomposition, particle and mesh to processor mapping, particle and mesh interprocess communication, particle-mesh interpolation, and cell [Hockney and Eastwood, 1988] and Verlet [Verlet, 1967] lists. Furthermore, a number of particle-mesh methods have been implemented using these core routines and are provided as part of the library. The library is written in Fortran 95 and makes extensive use of pre-processor macros to provide type overloading for all Fortran numeric types. Distributed-memory parallelism is achieved by transparently using the message passing interface (MPI).

A number of other libraries that provide data and operation abstractions for parallel implementations of numerical methods are available. For example the programming environment for parallelizing finite element and finite volume methods ASTRID [Bonomi et al., 1989], or POOMA [Reynders et al., 1996, Karmesin et al., 1998], a parallel C++ templated library for particle and mesh methods. Finally, OVERTURE [Brown et al., 1997] is an environment for the numerical solution of partial differential equations on adaptively refined meshes.

2.1. Features

The core modules of the PPM library consist of four parts: topologies, mappings, neighbor lists, and particle-mesh interpolations. PPM client applications (clients) use three basic data types: particles, meshes, and connections. Particles are defined by a position $\mathbf{x}_p$ and properties ω_p. Meshes in PPM are always regular, axis-aligned Cartesian meshes. Connections are used to explicitly model particle relationships such as in chemical bonds in MD simulations, triangulated surfaces or unstructured meshes. Based

on PPM's core algorithms and data structures a collection of frequently used numerical routines is also provided. This includes the evaluation of particle interactions based on either cell or Verlet lists, as well as mesh-based Poisson solvers using the Multigrid (MG) or Fast Fourier Transform (FFT). Furthermore, a number of utility routines for timing, parallel I/O, and mathematical operations are provided.

2.1.1. Topologies

In PPM a *topology* is a decomposition of the computational domain into subdomains and the assignment of these subdomains to processors (Figure 2.1, upper-right). Several decomposition types are provided that can be chosen depending on the nature of the simulated problem. All domain decompositions are axis-aligned. Recursive orthogonal bisections are computed using adaptive binary trees. For pencil (or slab) decompositions the programmer chooses one (or two) axis along which the domain is not subdivided. The decomposition algorithm then computes a subdivision along the remaining axes. Cuboid decompositions are computed using adaptive quad- (in 3D oct-) trees. The domain decomposition aims to provide sufficient granularity for load-balancing, while retaining a minimum number of particles per subdomain.

The subdomains of a domain decomposition are assigned to processors using either a greedy algorithm that assigns contiguous subdomain blocks to a processor until the target load has been reached, or by building a subdomain-neighborhood graph and computing a k-way partitioning using the METIS library [Karypis and Kumar, 1998]. The objective of subdomain assignment is to balance the computational cost across processors and to minimize interprocess communication.

Both domain decomposition and subdomain-to-processor assignment are performed by each processor individually, replicating the topology information on all processors. While multiple topologies may exist in a PPM simulation only one may be active at any time. A topology is called active when the data (particles and/or meshes) are distributed according to it.

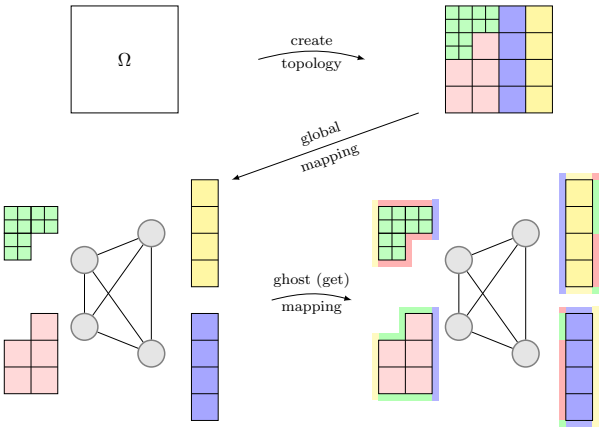


Figure 2.1. In PPM the computational domain Ω (upper-left) is decomposed and subdomains are assigned to processors (upper-right). Particles and meshes are then mapped onto this topology using a global mapping (lower-left). Ghost layers are communicated by ghost mapping (lower-right). In this example periodic boundary conditions are assumed.

2.1.2. Mappings

All interprocess communication in PPM is abstracted and encapsulated by three types of mapping operations: *global mappings*, *local mappings,* and *ghost mappings*. Mappings can be applied to all basic PPM data types: particles, meshes, and connections. Global mappings are used to map data onto a topology, thus activating it. In a global mapping (Figure 2.1, lower-left) each processor first determines which particles, mesh nodes, and connections currently residing on it should be sent to another processor. Then, all processors exchange data according to an optimized communication schedule. Finally, the target topology is marked as active. Local mappings are provided to transfer particles to neighboring processors when they have moved outside of a processor's subdomains. Since meshes do not move, no local mesh mappings are provided. Ghost mappings (Figure 2.1, lower-right) are used to populate and update ghost (halo) layers around each processor's subdomains. Ghost layers allow PPM clients to maintain a local view of the simulation's state. Each processor operates independently of

the others, except at clearly defined synchronization points (the mappings). The ghost layers provide copies (ghosts) of particles and mesh nodes within a predefined distance from interprocess boundaries. The ghost-get mapping updates the ghost values from the data on the neighboring processors, while the ghost put mapping updates real particles and mesh-nodes from their ghost-copies on the neighboring processors.

All mappings internally account for the boundary conditions at the edges of the computational domain. Neumann, Dirichlet, freespace, symmetric, and periodic boundary conditions are supported.

Since all mapping routines in PPM are implemented using *synchronous* MPI communication, a processor can only communicate with one other processor at a time. It is thus beneficial to first optimize the communication schedule. This is done using Vizing's [1964] approximate edge coloring algorithm. In chapter 5 we present a new vertex coloring algorithm with an improved runtime, replacing the previous implementation.

2.1.3. Neighbor lists and particle interactions

The PPM library also provides *cell (linked) lists* [Hockney and Eastwood, 1988] and *Verlet lists* [Verlet, 1967] for efficiently evaluating local particle interactions with constant cutoff radius. Direct particle interactions impose a nominal computational cost of $O(N^2)$. Using cell or Verlet lists, this can be reduced to $O(N)$ on average. PPM's neighbor list routines support both symmetric and asymmetric neighbor lists. In symmetric particle interactions every unique interaction pair is considered only once. PPM uses special symmetric neighbor lists that use 33% (in 3D 40%) less memory and communication compared to standard symmetric neighbor lists [Sbalzarini et al., 2006a]. In adaptive-resolution simulations, the interaction cutoff radius is a function of space. In this case, adaptive algorithms and data structures are needed to efficiently build neighbor lists. In chapter 4 we present fast neighbor lists for adaptive-resolution particle simulations extending PPM.

PPM also offers particle-particle interaction routines that make use of the different neighbor lists and that have several pre-defined particle interaction

kernels already implemented. New kernels can be added into provided source code templates. This ensures efficient execution (vectorization) of the particle interaction loop.

2.1.4. Particle-mesh interpolations

Particle-mesh interpolations provide "translations" between particles and meshes that allow implementing local interactions on particles while using meshes for more efficient implementations of long-range interactions. The PPM library provides highly optimized particle-to-mesh and mesh-to-particle interpolation routines implementing linear and M_4' interpolation schemes (c.f. section 1.1.3). PPM also offers particle remeshing (i.e., particle-to-mesh interpolation followed by particle reinitialization on the mesh) for reinitializing distorted particle distributions. Sections 6.1.2.1 and 6.2 present particle-mesh interpolations on heterogeneous hardware platforms. In section 6.1.2.1 we implement and compare particle-mesh interpolation using POSIX threads as well as OpenMP in order to improve the utilization of multicore hardware platforms. Furthermore, we present in section 6.2 a portable and generic OpenCL implementation of particle-mesh interpolations that is especially suitable for manycore general-purpose GPU platforms.

2.1.5. PPM numerics modules

In addition to implementations of the basic abstractions PPM also provides modules with frequently used numerical solvers. An ODE time integrator module provides explicit multistage integration schemes that can be used by PPM client applications. The user has to provide a callback routine implementing the right-hand side of the governing equation (i.e., the right-hand sides of Eqs. 1.1 and 1.2). This routine is then called by the ODE module at each step of the time loop.

Finally, PPM includes two mesh-based Poisson solvers: a FFT-based solver and a multigrid solver. In section 6.1.2.2 we present a multi-threaded version of PPM's multigrid solver using POSIX threads.

2.2. Previous applications of the PPM library

The PPM library has been used in several applications demonstrating both its usability and parallel efficiency.

Sbalzarini et al. [2006a] presented a particle simulation of diffusion in complex geometries using up to 1 billion particles on 242 processors, attaining 84% parallel efficiency. Furthermore, Sbalzarini et al. [2006a] presented a remeshed smooth particle hydrodynamics (rSPH) code simulating a three-dimensional compressible double shear layer. The largest simulation used 268 million particles and was run on 128 processors at 91% efficiency. An even larger simulation using the PPM library was presented by Chatelain et al. [2008]. Their Vortex-method client achieved 62% efficiency simulating a high Reynolds number aircraft wake using 10 billion particles on a IBM BlueGene/L with 16'384 processors. Walther and Sbalzarini [2009] describe a simulation of granular flow using the discrete element method (DEM) on 192 processors and employing 122 million particles at 40% parallel efficiency. Other examples of applications using PPM include [Adami et al., 2012, 2010] presenting generalized wall boundary conditions and a new surface-tension model for multi-phase SPH, [Milde et al., 2008] simulating a sprouting angiogenesis model, [Bergdorf et al., 2007] simulating vortex rings at high Reynolds numbers, [Chatelain et al., 2007] presenting particle-mesh astrophysics simulations, and [Altenhoff et al., 2007, Bian et al., 2012] presenting dissipative particle dynamics simulations.

2.3. Summary

The PPM library [Sbalzarini et al., 2006a] is a state-of-the-art hybrid particle-mesh library implementing abstractions for transparently handling domain decompositions, interprocess communication, and synchronization between processors. It provides cell and Verlet lists and efficient particle-interaction routines. Finally, it offers mesh-based solvers for efficiently computing long-range interactions, and particle-mesh interpolation routines as needed in hybrid particle-mesh methods. PPM has demonstrated its efficiency in several large-scale simulations [Sbalzarini et al., 2006a, Chatelain

et al., 2008, Walther and Sbalzarini, 2009], while the intermediate granularity of the abstractions simplifies the implementation of parallel hybrid particle-mesh simulations [Sbalzarini, 2010].

In the following chapters we present our work in redesigning and reimplementing PPM's core routines using object-oriented programming techniques in Fortran 2003 (chapter 3). We then describe new neighbor lists for adaptive-resolution particle methods (chapter 4). Chapter 5 introduces a novel vertex-coloring algorithm for communication scheduling. Chapter 6 details the approaches we have taken to allow PPM to make better use of multi- and many-core hardware platforms. Finally, we introduce and describe a domain-specific language (DSL) allowing the user to write PPM clinets in a high-level syntax in terms of the abstractions initially described by Sbalzarini [2010].

Part II.

Extending PPM

Supported Features

CHAPTER 3

Toward an object-oriented PPM core

[1]As parallel scientific software is becoming more complex, the need for more structured software architectures is growing. Many state-of-the-art scientific software libraries implement numerous algorithms targeting different hardware platforms and memory models. Furthermore, libraries typically offer a set of abstractions through their application programming interface (API) that are used and recombined in other libraries. Also, an efficient implementation of a numerical method (or, in general, an algorithm) often necessitates a number of auxiliary data structures and routines. These problems are commonly subsumed under the term *software crisis*, which has been in use since the early 1970s. The introduction of abstraction layers and libraries in parallel and scientific computing has helped alleviate some of the difficulties. Middleware libraries such as the message

[1]This work has been done together with Dr. Sylvain Reboux and Ömer Demirel who have helped redesigning the software architecture and who have contributed much of the new core library code.

passing interface [MPI Forum 2012], FFTW, POOMA [Reynders et al., 1996], FFTW, PETSc [Balay et al., 2004], and the Parallel Particle Mesh library (PPM) [Sbalzarini et al., 2006a] provide abstract interfaces that can be combined to powerful software frameworks. Rouson et al. [2010a] point out that one possible approach to meaningfully structure a software design is to use object-oriented abstract data types. In the past decade, several scientific softwares have appeared that showcased the successful use of object-oriented programming and design patterns to tackle increasingly complex problems. These libraries include Trilinos [Heroux et al., 2005], POOMA [Reynders et al., 1996], and DUNE [Dedner et al., 2010].

We revisit the design of the PPM library, explicitly accounting for the data and operation abstractions described by Sbalzarini [2010]. We furthermore extend the existing *particles* and *mesh* data types to also support adaptive-resolution simulations.

3.1. Overall design

In order to simplify overall design and implementation of the PPM library, and for practical reasons such as code management and compilation time, we split the library into two parts: the PPM core library and the PPM numerics library.

- The *PPM core* library provides the data types and operations required to build PPM client applications. In particular, it provides implementations of particles, meshes, connections, topologies, mappings, particle-mesh interpolations, and neighbor lists (c.f. chapter 2). Furthermore, it contains utility routines for memory management, parallel file I/O, timing, and debugging.

- The *PPM numerics* library contains all numerical routines originally provided by PPM [Sbalzarini et al., 2006a]. This includes time integration schemes, the multigrid and FFT-based Poisson solvers, and various particle interaction kernels.

The PPM numerics library is partially built on top of the PPM core library and uses the PPM abstractions.

We supplement both libraries with a unit testing framework based on `funit`[2]. A unit testing framework greatly reduces the complexity of systematically testing the library and allows the programmer to test functional units in isolation. We extend the testing framework to allow parameterized tests. Hence, we are able to specify test cases over entire parameter ranges, which are systematically checked during testing. The unit testing framework consists of test modules, each being responsible for one PPM module. A testing sequence executes all test cases in a module. Each test module contains an initialization and a finalization routine that are called at the beginning and end of the testing sequence, respectively, and can for example be used to initialize the PPM library. Furthermore, a setup and a teardown routine are specified to be executed before and after each test case, respectively. Test cases are written in Fortran, but are supplemented with assertion statements. These statements are expanded by the unit testing framework into Fortran conditional expressions that collect test successes and failures. The framework reports failed test units and summary statistics (Listing 3.1).

When introducing more complex data structures into numerical and high-performance computing libraries, it is crucial that the design takes data-access into account and does not impose a significant performance toll. This means that function calls and pointer indirections in the body of the main computation loops must be avoided. We account for this by only defining abstract data types of proper granularity. Individual particles or mesh nodes hence have no object representation and are directly stored in simple arrays. Likewise, cell [Hockney and Eastwood, 1988] and Verlet [Verlet, 1967] lists are stored as Fortran arrays and allow direct index access.

Even though the new PPM core is not fully backward compatible, most data types and routines defined in the PPM library's original implementation are either still available or can be accessed as a component of one of the newly defined derived types.

```
>$ FUNITFLAGS="--procs=1,3" make ftest
Makefile:73: Checking for directories...
Makefile:79: done.

============================[ FUNIT STARTED ]=============================

------------------------[ expand test suites ]-------------------------
        ppm_module_ode_typedef                                 regenerated!

-----------------------------[ compile ]-------------------------------
        computing dependencies                                       done!
        locating sources                                             done!
        writing makefile                                             done!
        compiling                                                    done!

============[ Starting test series for 1,3 processor(s) ]=============

==================[ STARTING TEST ON 1 PROCESSOR(S) ]====================
 ppm_module_ode_typedef test suite:
[0] (ppm_init) : *** This is the PPM library starting on inf.ethz.ch
[0] Passed 1274 of 1274 possible asserts comprising 55 of 55 tests.

============================[ SUMMARY ]===================================

        ppm_module_ode_typedef                                       passed

==================[ STARTING TEST ON 3 PROCESSOR(S) ]====================
 ppm_module_ode_typedef test suite:
[0] (ppm_init) : *** This is the PPM library starting on inf.ethz.ch
[1] Passed 902 of 902 possible asserts comprising 55 of 55 tests.
[2] Passed 906 of 906 possible asserts comprising 55 of 55 tests.
[0] Passed 1254 of 1254 possible asserts comprising 55 of 55 tests.

============================[ SUMMARY ]===================================

        ppm_module_ode_typedef                                       passed
```

Listing 3.1 Report generated when running the unit testing framework of the PPM numerics library with one and three MPI processes.

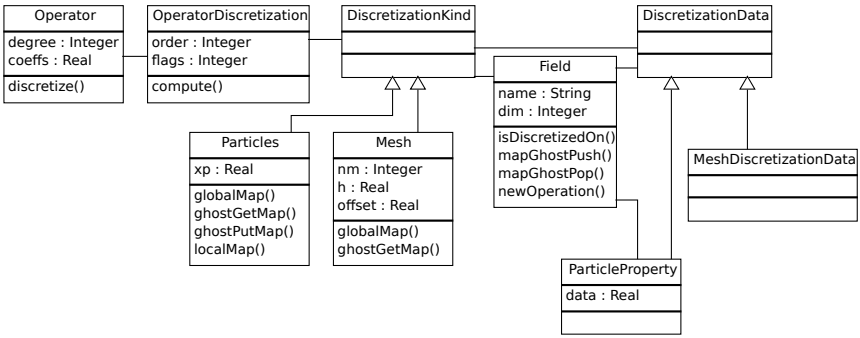


Figure 3.1. The object-oriented design of PPM core. The main abstract data types and their concrete subtypes are shown. We show a subset of the data and methods of implemented in these types.

3.2. The PPM core library

The new PPM core design follows the general idea of Rouson et al. [2010a], which is to structure the library into abstract data types that offer access and manipulation routines. We implement the data types using Fortran 2003 derived types. Figure 3.1 summarizes the main types implemented in the PPM core. As can be seen from the diagram, we have added explicit implementations (in the form of derived types) of *particles* and *meshes*.

In the previous implementation of PPM, particles were represented by their position array `xp` and unrelated property arrays. The newly added type serves as a container for all data and parameters associated with particles. Furthermore, the particles type contains a bookkeeping data structure that tracks whether particles have been mapped onto a topology, have up-to-date ghosts (c.f. chapter 2), and whether their properties have been mapped. We further extend the particles type by subtypes for variable-size particles and self-organizing particles [Reboux et al., 2012]. We also create a generic neighbor list type that encapsulates Verlet lists [Verlet, 1967], adaptive-resolution neighbor lists (c.f. chapter 4), and cross-set neighbor lists. Figure 3.2 shows a simplified class diagram of the particles type and other associated types.

PPM's former mesh data type was only used internally. The user only had a handle to the mesh instance. Also, a mesh instance only contained parameters describing the Cartesian mesh, but no data. The new mesh type combines Cartesian mesh parameters, such as the grid spacing h, a staggering offset, and the ghost layer width with the actual mesh data. Similar to what is done for particles, the mesh type offers generic, type-bound Fortran 2003 procedures for construction and destruction, data access, as well as global and ghost mappings. We have further extended the mesh type with a mesh-patch data structure. Using this new data type it is possible to define mesh patches of different resolutions on top of a coarse background mesh. Mesh data are stored in sub-patches, which are defined as intersections of subdomains (c.f. chapter 2) with patches. Regardless of the extent and resolution of a mesh, only the patches living on it are actually allocated. This enables the implementation of parallel

[2] http://nasarb.rubyforge.org/funit/

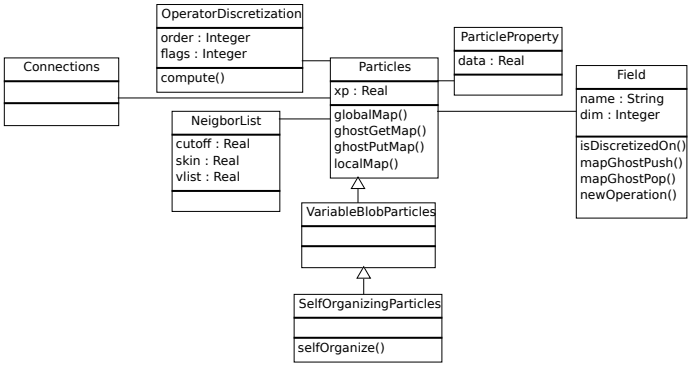


Figure 3.2. Class diagram of the PPM particles type and associated types. We show a subset of the data and methods of implemented in these types.

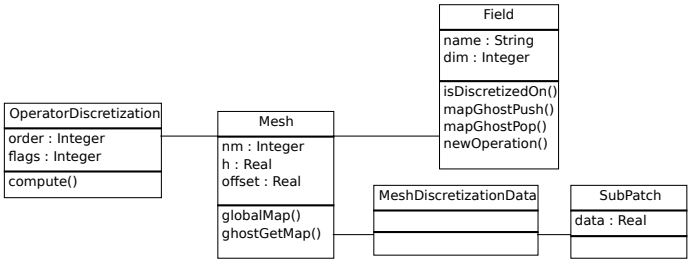


Figure 3.3. The mesh data type and associated types. We show a subset of the data and methods of implemented in these types.

adaptive-resolution hybrid particle-mesh methods [Rasmussen et al., 2011].

In addition to particles and meshes, we have extended the PPM library with two entirely new data types: *field*, and *operator*. Instances of field and operator types encapsulate abstract concepts of a model's governing equations (i.e., Eqs. 1.1 and 1.2). The field type encapsulates the mathematical description of a continuous quantity. Fields can be discretized onto instances of particles or mesh types, instructing the latter instances to internally allocate the necessary memory for a new particle property or mesh data, respectively. The operator type represents differential operators in the governing equations. An operator instance can be discretized onto

one of the two extended types of *DiscretizationKind*, particles, or mesh. *OperatorDiscretization* subtypes implement numerical schemes that allows applying the differential operator to a mesh or particles instance.

3.2.1. Using the new API

Listing 3.2 shows a small example of how PPM's new API is used. In this example, we first create a field U and name it "Concentration". This symbolic name is later used for meaningful log messages and VTK output. We then initialize a particles instance pset. We assume that we have already created a topology so we can use information about domain size and boundary conditions from the topology object when creating the particles. The initialization routine defaults to placing `globalNp` particles on a Cartesian mesh filling the computational domain, but other initialization strategies can be chosen or added. The third command instructs PPM to store in this particles instance a cutoff radius that can later be used to construct neighbor lists. Then, we perform a global mapping of the particles onto the topology designated by the `topoid` handle. After the particles have been mapped onto this topology, we discretize the field U onto them, internally allocating and attaching a new particle property array. We can access and manipulate the data in this array at any time using the field and particles instances. After mapping the particle ghosts, we finally construct a Verlet list on the particles.

3.2.2. New PPM core utilities

[3]We have extended the PPM core library by several useful utility subroutines. Most notably, we created a VTK particle and mesh output routine, a configuration file and command line processing framework, and particle sorting according to space-filling Peano-Hilbert curves.

The Visualization Toolkit (VTK) is an open-source software library and file

[3]We thank Joachim Stadel (University of Zurich) for kindly contributing his C implementation of Cartesian-to-Hilbert-space mapping functions. The configuration file parser and parts of the VTK file I/O routines were implemented by Milan Mitrović.

```
type(ppm_t_field)          :: U
type(ppm_t_particles_d)    :: pset

call U%create(1,info,name="Concentration")
call pset%initialize(globalNp,info,topoid=topoid)
call pset%set_cutoff(3._mk * pset%h_avg,info)
call pset%map(info,global=.true.,topoid=topoid)
call U%discretize_on(pset,info)
call pset%map_ghosts_get(info)
call pset%comp_neighlist(info)
```

Listing 3.2 Creating a field, initializing particles, discretizing the field onto the particles, and performing global and ghost mappings in the new PPM API. Finally, we compute a Verlet list on the particles. The two top lines show the required type declarations.

format for 2D and 3D visualization. It is also the basis of the popular 3D visualization application Paraview. We created a PPM module that allows writing particle and meshdata into VTK-formatted files. When executed in parallel, each processor writes its own VTK-part file, since the VTK format natively supports splitting data across several files. Our implementation makes use of the name strings set by the user when initializing new fields, particles, or meshes.

In order to simplify the creation and processing of configuration files for PPM client applications, and the handling of command-line arguments, we have created a new control file module. This module offers a simple interface for programmatically creating configuration files, including parameter grouping and type checking. Furthermore, configuration parameters can be supplied as command-line arguments that are processed at client initialization. These arguments override configuration file settings.

Lastly, we have added a simple implementation of Peano-Hilbert curve key-sorting for particle positions. Space filling curves can improve the cache locality of spatial data due to their locality properties [Gotsman and Lindenbaum, 1996, Griebel and Zumbusch, 1998, Kowarschik and Weiß, 2003]. Specifically, we have implemented a Fortran wrapper for a fast bit-manipulation C implementation of Peano-Hilbert curve key-sorting. The method first iterates once through all particle positions, determining the particles' positions in Hilbert space and storing the so-obtained linear

key in a temporary array. Then, the particles are sorted according to the Hilbert key. The current implementation provides basic functionality and its correctness has been validated. However, it requires further benchmarking to quantify the impact of space-filling curves on PPM client efficiency.

3.3. The PPM numerics library

We focused on providing an entirely new, object-oriented, flexible, and extensible implementation of PPM's time integration module. Other PPM numerics modules yet have to be ported and tested in order to make full use of the new PPM core data types. They are, however, still functional.

An ODE is defined by creating an instance of the `ode` type and supplying it with a user-defined callback routine implementing the ODE's right-hand side, a list of fields and discretizations to be passed to the right-hand side, and a list of fields and discretizations to be updated by the ODE integration scheme. One then calls the ODE's `step` procedure at each iteration of a simulation time loop.

The `integrator` type is internally used by to the library and is not seen by the user. Each ODE integration scheme must extend `integrator` and implement its `create` and `step` procedures. Thanks to the PPML domain specific language (c.f. chapter 7.3), a new explicit time-integration scheme can be implemented very easily by adding a new subtype and implementing two routines.

3.4. Performance benchmarks

Enforcing data hiding and encapsulation allowed simplifying many of the subroutine interfaces in PPM. The new design of PPM 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 ported two existing PPM clients to the newly designed PPM core library. The first PPM client was

ported to an intermediate version of PPM [Awile et al., 2010] that used the above-described modular design, but did not yet introduce the particles and mesh types as Fortran derived types. This benchmark client solves the diffusion equation in complex, three-dimension geometries using the method of particle strength exchange (PSE) [Degond and Mas-Gallic, 1989b]. This client has previously been tested and used with the original version of PPM [Sbalzarini et al., 2006b,a]. In order to port 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 produced the exact same numerical results on all digits as the original client. We measured the wall-clock time per time step and the parallel efficiency by running the PSE test client on an increasing number of processors using the original PPM and the new core 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 3.4. 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.

The second PPM benchmark client was backported from the final version of the redesigned object-oriented PPM core library to the PPM library at version 1.1[4]. This client performs a molecular dynamics simulation of a Lennard-Jones gas (c.f. section 8.3) Jones [1924], Verlet [1967]. Backporting the client required the addition of 60 calls to the PPM core library. Furthermore, functionality such as automatic error handling and command-line argument handling through the library was lost. We measured the total wall-clock time and the parallel efficiency by running the Lennard-Jones test client on an increasing number of processors using the original PPM and the new object-oriented core implementation. These benchmarks were executed on the same hardware environment and optimization flags as the PSE client benchmarks. The results are shown in Figure 3.5. The object-oriented code is up to 50% slower while at the same time its parallel efficiency is 5% lower than the code using the original PPM core library.

[4]The version before PPM's split into core and numerics libraries

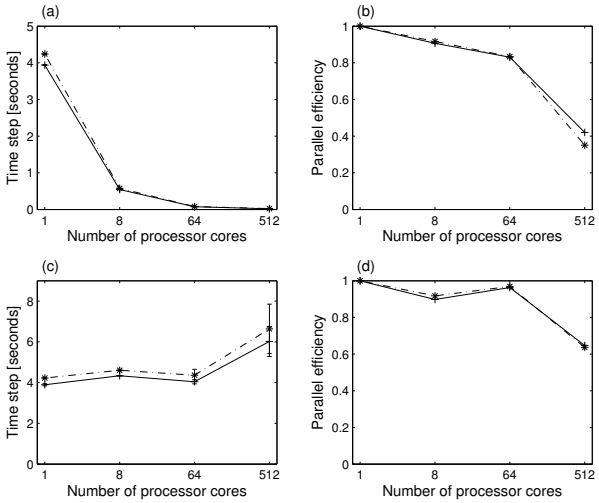


Figure 3.4. Benchmark of a PPM client simulating diffusion in complex shapes. Average (symbols) and standard deviation (error bars) of the maximum (over all processor cores) wall-clock time per time step, sampled over 100 steps, and parallel efficiency for the old (dashed lines) and new (solid lines) PPM core 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.

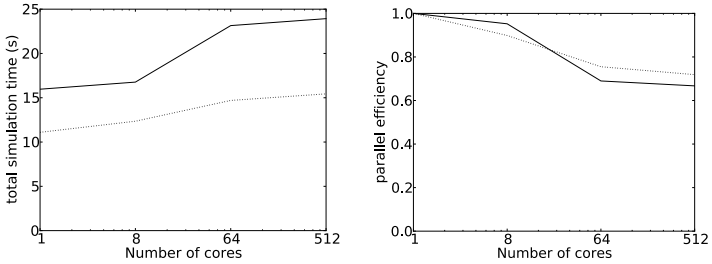


Figure 3.5. Wall-clock time (a) and parallel efficiency (b) of a molecular dynamics simulation benchmark client using the latest object-oriented PPM core library (solid line) and the backported code using the original PPM library (dashed line) on a scaled-size problem with 1 million particles per core (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.

Hence, the simplified interfaces and gained functionality of the new PPM core library come at a non-negligible performance cost. We attribute the decreased parallel efficiency to memory congestion when using all cores of the cluster nodes (c.f. section 8.2) rather than communication, since both versions of the PPM library internally use the same particle mapping routines. In chapter 8 we present further test cases and benchmarks using the latest version of PPM and the domain-specific language PPML.

3.5. Summary and Conclusions

As hardware complexity increases, the requirements in terms of flexibility, maintainability, and usability of scientific and numerical software also increase. It hence becomes increasingly important to design software in a modular and extensible fashion. One possible approach is using object-oriented programming paradigms [Rouson et al., 2010a, Decyk et al., 1998]. We have presented a redesign and partial rewrite of the PPM library [Sbalzarini et al., 2006a]. We have divided the library into two parts, one offering

object-oriented implementations of the abstractions for hybrid particle-mesh methods [Sbalzarini, 2010], the other containing PPM's numerical solvers. The main types provided by the PPM core are particles, mesh, field, operator, and operator discretization. These types encapsulate the data used in hybrid-particle mesh simulations and allow accessing, manipulating, and mapping these data. Furthermore, we have implemented a number of new utility modules that provide parallel VTK file output, configuration file handling, and Peano-Hilbert curve key-sorting. Finally, we have supplemented the PPM core and numerics libraries with a unit testing framework. This allows systematically testing new implementations and prevents code regression.

The new design of PPM represents an incremental step in the development of the library. The presented benchmarks suggest that the new design has some negative effects on the performance of a PPM client. More work is thus required in the PPM core library to achieve better performance as well as an even simpler and cleaner design, for example by using more formal design patterns [Rouson et al., 2010b]. Improved implementations of the Fortran 2003 standard in mainstream compilers, and the adoption of Fortran 2008, will enable further improvements in the implementation of PPM.

CHAPTER 4

Fast neighbor lists for adaptive-resolution particle simulations

[1]The efficient evaluation of pairwise particle–particle interactions is a key component of any particle-based simulation. Formally, a set of N interacting particles defines an N-body problem with a nominal computational cost of $O(N^2)$. In many practical applications, however, the particle–particle interactions have a finite range or are truncated with a certain cutoff radius. This reduces the computational cost to $O(N)$ if each particle can find its interaction partners ("neighbors") in $O(1)$ operations.

For constant cutoff radii, two classic data structures are available to provide fast neighbor lists with $O(1)$ access *per particle*: cell (linked) lists [Hockney and Eastwood, 1988] and Verlet lists [Verlet, 1967]. A cell (linked) list

[1]This work has been done together with Ferit Büyükkeçeci and Dr. Sylvain Reboux. FB has helped designing parts of the algorithm and provided the Fortran implementation in the PPM library. SB helped designing the performance benchmarks.

divides the domain into equisized cubic cells with edge lengths equal to the interaction cutoff radius. Each cell then stores a (linked) list of the indices of all particles inside it. When computing particle–particle interactions, each particle can find its neighbors in $O(1)$ time by searching only over the cell it is in and the immediately adjacent cells. Being in one of the neighboring cells is a necessary condition for any particle to be an interaction partner, but the condition is not sufficient. Cell lists hence are conservative and more interaction partners are considered than actually required (up to $3^4/4\pi \approx 6$ times more for a uniform particle distribution in 3D). This overhead can be avoided at the expense of higher memory consumption when using Verlet lists [Verlet, 1967], where each particle stores an explicit list of the indices of all its interaction partners. Verlet lists rely on intermediate cell lists for their efficient construction and they commonly include a safety margin (called "skin") in order to avoid their reconstruction every time any particle has moved. This implies a tradeoff between the number of interactions that are computed in excess and the frequency of rebuilding the Verlet lists. For certain systems, optimal skin thicknesses can be found [Chialvo and Debenedetti, 1990, 1991, Sutmann and Stegailov, 2006]. Due to the importance and widespread use of cell and Verlet lists, much work has been done to compare and improve their performance [Allen and Tildesley, 1987, Mattson and Rice, 1999, Heinz and Hünenberger, 2004, Yao et al., 2004, Welling and Germano, 2011, in't Veld et al., 2008].

One of the key advantages of particle methods is their inherent adaptivity. In discrete systems, particles are only needed where the corresponding objects are present. In continuous systems, the particles naturally follow the flow map, again restricting computation to where it is required. The adaptive dynamics of particles, however, can lead to the formation of dense particle clusters. In the worst case, a cluster that is smaller than the particle–particle interaction cutoff may contain all the particles. The computational cost of particle methods then deteriorates to $O(N^2)$. This can be avoided by locally adapting the interaction cutoff to the density of particles, leading to *adaptive-resolution* particle methods. Adaptive-resolution methods are required for the efficient simulation of multiscale systems. Hou [1990] and Cottet et al. [2000] provide two examples of adaptive-resolution particle methods for fluid dynamics; the adaptive-resolution smoothed particle hydrodynamics (SPH) method [Shapiro et al., 1996] provides an example from cosmology. In adaptive-resolution simulations, the interaction cutoff is

defined by a unique-valued map $\vec{x} \in \mathbb{R}^d \mapsto r_c(\vec{x}) \in \mathbb{R}^+$. This is in contrast to *multi-resolution* simulations where there can be multiple cutoff radii (resolution scales) at any given location. Adaptive-resolution simulations are related to *range-assignment problems* as studied in theoretical computer science, computational geometry, and communication networks [Clementi et al., 2001], where each particle can have a different cutoff radius. If the interaction cutoff is a function of space and hence varies across particles, uniform-resolution cell lists become inefficient and other fast neighbor lists are required.

A number of algorithms and data structures have been proposed to address this or similar problems. *K-d trees* [Bentley, 1975] are K-dimensional space-partitioning data structures with a wide range of applications in computational geometry and numerical simulations. They allow efficient k-nearest neighbor searches, but do not support search within a given interaction *radius*. *R-trees* [Guttman, 1984, Arge et al., 2004] relax this constraint by allowing neighborhood searches over bounding boxes. They are prominently used in geographic databases. in't Veld et al. [2008] have proposed multi-resolution cell lists for colloidal mixtures in explicit-solvent molecular dynamics simulations. Their approach assumes a finite number of discrete resolution levels, for each of which a separate uniform-resolution cell list is built.

4.1. Adaptive-resolution cell lists

We generalize cell lists to situations where the cutoff radius of the particle–particle interactions is a potentially continuous function of space. Each particle interacts with all other particles within a spherical neighborhood around it. The radius of this neighborhood depends on the location of the center particle. This is most generally modeled by attributing to each particle p its own interaction cutoff radius $r_{c,p}$. We consider the situation where N particles $p = 1, \ldots, N$ are distributed in a cuboidal domain. Boundary conditions and parallelism are handled by decomposing the computational domain into subdomains and extending each subdomain with a halo layer as illustrated in Fig. 4.1. In a parallel domain-decomposition setting, N hence is the number of particles on the local processor. Since the interaction cutoff

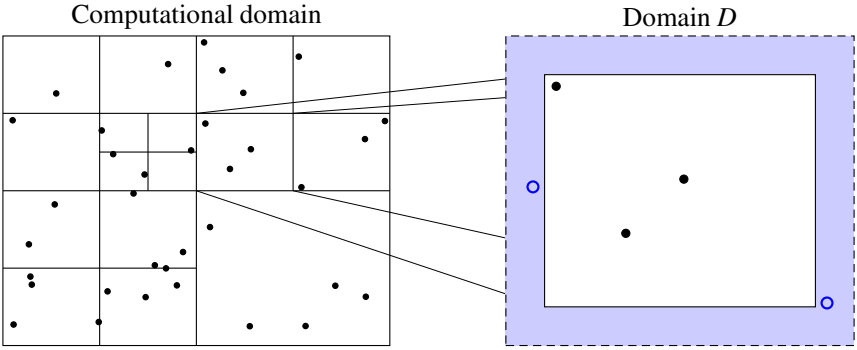


Figure 4.1. The computational domain is decomposed into cuboidal subdomains with halo layers (light blue). The halo layers contain ghost particles (blue circles) that are copies of real particles (black dots) from the adjacent subdomains. Independently applying the present algorithm to each extended subdomain (including the halo layers) D allows transparent implementation of boundary conditions and (distributed-memory) parallelism.

locally changes, the halo layers on different sides of a subdomain may have different widths. Populating the halo layers with ghost particles that are copies of real particles from the adjacent subdomains, and treating boundary conditions by imposing specific values on the ghost particles, is assumed to be done prior to adaptive-resolution (AR) cell list construction. This is typically the case in parallelization frameworks such as the PPM library or PETSc [Balay et al., 2010]. In order to evaluate the particle–particle interactions in any subdomain, only particles within that subdomain and its halo layer need to be considered. We thus build a separate AR cell list for each extended (including the halo layers) subdomain, hereafter referred to as "domain" D (dashed box in Fig. 4.1).

Each particle is defined by its position $\vec{x}_p \in \mathbb{R}^d$ (for $d = 2$ or 3) and its interaction cutoff radius $r_{c,p} = r_c(\vec{x}_p) \in \mathbb{R}^+$. The cutoff radii of neighboring particles may differ by several orders of magnitude and they can take values in a continuum. Two particles are considered neighbors (and hence interact) if

$$||\vec{x}_p - \vec{x}_q|| \leq \min(r_{c,p}, r_{c,q})\,, \tag{4.1}$$

that is, if both are within the interaction radius of the respective other.

Following the nomenclature of Hernquist and Katz [1989], this neighborhood condition defines a *gather*-type sampling of a particle's neighborhood. For *scatter* interactions, the right-hand side in Eq. (4.1) would be replaced with $\max(r_{c,p}, r_{c,q})$, and for collision detection with $r_{c,p} + r_{c,q}$ [De Michele, 2011]. However, we do not consider these two alternative cases since they may require different data structures than the ones presented here.

In AR cell lists, regions containing particles with small cutoff radii ("small particles") are subdivided into small cells, while regions containing particles with large cutoff radii ("large particles") are subdivided into large cells. These cells are defined as the leafs of an adaptive tree (quad-tree in 2D, oct-tree in 3D). Starting from the entire domain D as the root box of the tree, a tree node is subdivided if it contains particles with a cutoff radius smaller than half the edge length of the cell associated with this node (see Fig. 4.2, left panel). The association of particles to cells is computed using an in-place Quicksort-like algorithm. The tree nodes are numbered consecutively per level. Numbers corresponding to empty nodes are skipped (see Fig. 4.2, right panel). This *level-order indexing* of the cell-tree nodes assigns to each tree cell c a unique index $J(c)$ from which it is possible to compute the indices of its neighbor, parent, and child cells in $O(1)$ operations. The resulting cell tree is not stored explicitly, but computed on demand from the particle positions and their levels in the tree.

4.1.1. Constructing AR cell lists

Standard cell lists organize the particles spatially by sorting them into the cells of a uniform Cartesian mesh. In AR cell lists we additionally organize the particles with respect to their cutoff radii using an adaptive tree data structure. A particle's cutoff radius directly relates to the tree level to which the particle is assigned. The construction of AR cell lists is summarized in Algorithm 4.1. This algorithm has two phases:

Phase I: The particles are sorted in order of descending cutoff radii. As this simply amounts to sorting with respect to a scalar property, any efficient sorting algorithm can be used. After the particles have been sorted we determine the tree level each particle belongs to. This starts by computing

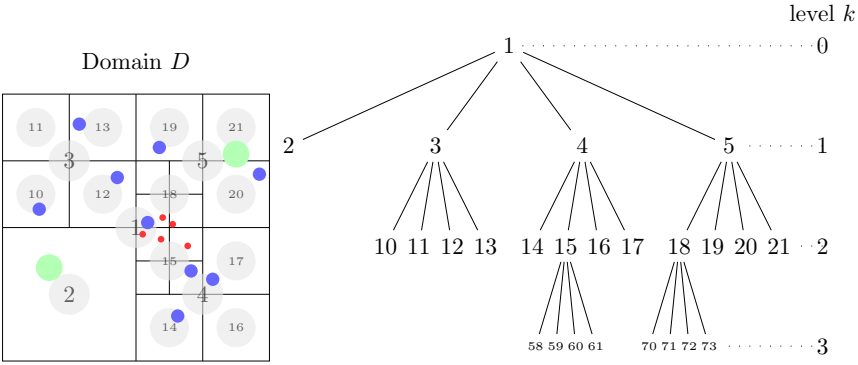


Figure 4.2. Left: sketch of an AR cell list with large (green), medium (blue), and small (red) particles. The domain D is adaptively subdivided (black lines). Right: The corresponding cell tree with cells c_k on levels k and level-order indices $J(c_k)$, corresponding to the numbers given in the gray circles in the left panel.

the level k of the first particle such that

$$D_m/2^k > r_{c,1} \geq D_m/2^{k+1}\,, \quad k = 0, \ldots, \text{maxlevel} - 1\,. \tag{4.2}$$

D_m is the minimum edge length of the domain[2]. Subsequently, we linearly iterate through all particle radii $r_{c,p}$, $p = 2, \ldots, N$ and increment k by one whenever $r_{c,p} < D_m/2^{k+1}$.

Phase II: After all particles have been assigned to their respective cell-tree levels we also sort them with respect to their spatial location. This is done using a recursive divide-and-conquer algorithm (Algorithm 4.2) analogous to Quicksort [Hoare, 1962]. In each recursion of the algorithm we are given a set of particles located in the bounding box of a certain tree cell. We first determine the center of the tree cell, $\vec{m}$. We then use $\vec{m}$ to partition the set of particles in that cell along each dimension into 4 (in 2D)

[2]In practice we first render the domain cubic by extending it in all directions to its maximum edge length. This avoids constraining the tree depth by the domain's aspect ratio.

or 8 (in 3D) subsets. This is done by successively using the i^{th} component, $i = 1, \ldots, d$, of $\vec{m}$ as the respective pivot and $\geq$ as the comparison operator. The same partitioning procedure is then recursively applied to each of the resulting subsets in their respective sub-cells. The recursion stops after k iterations for all particles living on tree level k. The partitioning recursion is separately done for each non-empty tree level, always starting from the entire domain D. This causes the particles on each level to sift down to their respective leafs, starting from the root of the tree.

After Phase II, the particle array is partitioned both by tree levels and by particle positions. Furthermore, the position sorting procedure returns all pairs of indices of the first and last particle in each cell. This information is stored in a lookup table such that the particles belonging to a certain cell can be found in $O(1)$ operations.

4.1.2. Operations on AR cell lists

Once the AR cell lists are built, various operations on them are required in order to efficiently compute particle–particle interactions or construct Verlet lists. These operations are:

Op1: Finding a cell

The cell c_k in which a position $\vec{x}$ and cutoff radius r_c is located can be determined by first computing the level in the cell tree as

$$k = \lceil \log_2(D_m / r_c) \rceil$$

and then traversing the tree from its root to level k. During traversal we check for each tree node in which of its quadrants (in 2D) or octants (in 3D) $\vec{x}$ is located and descend into the respective child node to locate c_k.

Algorithm 4.1 Constructing AR cell lists in d dimensions.

INPUT: particles $p = 1, \ldots, N$ with positions $\vec{x}_p$ and interaction cutoff radii $r_{c,p}$; cuboidal domain D with edge lengths $(D_1, \ldots, D_d)$; $D_m = \min_{i=1,\ldots,d}(D_i)$

OUTPUT: `cells` lookup table containing the cell indices and indices of the first and last particles in each cell.

1. sort particles in descending order by $r_{c,p}$
2. maxlevel $= \lceil \log_2(D_m / \min_p(r_{c,p})) \rceil$
3. assign particles to cell-tree levels:
   A particle with cutoff radius $r_{c,p}$ is assigned to level k, where $D_m/2^k > r_{c,p} \geq D_m/2^{k+1}$, $k = 0, \ldots,$ maxlevel $- 1$
4. **for** $k = 0, \ldots,$ maxlevel $- 1$
   a) partition particles p_k in level k using Algorithm 2.
   Start the recursion of Algorithm 2 with arguments $p = p_k$, $c = D$, `curr_level`$= 1$, and `target_level`$= k$.
   b) insert the indices of the first and last particle in each leaf of the partitioning into `cells`. Empty leafs are not added; the cell indices in `cells` are hence not contiguous.

Algorithm 4.2 Sorting the particles by their position.

INPUT: particles p with positions $\vec{x}_p$; cell c in which these particles live; the level to be partitioned in this recursion `curr_level`; the level on which the particles p live `target_level`.

OUTPUT: the sorted particle array and the indices of the first and last particle in that array belonging to the cell c.

1. compute the center $\vec{m} = (m_1, \ldots, m_d)$ of the cell c and the bounds of the equisized subcells $c_1, \ldots, c_{2^d}$
2. set initial partition to contain all particles, $P_1 = \{p\}$, and initial set of partitions $S = \{P_1\}$
3. **for** $i = 1, \ldots, d$
   a) **for** $j = 1, \ldots, 2^{i-1}$
      i. partition P_j along m_i into $P_j^{<m_i} = \{p : x_{p,i} < m_i\}$ and $P_j^{\geq m_i} = \{p : x_{p,i} \geq m_i\}$
      ii. replace P_j in S with $P_j^{<m_i}, P_j^{\geq m_i}$

   The resulting partitioning divides the particles into 2^d disjoint sets $\{p : \vec{x}_p \in c_i\}, i = 1, \ldots, 2^d$
4. **if** `curr_level == target_level`
   a) **return**
5. **else**
   a) **for** $i = 1, \ldots, 2^d$
      i. `Algorithm2(`$\{p : \vec{x}_p \in c_i\}, c_i$`,curr_level+1,target_level)`

Op2: Finding all particles in a cell

Given a cell index, we can look up the index of the first and last particle inside that cell in the `cells` table. Since not all cell indices exist, this can be done in $O(1)$ time by implementing `cells` as a hash table with the cell index as its key and the pair of particle indices as its value.

Op3: Finding the child cells of a cell

The indices of the children of a cell c are given by $J(c) \cdot 2^d + l$, $l = -2^d + 2, \ldots, 1$.

Op4: Finding the parent cell

The index of the parent cell of a cell c is $\lfloor (J(c) + 2^d - 2)/2^d \rfloor$.

Op5: Finding neighboring cells

The neighbor cells of a cell c are found by adding/subtracting the cell-edge length to/from the center $\vec{m}$ of cell c and using these locations $\vec{x}$ and the cutoff radius r_c of the tree level of cell c as arguments to Op1. If a neighbor cell does not exist in the `cells` data structure, this means that there are no particles in its region on this level and below, or that the requested cell lies outside the domain.

4.1.3. Using AR cell lists

Using the AR cell list data structures and the above-defined five operations, every particle can efficiently find all other particles within its neighborhood. This is done by retrieving for each particle all particles in the same cell, in all neighboring cells, and in all descendent cells of the cell tree.

This can also be used to efficiently construct Verlet lists [Verlet, 1967] in

adaptive-resolution particle simulations. A Verlet list is a data structure that explicitly stores the interaction partners of each particle, allowing each particle to directly access its neighbors. This further reduces the overhead compared to directly using AR cell lists for computing the particle–particle interactions, provided the Verlet lists do not need to be reconstructed at each time step of a simulation. In order to ensure this, the cutoff radius of each particle is enlarged by a safety margin, called "skin". The Verlet lists then only need to be reconstructed once any particle has moved further than this skin thickness.

Evaluating particle–particle interactions or constructing Verlet list based on AR cell lists starts from the particles living on the highest (coarsest) non-empty level of the cell tree and then proceeds level by level. It is therefore convenient to iterate through the particle array in the order given by the sorting produced by Algorithms 4.1 and 4.2.

We refer to interactions as *symmetric* when an interaction between particle p and q implies the same (possibly with negative sign) interaction between q and p. This symmetry can be exploited when evaluating particle interactions in order to avoid redundant calculations.

For each particle p we use Op1 to determine the cell c_k in which it lives and then retrieve the $3^d - 1$ neighboring cells using Op5. When building Verlet lists or evaluating *symmetric* interactions, we only need to find one partner in every interaction *pair*. This is illustrated in Fig. 4.3. We use Op2 to loop over particles q in c_k. For symmetric interactions or when building Verlet lists, this loop only considers particles in c_k with an index $> p$. Subsequently, we loop over all particles q in the neighboring cells. For symmetric interactions and when building Verlet lists, it is sufficient to consider only those neighbors of c_k with an index $> J(c_k)$. For each pair (p, q) we check whether the particles fulfill Eq. (4.1). Depending on whether the particle interactions are symmetric or not, only q is added to the Verlet list of p, or the two interaction partners are mutually added to each other's lists.

We then recursively use Op3 and Op5 to visit all descendent cells of c_k and their respective neighbors. We consider only those descendent cells c_k^δ of c_k on all finer levels $\delta = k + 1, \ldots,$ maxlevel that contain the position $\vec{x}_p$. Since the cell tree is not stored explicitly, but computed on demand, we first

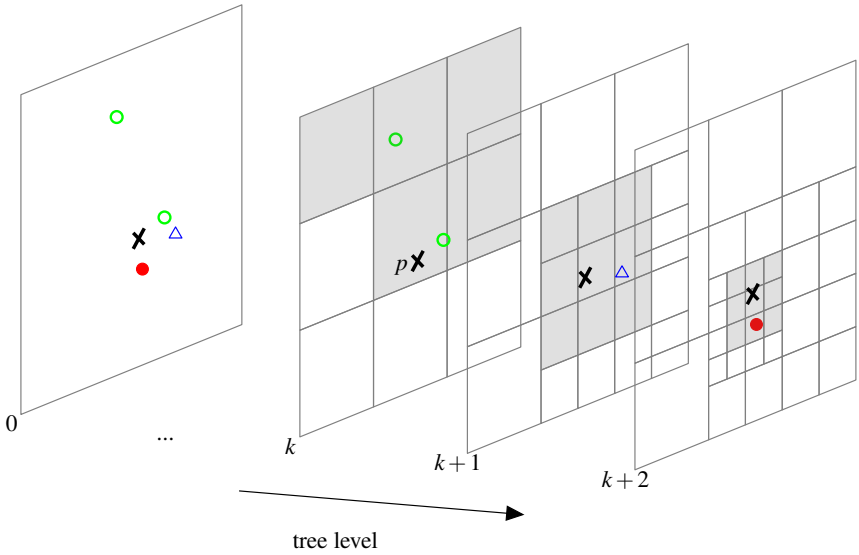


Figure 4.3. Finding interaction partners of a particle in an AR cell list (one iteration of Algorithm 4.3). The back plane (tree level 0) shows all particles without the cell-tree decomposition. In order to compute a symmetric interaction (or construct the Verlet list) of particle p (black cross) on level k we first iterate through all particles (green circles) in half of the neighboring cells $\mathcal{N}(c_k)$ on the same tree level (shaded cells on level k). Then, the finer tree levels are searched for interaction partners in the descendent cells c_k^δ and their neighbors $\mathcal{N}(c_k^\delta)$ (shaded cells on levels $k+1$ and $k+2$) on all finer levels $\delta = k+1, \ldots, \mathrm{maxlevel}$. On these finer levels, all neighboring cells must be visited in order to include all interaction pairs across different levels of resolution (blue triangles on level $k+1$ and red dots on level $k+2$). Transparent cells are not considered when computing this interaction.

determine the positions and edge lengths of c_k^δ and its neighboring cells $\mathcal{N}(c_k^\delta)$. Note that because we are iterating from large to small particles, we have to visit all 3^d-1 neighbors of c_k^δ in order to find all neighboring particles of p on higher levels of resolution, irrespective of whether the interactions are symmetric or not. We then retrieve all particles q in $c_k^\delta \cup \mathcal{N}(c_k^\delta)$ and check whether they fulfill Eq. (4.1) with particle p. Those that fulfill Eq. (4.1) are added to the Verlet list of p (and vice versa for asymmetric interactions), or their interactions with p are computed.

The complete procedure for computing particle–particle interactions or building Verlet lists based on AR cell lists is summarized in Algorithm 4.3. Note that even though Op4 is not used here, it would be necessary if one were to compute asymmetric particle–particle interactions directly based on AR cell lists, i.e., without building Verlet lists. We do, however, not consider this case.

Special treatment of halo layers for symmetric particle interactions

In a domain-decomposition setting, the present AR cell list algorithm operates independently on each subdomain of the computational domain (see Figs. 4.1 and 4.4). We rely on prior domain decomposition and population of the halo layers by the software in which the algorithm is embedded. This can also directly account for periodic boundary conditions, as also illustrated in Fig. 4.4. A parallel implementation of Algorithms 4.1 to 4.3 is hence not required. If the particle interactions are symmetric, halo layers are only needed on half of the (sub-)domain faces, halving the communication volume. This is illustrated in Fig. 4.4b. Since the interaction cutoff locally changes, the halo layers on different sides of a (sub-)domain may have different widths. Symmetric particle interaction schemes also change the properties (values) of ghost particles. These ghost contributions then have to be sent back to the corresponding real particle and properly accounted for (for example using the `ghost_put` mapping of the PPM library [Sbalzarini et al., 2006a]). Symmetric interactions can additionally result in two ghost particles interacting. These ghost–ghost interactions are efficiently found using bitwise operations as follows: Each ghost particle is assigned a d-bit string where the k^{th} bit is 1 if the particle is in the halo layer in dimension k and 0 otherwise. If a bitwise `AND` operation on the bit

strings of two ghost particles results in 0, these ghosts interact.

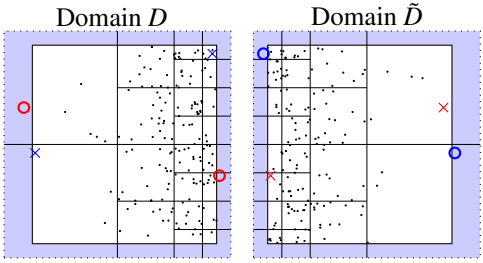
4.2. Results

We implemented Algorithms 4.1 through 4.3 in Fortran 90 as part of the PPM library and performed several computer experiments to benchmark their computational efficiency and evaluate the performance gain over uniform-resolution cell lists as a function of the spectrum of scales spanned by the cutoff radii and of the total number of particles in the domain. In all benchmarks, we verified that the AR cell lists found the correct set of interactions. The reference implementations of uniform-resolution cell and Verlet lists were taken from the PPM library and are also implemented in Fortran 90. All benchmark codes were compiled using the Intel Fortran compiler version 12.0 with the `-O3` optimization flag. The benchmarks were run on a 2.8 GHz Intel Xeon E5462 CPU.

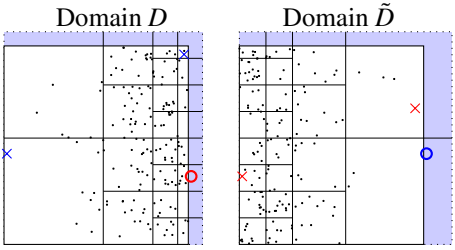
4.2.1. Benchmarks

We measure the computational time for building and using AR cell lists over different particle distributions. In all distributions we place a fixed number of 10×10 particles on a uniform Cartesian mesh with spacing $h_b = 0.1$ and set their interaction radii $r_{c,b} = 3h_b/2$. For each distribution we then choose a *resolution span* $\lambda = \max_p(r_{c,p}) / \min_p(r_{c,p})$ and a number of small particles N. These additional small particles are given interaction radii $r_{c,s} = r_{c,b}/\lambda$ and are placed on a uniform Cartesian mesh with spacing $h_s = 2r_{c,s}/3$ adjacent to the coarse mesh. Figure 4.5 shows an example of a resulting adaptive-resolution particle distribution. Similar particle distributions may arise in simulations of shock waves in compressible fluids. For the present benchmarks, the interaction radii are chosen such that each particle always has exactly 8 interaction partners, which allows comparing timing results across resolution spans.

We first measure the runtime scaling for constructing AR cell list and conventional cell list for increasing numbers of particles and constant λ. We repeat this experiment for $\lambda = [1, 10, 100, 1000]$ to cover a wide range of



(a) asymmetric interactions



(b) symmetric interactions

Figure 4.4. Halo layers for symmetric and asymmetric neighbor lists and treatment of periodic boundary conditions. The computational domain is decomposed into two (sub-)domains D and $\tilde{D}$ (cf. Fig. 4.1). On each (sub-)domain and its respective halo layers, a separate AR cell tree (black lines) is built. Blue crosses indicate particles in domain D that are ghosts in the halo layer of domain $\tilde{D}$ (blue circles). The red crosses highlight two particles from domain $\tilde{D}$ that are ghosts on domain D (red circles). For each color, two example-particles are shown: one for periodic boundary conditions, the other for internal (sub-)domain boundaries.

Algorithm 4.3 Computing particle–particle interactions or building Verlet lists based on AR cell lists.

INPUT: particles $p = 1, \ldots, N$ with positions $\vec{x}_p$ and cutoff radii $r_{c,p}$.
OUTPUT: result of the particle–particle interaction or Verlet list storing for each particle the indices of all particles within its neighborhood.

for each particle p

1. determine the cell c_k containing p ($\vec{x}_p$, $r_{c,p}$) using (Op1).
2. **if** computing symmetric particle–particle interactions or constructing Verlet lists **then**
   a) retrieve those neighbors of c_k with index $> J(c_k)$, $\mathcal{N}(c_k)$, using (Op5).
   b) **for** each particle $q > p \in c_k$ and each particle $q \in \mathcal{N}(c_k)$ (Op2)
      i. **if** $||\vec{x}_p - \vec{x}_q|| \leq \min(r_{c,p}, r_{c,q})$ **then** add q to the Verlet list of p (and vice versa when later computing asymmetric interactions based on these Verlet lists), or compute the interaction between particles p and q.
3. **else**
   a) retrieve all neighbors of c_k, $\mathcal{N}(c_k)$, using (Op5).
   b) **for** each particle $q \in (c_k \cup \mathcal{N}(c_k))$ (Op2)
      i. **if** $||\vec{x}_p - \vec{x}_q|| \leq \min(r_{c,p}, r_{c,q})$ **then** compute the interaction between particles p and q.
4. **for** $\delta = k + 1, \ldots,$ maxlevel
   a) use (Op3) to determine the cell c_k^δ that is the $(k-\delta)^{\text{th}}$ descendant of c_k and contains the location $\vec{x}_p$.
   b) retrieve all neighbors of c_k^δ, $\mathcal{N}(c_k^\delta)$, using (Op5)
   c) **for** each particle $q \in (c_k^\delta \cup \mathcal{N}(c_k^\delta))$ (Op2)
      i. **if** $||\vec{x}_p - \vec{x}_q|| \leq \min(r_{c,p}, r_{c,q})$ **then** add q to Verlet list of p (and vice versa when later computing asymmetric interactions based on these Verlet lists), or compute the interaction between particles p and q.

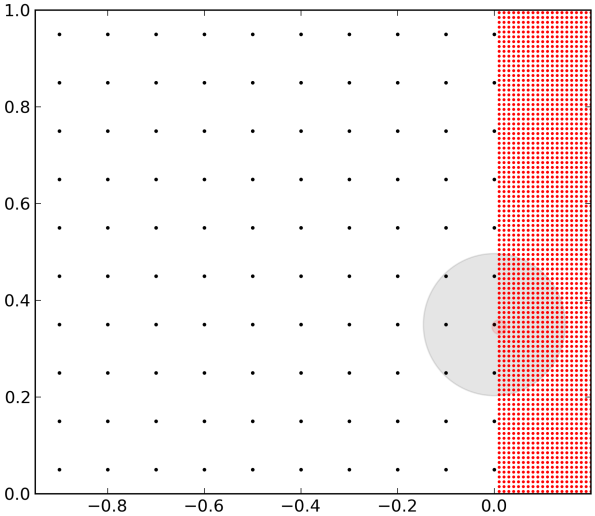
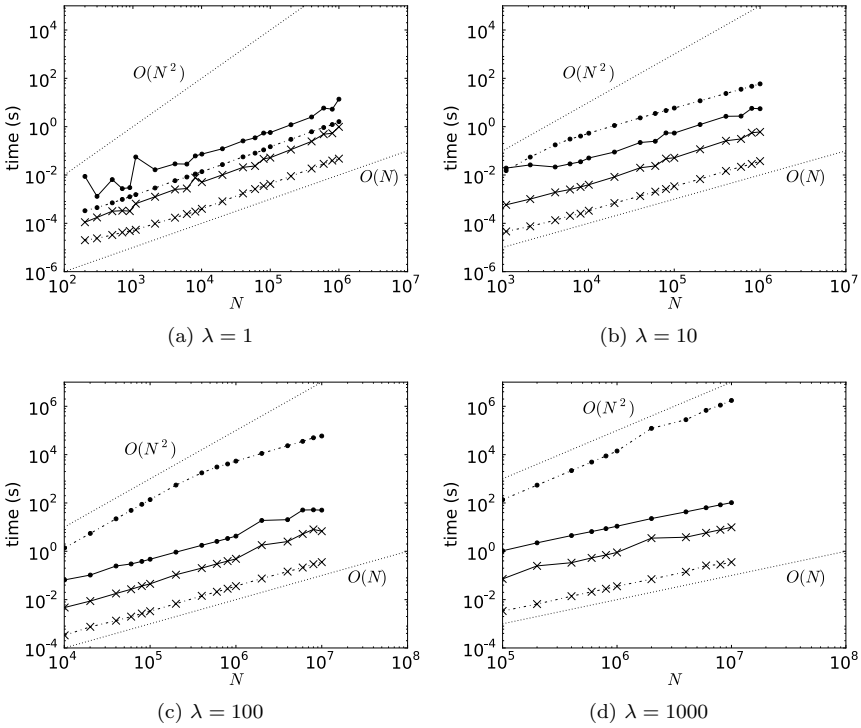


Figure 4.5. An example particle distribution used for the present benchmarks. In this figure $N = 2000$ and $\lambda = 10$. The "large" black particles have a cutoff radius of 0.15, while the "small" red particles have a cutoff radius of 0.015 ($\lambda = 0.15/0.015 = 10$). For comparison, the interaction ranges of two neighboring particles at the resolution interface are shown as shaded circles of the respective color.

resolution spans. The results are shown in Fig. 4.6. Constructing AR cell lists is about one order of magnitude slower than constructing conventional cell lists. A quick analysis of Algorithm 4.1 shows that Step 1 can be accomplished in $O(N \log N)$ time. Step 2 can directly be computed in $O(1)$. Step 3 is essentially a linear iteration through the N particles and therefore has a runtime of $O(N)$. Step 4 linearly depends on the number of cell tree levels, which in turn depends on λ. If the number of interaction partners of each particle is bounded by a constant, the overall runtime of the algorithm is $O(\text{maxlevel} \times N \log N)$. This is a higher computational complexity than the $O(N)$ runtime for building conventional cell lists.

Figure 4.6 also shows the total runtimes to construct conventional and AR cell lists *and* build Verlet lists based on them for $\lambda = [1, 10, 100, 1000]$. For $\lambda = 1000$ and 10^6 particles (Fig. 4.6d), building the cell lists and the Verlet lists for all particles is almost three orders of magnitude faster when using AR cell lists instead of conventional ones. Figures 4.6b and c further show that the runtime for constructing Verlet lists based on conventional cell lists is first $O(N^2)$ and then decreases to $O(N)$ beyond a "saturation point". This can be understood as follows: Since the cells of conventional cell lists are as large as the largest cutoff radius in the domain, the runtime of the particle–particle interactions using conventional cell lists in the present test case is about $100 + N^2/N_{\text{cells}}$ for 100 particles with large $r_{c,b}$ and N particles with small $r_{c,s}$. For small N the total number of cells in the cell list, N_{cells}, is constant and the quadratic term dominates, leading to a quadratic runtime as more and more small particles are added into the constant number of cells covering the domain. For large-enough N, after the rightmost column of cells has been completely filled with small particles, N_{cells} increases proportionally with N, rendering the runtime linear beyond this saturation point. This can be seen in Figs. 4.6b and c as a reduction in the slope of the particle–particle interaction runtime curve. As λ increases, the saturation point shifts to larger N.

The runtimes of AR and conventional cell lists depend on the spectrum of scales λ present in the particle distribution. For $\lambda = 1$ conventional cell lists are more efficient (see Fig. 4.6a). For increasing λ, the additional overhead for constructing the AR cell lists is gradually amortized by their higher efficiency when computing particle–particle interactions. We therefore repeat the benchmarks for different values of λ between 1 and 10^4 and measure the



(a) $\lambda = 1$ (b) $\lambda = 10$

(c) $\lambda = 100$ (d) $\lambda = 1000$

Figure 4.6. Runtime for increasing numbers of particles N and resolution spans $\lambda = [1, 10, 100, 1000]$ (a–d). Each plot shows the total runtime for constructing (crosses) conventional (dashed lines) and AR (solid lines) cell lists in 2D and for constructing the cell lists *plus* constructing Verlet lists based on them (dots). The theoretical slopes of an $O(N)$ and an $O(N^2)$ algorithm are indicated by the dotted lines.

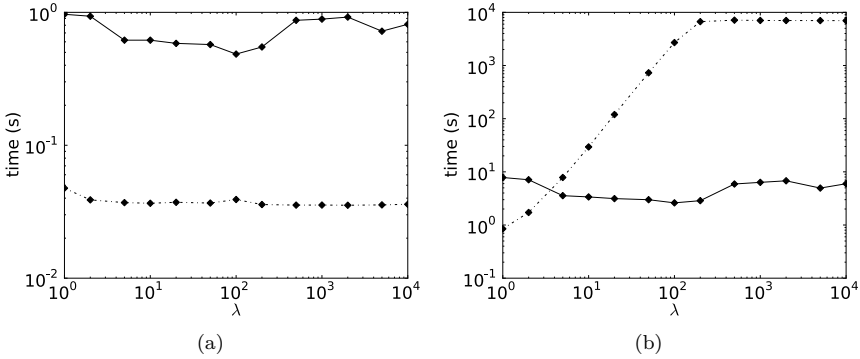


Figure 4.7. Total runtime for increasing resolution spans $1 \leq \lambda \leq 10\,000$ and constant $N = 10^6$. (a) Runtime for constructing conventional (dashed line) and AR (solid line) cell lists in 2D. (b) Total runtime for constructing conventional (dashed line) and AR (solid line) cell list *and* constructing Verlet list based on them.

total runtime. The measured runtimes are shown in Fig. 4.7. As expected, the cost of constructing conventional cell lists is independent of λ and about one order of magnitude lower than for the AR variant (Fig. 4.7a). When using AR cell lists to build Verlet lists, however, the computational cost is virtually independent of λ, whereas for conventional cell lists it rapidly grows with λ (Fig. 4.7b). This is expected as the particles cluster more and more and the average number of particles per cell grows (quadratically in 2D and cubically in 3D) for conventional cell lists, whereas it remains constant in AR cell lists. The runtime for building the Verlet lists using conventional cell lists reaches a plateau at $\lambda \approx 200$. This can be explained by the specific arrangement of particles used in the present benchmark. At $\lambda > 200$ the particles with small cutoff radii are so tightly arranged that they all fit into the minimum number of cells required to cover the interface between the large and small particles.

We determine the break-even value of λ where the overall runtime for constructing AR cell lists and using them to construct Verlet lists drops below that for constructing conventional cell lists and building Verlet lists based on them. For $\lambda = 1$, constructing Verlet lists form conventional cell lists is about 25% faster than constructing them from AR cell lists.

Already for $\lambda = 3.65$, however, the overall runtime for AR cell lists is equal to that for conventional cell lists. For resolution spans of about $\lambda = 10$, AR cell lists are about one order of magnitude faster than conventional ones. This indicates that the use of AR cell lists is advantageous in most adaptive-resolution particle simulations, even for modest resolution spans.

4.2.2. Example application

As an example application where AR neighbor lists may be advantageous we consider diffusion on a curved surface simulated using an adaptive-resolution variant of a smooth particle method [Bergdorf et al., 2010]. The surface is represented implicitly as a level set [Sethian, 1999] that is discretized using particles as collocation points [Hieber and Koumoutsakos, 2005]. Diffusion amounts to interactions between neighboring particles as defined by DC-PSE operators [Schrader et al., 2010].

We consider a surface of revolution generated by three arcs of circles, resembling a small bud pinching off from a larger sphere (see Fig. 4.8). This models the geometry of a dividing yeast cell. The radii of the bud and of the sphere are fixed to 1 and 3, respectively. The radius of curvature at the neck, r_P, is varied parametrically in order to tune the resolution span present in the problem.

In order to properly resolve the geometry, the density of particles needs to be larger (and their interaction radii smaller) in regions where the surface has a large curvature. We hence place the particles such that the distance between neighboring particles is proportional to the local radius of curvature of the surface. The cutoff radii hence span a continuous spectrum of scales and the geometry is well resolved everywhere, as shown in Fig. 4.8. Particles are only placed in a narrow band around the surface and the rest of the volume remains empty [Bergdorf et al., 2010]. Varying the neck curvature r_P leads to different ratios between the largest and the smallest curvature of the surface, and hence to different resolution spans λ. The mean resolution h_0 on the larger sphere is fixed in each run, so that decreasing r_P (i.e., increasing λ) leads to an increase in the total number of particles N.

We measure the computational cost of constructing and using cell lists

in the present AR method and compare it with the cost of conventional cell lists for mean resolutions $h_0 = [0.1, 0.2, 0.45]$ and λ varying between 3 and 2000. Figure 4.9a shows the total runtime for constructing the Verlet lists using either AR cell lists or conventional cell lists. For the coarsest resolution, the break-even point is around $\lambda = 60$. This reduces to $\lambda = 4$ for $h_0 = 0.2$ and to $\lambda < 2$ for the finest resolution considered. Figure 4.9b shows the runtime per particle for constructing the cell and Verlet lists, demonstrating that AR cell lists provide neighbor access with a runtime that is insensitive to the resolution span λ and to the total number of particles N as realized by the different resolutions. This is in contrast to conventional cell lists whose runtime significantly increases with λ and with increasing N (decreasing h_0).

4.3. Conclusions

We have presented data structures and algorithms for efficiently finding the interaction partners of each particle in a particle-based simulation with short-range interactions whose cutoff radii vary between particles. This enables efficient computation of limited-range particle–particle interactions in adaptive-resolution simulations with a potentially continuous spectrum of cutoff radii. Constructing adaptive-resolution (AR) neighbor lists is computationally more expensive than constructing conventional uniform-resolution neighbor lists. This additional overhead, however, is quickly amortized by the gain in performance when using the AR cell lists to evaluate particle–particle interactions or to construct Verlet lists for adaptive-resolution particle distributions. Already at modest ratios between the cutoff radii of the largest and smallest particles in a simulation, AR cell lists are faster overall. The actual break-even point, however, depends on the specific particle distribution. The larger the spectrum of scales that are present in a simulation, the bigger the computational saving becomes. For realistic adaptive-resolution simulations, the present AR cell lists can be several orders of magnitude faster than conventional cell lists.

We have implemented both AR cell lists and Verlet lists based on AR cell lists in the PPM library in order to make them available for adaptive-resolution simulations on parallel distributed-memory computers. In PPM,

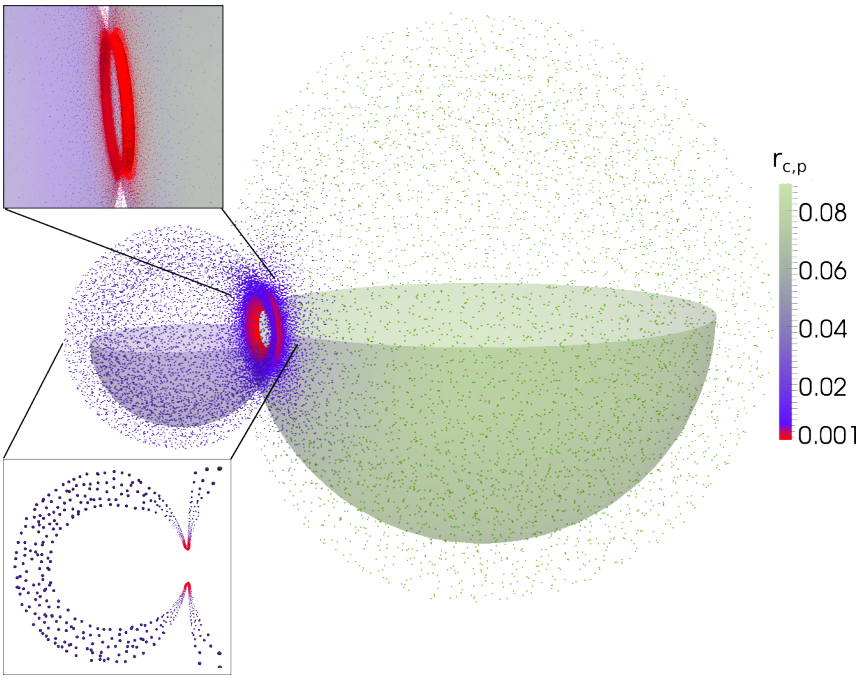


Figure 4.8. Particle distribution used in the present example of an adaptive-resolution simulation of diffusion on a surface. The surface is axially symmetric and only its lower half is shown. The surface is represented as a level set discretized on the particles and restricted to a narrow band. Both the width of the narrow band and the cutoff radii of the particles, represented by the color code, depend on the local surface curvature. The high curvature at the neck between the two sphere shells requires a locally increased resolution.

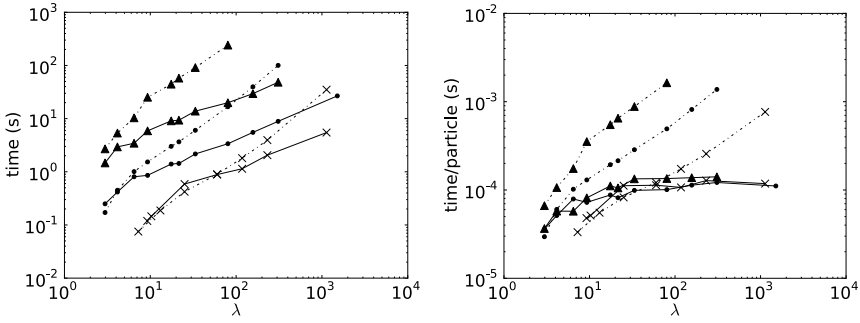


Figure 4.9. Runtime for constructing the cell lists *and* using them to construct Verlet lists for the test case shown in Fig. 4.8. For both conventional (dashed lines) and AR (solid lines) cell lists we vary the resolution span λ and the mean resolution h_0 on the larger sphere, hence varying the total number of particles N. Crosses, dots, and triangles correspond to $h_0 = 0.1$, 0.2, and 0.45, respectively. Note that N increases with λ. The two panels show: (a) the total runtime and (b) the runtime per particle.

the presented algorithms are applied locally per subdomain (i.e., per processor) of a domain decomposition. They thus have no impact on the communication overhead of a parallel simulation, assuming that the halo layers are populated beforehand.

CHAPTER 5

A new edge-coloring-based communication scheduler

[1]Parallel simulations of tightly coupled problems often entail a significant overhead due to interprocess communication. This is particularly true when the problem is decomposed and distributed across the processors of a distributed-memory computer, and when using synchronous communication. In this case, a process may only communicate with one other process at a time. Hence, even local communication, where processes only exchange data with their neighbors, must be decomposed into rounds. These rounds are sequentially executed until all processor pairs have completed their communication. The problem of communication scheduling with a minimum number of rounds can be abstracted as the graph-theoretical problem of graph coloring.

The graph (or vertex) coloring problem consists in coloring the vertices V

[1]This work has been done together with Ferit Büyükkeçeci, who provided the Fortran implementation in the PPM library.

of a undirected graph $G = (V, E)$, such that no two vertices connected by an edge $e \in E$ have the same color. The minimum number of colors needed to color G is called the *chromatic number* $\chi(G)$. A coloring using $k \geq \chi(G)$ colors is referred to as a k-coloring of G. Analogously, the *edge coloring* of a graph colors all edges of the graph such that no two edges sharing a vertex carry the same color. The minimum number of colors needed for a *proper edge coloring* is called the *chromatic index* $\chi'(G)$. Since an edge coloring of G is equivalent to a vertex coloring of the line graph $L(G)$ of G, it follows that $\chi'(G) = \chi(L(G))$.

Graph coloring problems have first been studied in the 19th century. In the 1970s, optimal graph coloring was identified as one of 21 NP-complete problems [Karp, 1972]. However, many heuristic algorithms have since then been found to determine a k-coloring of G [Brélaz, 1979, Turner, 1988, Kosowski and Manuszewski, 2004]. In the case of edge coloring, it is clear that the chromatic index must be at least as large as the graph's maximum degree $\Delta(G)$ (i.e., the maximum number of edges connected to any vertex). Furthermore, Vizing's theorem states that $\chi'(G)$ for any graph is either $\Delta(G)$ or $\Delta(G) + 1$ [Vizing, 1964].

The generic problem of scheduling a set of jobs E using shared resources V, where only one job at a time can access the resource, can be abstracted as the edge coloring problem. The chromatic index of this graph is the minimum number of rounds in which all jobs can be completed. A common application of graph coloring in computing is register allocation [Chaitin, 1982]. In order to reduce the number of memory accesses in computer programs, compilers must find an optimal schedule for assigning processor registers to variables.

Coffman et al. [1985] proposed to also abstract the problem of communication scheduling as an edge coloring problem. They found both a sequential and a distributed polynomial-time approximation algorithm. The general communication scheduling problem is abstracted as follows: The processes or computer nodes are represented by the vertices of a graph $\tilde{G}$. If two processes have to communicate, their vertices are connected by an edge. Following our assumption that a process can only communicate with one other process at a time, we see that two edges connected to the same vertex must be of different color. Alternatively, vertex coloring algorithms can be

applied to the line graph $L(\tilde{G})$ of the communication graph $\tilde{G}$, which is what we do here.

5.1. A heapified implementation of DSATUR for communication scheduling

The DSATUR algorithm was proposed by Brélaz [1979] as a heuristic algorithm for finding vertex colorings of a graph. The algorithm introduces the *saturation degree* of a vertex, $sat(\cdot)$, as the number of distinct colors to which it is adjacent. The algorithm proceeds as follows until all vertices are colored:

> Select an uncolored vertex with maximum saturation degree. Ties are broken by choosing the vertex with maximum degree (number of adjacent vertices, $deg(\cdot)$). Color this vertex with the next available color.

The asymptotic runtime of this algorithm is $O(n^2)$ [Brélaz, 1979]. Here, n denotes the cardinality of the vertex set $V$of G, while m is the cardinality of the edge set E of G. Turner [1988] showed that by using a heap data structure it is possible to further reduce the runtime of DSATUR to $O((n+m)\log n)$. We generalize this implementation using a *list of heaps* (Fig. 5.1). The list is indexed by the saturation degree while the heaps use

$$heap(v_i) < heap(v_j) \iff deg(v_i) > deg(v_j)$$

as their ordering relationship; $heap(v_i)$ is the position of v_i in the heap. Using this data structure allows us to partition the binary heap used by Turner [1988] in order to allow easy traversal of the vertex set by both saturation and adjacency degree.

The heapified DSATUR algorithm (Algorithm 5.1) starts by adding all vertices of G into the $sat(\cdot) = 0$ heap ($heap_0$) and initializing their saturation degrees to 0. While there are still uncolored vertices, the algorithm pops the head from the heap with highest saturation and colors it with the next available color. For all adjacent vertices, it then checks whether their satu-

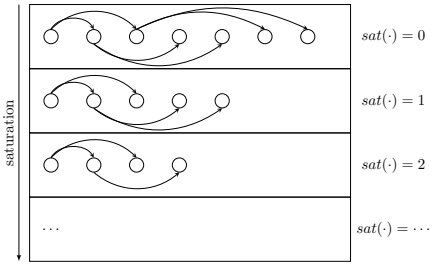


Figure 5.1. A list of heaps for fast access to vertices with maximum degree and given saturation degree.

ration has increased and updates their position in the list of heaps. Since all heap operations remain unaltered, the heaps can directly be accessed by index, and the overall structure of the algorithm is the same as in [Turner, 1988]. We hence argue that the overall runtime of our algorithm remains $O((n + m) \log n)$. This holds under the assumption that the number of neighbors per vertex is bounded by a constant much smaller than n.

5.1.1. Using heapified DSATUR in PPM

We implemented the heapified DSATUR algorithm as a Fortran 95 module in the Parallel Particle Mesh (PPM) library [Sbalzarini et al., 2006a]. The PPM library's local and ghost mapping operations entail local communication between all MPI processes and their neighbors. The neighborhood relationship emerges from PPM's domain decomposition. The computational domain of a parallel particle-mesh simulation is decomposed into subdomains that are then assigned to MPI processes. Hence, two processes holding two neighboring subdomains are neighbors. Figure 5.2 shows how a communication graph is constructed from such a domain decomposition.

Since local and ghost mappings are implemented using synchronous message passing, each processor can only communicate with one neighbor at a time. We therefore determine the communication schedule using vertex coloring of the line graph $G = L(\tilde{G})$ of the communication graph $\tilde{G}$. The schedule is stored and reused. It is only necessary to recompute the schedule after

Algorithm 5.1 DSATUR algorithm using s *list of heaps*. `heapify`, `pop`, `delete`, and `insert` are the standard binary heap operations.

INPUT: A graph $G = (V, E)$
OUTPUT: A vertex coloring of G.

1. $heap_0 \leftarrow$ `heapify(`V`)`
2. $\forall v \in V : \ sat(v) = 0$
3. $W \leftarrow V$
4. **while** $W \neq \emptyset$
   a) $i \leftarrow \arg\max_i(heap_i \neq \emptyset)$
   b) $v \leftarrow$ `pop(`$heap_i$`)`
   c) color v with the least available color in its neighborhood
   d) **foreach** $w \in neigh(v)$
      i. **if** $\forall u \in neigh(w) \setminus v : \ color(v) \neq color(u)$
         A. `delete(`$heap_{sat(w)}, w$`)`
         B. $sat(w) \leftarrow sat(w) + 1$
         C. `insert(`$heap_{sat(w)}, w$`)`
   e) $W \leftarrow W \setminus v$

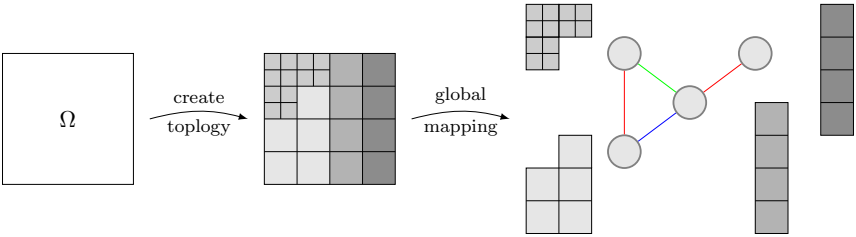


Figure 5.2. First, we create a topology for the computational domain Ω (left panel) by decomposing it into subdomains and assigning them to processors (center panel). Then, we map the data onto this topology. The subdomain neighborhood defines a processor neighborhood (right panel). This neighborhood is abstracted as a graph $\tilde{G} = (V, E)$, where V is the set of processors and $E = \{(v_i, v_j)$ if v_i, v_j have neighboring subdomains$\}$. An edge-coloring of $\tilde{G}$ is computed by applying the vertex coloring algorithm DSATUR (Algorithm 5.1) to the line-graph $G = L(\tilde{G})$. Free space boundary conditions are assumed.

a global mapping has been performed.

5.2. Benchmarks

We compare the performance of the heapified DSATUR line graph vertex-coloring (i.e., edge-coloring) Fortran 95 code (`dsatur`) with the previous edge-coloring routine in PPM, which is based on the *Stony Brook Algorithm Repository* C++ implementation (`vizing`) of the edge-coloring algorithm that follows from the constructive proof [Misra and Gries, 1992] of Vizing's theorem [Vizing, 1964]. All benchmarks are performed on an Intel Core i5 CPU and compiled using the `-O3` flag. For all benchmarks we measure for `dsatur` the total time of first computing the line graph $L(\tilde{G})$ and then computing the vertex coloring on $G = L(\tilde{G})$.

In order to study the performance of our edge-coloring code we generate a number of graphs with different properties and measure the time and number of colors used by both `dsatur` and `vizing`. For all tested graphs the number of colors used by `dsatur` is equal or one less than `vizing`. Figure 5.3 compares the two implementations for random graphs of increasing numbers

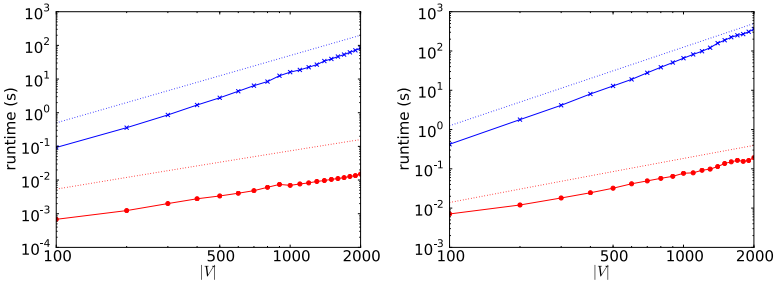


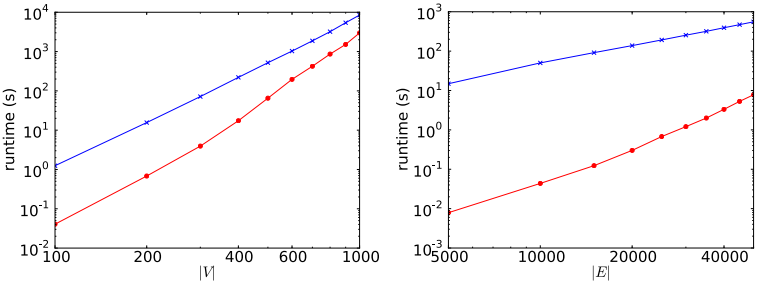
Figure 5.3. Runtime of Vizing's edge coloring (blue crosses) and heapified DSATUR (red circles) for random graphs of increasing numbers of vertices, each of which having a degree of 10 (a) or 25 (b). Heapified DSATUR performs better in both cases and scales better than Vizing's algorithm. The asymptotic runtimes of Vizing's algorithm ($O(nm)$: blue) and heapified DSATUR ($O((n+m)\log m)$ red) are shown as dotted lines. Heapified DSATUR is applied to $G = L(\tilde{G})$ while Vizing is directly applied to $\tilde{G}$.

of vertices $n = |V|$. The adjacency degree of the vertices is fixed to 10 (Fig. 5.3a) or 25 (Fig. 5.3b). The `dsatur` implementation has an overall better time performance than `vizing`. Moreover, `dsatur` also exhibits a slower runtime increase which is due to the algorithm's asymptotic complexity. Vizing's algorithm has an asymptotic runtime of $O(nm)$ [Kosowski and Manuszewski, 2004], whereas our algorithm has an asymptotic runtime of $O((n+m)\log m)$[2].

Examining the runtimes of `dsatur` and `vizing` with respect to the density of the graph (Figure 5.4), we notice that both `dsatur`'s and `vizing`'s runtimes show a steeper increase than in Figure 5.3. This is due to the fact that the number of neighbors per vertex is no longer in $O(1)$. This however, is of no concern for communication scheduling in local and ghost mappings since the neighborhood of any processor remains local.

We thus also generate 2 and 3 dimensional grids as benchmark cases for

[2]Since we are applying the heapified DSATUR algorithm on the line graph of the communication graph, the runtime here differs from the one given in section §5.1.



(a) (b)

Figure 5.4. Runtimes of dsatur and vizing for graphs of increasing $|V|$ and $deg(v) = |V|/2 \ \forall v \in V$ (a), and graphs with $|V| = 1000$ and increasing $|E| = (|V|deg(v))/2$ (b). Both vizing and dsatur's runtimes rapidly increase with increasing density of the graph.

dsatur. This simulates Cartesian PPM domain decompositions and thus provides an application benchmark. We again compare the runtime performances of dsatur and vizing. The number of vertices (processes) ranges from 4 (8 in 3D) to 4096, and we consider an 8 (26 in 3D) neighborhood around each vertex (process). Figure 5.5 a and b show the results. In both cases dsatur has an improved runtime over vizing. For the largest test cases (4096 vertices), dsatur is more than 100 times faster than vizing and completes the edge coloring in less than 0.1s. We have also tested dsatur's runtime for two larger 2D and 3D Cartesian grids that could not be colored by vizing within one hour. The results are summarized in Table 5.1.

Comparing all test cases dsatur always generated a coloring with at most the same number of colors as vizing. This means that the dsatur edge colorings are for our test cases at most $\chi'(\tilde{G}) + 1$.

5.3. Summary and Conclusion

We have implemented a new graph vertex coloring based communication scheduler for synchronous message passing in parallel high-performance

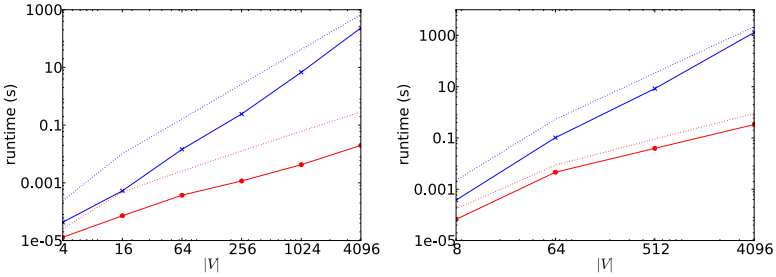


Figure 5.5. dsatur (red) and vizing (blue) runtime performance benchmarks on a cartesian mesh in 2D with 8-neighborhood (a) and 3D with 26-neighborhood (b). Asymptotic runtimes for Vizing's algorithm ($O(nm)$: blue) and heapified DSATUR (on line-graphs, $O((n+m)\log m)$ red) are shown in dotted lines.

vertices	time
262144 (2D, 8-neighborhood)	1.825s
110592 (3D, 26-neighborhood)	10.878s

Table 5.1. Runtimes of the dsatur implementation of the heapified DSATUR algorithm on line-graphs. No benchmarks could be obtained for vizing within an hour.

applications. Our method is based on Brelaz's [1979] heuristic vertex coloring algorithm. Extending Turner's [1988] idea of using a binary heap for fast vertex lookup, we proposed building a heap-list data structure that stores one heap per saturation degree and allows direct access to vertices of maximum saturation (and adjacency) degree. Using this data structure, we expect an overall asymptotic runtime of $O((n+m)\log n)$ [Turner, 1988]. In the context of communication scheduling, this algorithm is applied to the line graph of the communication graph. We implemented this algorithm as a Fortran 95 module in the Parallel Particle Mesh (PPM) library and benchmarked it on a number of different graphs. We generated random graphs of different sizes and with different adjacency degrees, and we compared the runtime performance of the new Fortran routine with a C++ implementation of Vizing's algorithm [Misra and Gries, 1992]. Furthermore, we compared the performance of the two implementations on 2D and 3D Cartesian grids modeling realistic PPM communication graphs. Our implementation outperformed the previous communication scheduler in the PPM library in all tested benchmarks. However, we noticed that our implementation is unsuitable for dense graphs with high adjacency degrees. We ascribe the increased runtime for such graphs to the fact that the neighbors of a vertex cannot be accessed in $O(1)$ time anymore. Nevertheless, the presented new communication scheduler is able to compute colorings for 3D Cartesian grids with more than 10^5 nodes in ca. 10 seconds.

CHAPTER 6

PPM on multi- and manycore platforms

In 2005 the semiconductor industry made a technological move that has significantly influenced computer science. It had become clear that traditional chip designs were not able to meet the ever-increasing demand for performance. In the previous two decades the miniaturization of transistors and the increase in processor clock speed were the two main drivers of progress in the hardware industry. Consequently, software was able to benefit from an increased performance when executed on newer hardware. However, physical limitations such as the heat dissipation and power consumption of chips, as well as design limitations such as an increasing gap between CPU and memory speeds, led chip manufacturers to rethink the chip design [Asanovic et al., 2009, Geer, 2005].

Instead of further increasing the clock frequency, chips now comprise multiple cores. Newer designs have become more power efficient, cooler and more performant. Since 2005, the number of cores has increased from two

to four, six, and recently 8 (figure 6.1).

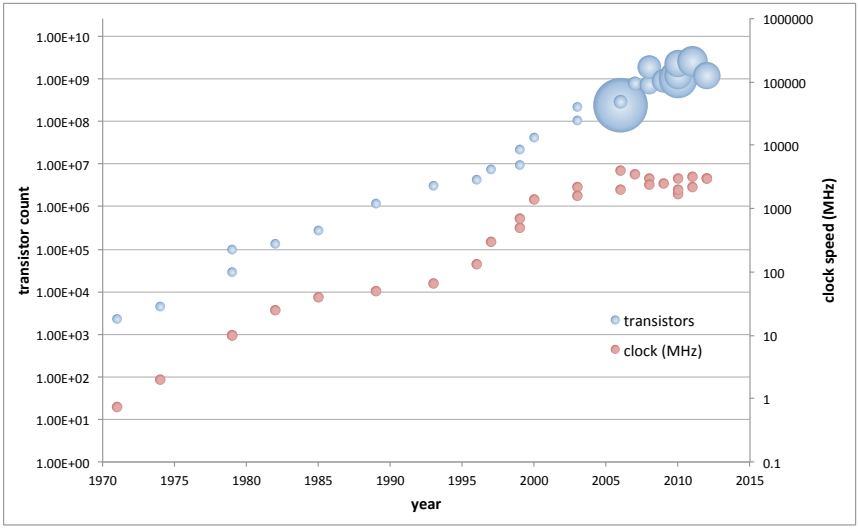


Figure 6.1. Transistor numbers have been steadily increasing, following Moore's law, while clock speed has leveled off since 2005. At the same time, new processors have emerged with two or more cores (indicated by the size of the bubbles) (source of raw data: Wikipedia, Intel).

These changes have in turn presented new challenges to algorithms and software engineering. In order to optimally exploit the new processors, algorithms and software have to be designed with parallelism in mind [Asanovic et al., 2009]. Most of the difficulties in programming parallel systems stem from the fact that resources have to be shared between several concurrent execution threads. Some of the pioneering work in concurrent programming is decades old. For instance, *mutual exclusions (mutex)* [Dijkstra, 1965] ensure that two processes (or threads) can access the same shared memory location without interfering with each other. *Transactional memory* allows a pair of load and store instructions to be executed atomically. Transactional memory has been implemented both in hardware [Herlihy and Moss, 1993] and software [Shavit and Touitou, 1997], and it is used in IBM's latest Blue Gene\Q supercomputer design.

In order to further ease building and updating parallel software, frameworks and libraries providing the tools for shared-memory parallel programming have become crucial to the computer industry and the scientific community.

Scientific computing has particularly benefited on several levels from the advent of multicore processors. On the one hand, multicore processors (and even more so manycore processors) turn the personal computers and workstations of engineers and scientists into powerful parallel computing machines. This makes it desirable that programming such machines be accessible to non-experts, such that the users of such workstations can optimally exploit their computing resources [Marowka, 2007, Brodtkorb et al., 2010, Perez et al., 2008]. On the other hand, multicore chips have enabled the construction of petascale supercomputers, while the next-generation manycore CPUs will be at the heart of upcoming exascale supercomputers. These systems are heterogeneous in nature, featuring not only multicore CPUs, but recently also GPGPUs, APUs, and even FPGAs [Tsoi and Luk, 2010].

GPUs in particular have been increasingly employed in applications outside computer graphics. Originating from early prototypes by Olano and Lastra [1998] graphical processing units have become increasingly flexible in allowing access to their functional units via vertex and shader programs. Today, featuring hundreds of parallel streaming processors, GPUs have become popular platforms for SIMD-parallel algorithm implementations. First implementations of a linear algebra algorithm have appeared as early as 2001 [Larsen and McAllister, 2001]. In the following years, more generic numerical algorithms have been ported to GPUs [Krüger and Westermann, 2005, Bolz et al., 2005], making use of the continuously improving hardware and its interfaces. In 2007 Nvidia released the first version of its CUDA SDK, a high-level language and API for using Nvidia's GPU platforms as general purpose parallel computing engines[NVIDIA, 2012]. OpenCL, an open standard for computing on CPUs, GPUs, APUs, and FPGAs, was released in 2008. Both competing platforms have fostered the implementation of many algorithms on GPUs.

Especially for tightly coupled problems it has become increasingly important to optimize implementation for multicore and manycore architectures. This has for example been discussed by Dubey et al. [2010],

Speck et al. [2011], Madduri et al. [2009]. Furthermore, Madduri et al. [2011] provide an extensive discussion of the opportunities and challenges of multi- and manny-core architectures for Particle-In-Cell (PIC) methods. They concluded that manycores and GPUs may both offer considerable speedups. However, the benefit and cost of employing GPUs and accelerators must be carefully assessed for the particular problem at hand. One important challenge of large heterogeneous platforms, which has also been pointed out by Rabenseifner et al. [2009], is load balancing. Dynamic load balancing may be expensive and incur a communication overhead in purely distributed-memory parallel implementations. By using hybrid shared-/distributed-memory programming, however, it is possible to alleviate this problem by saving some of the communication overhead and providing dynamic work item scheduling constructs. One such example is OpenMP's `schedule=dynamic` clause.

A number of peta-scale simulations performed on heterogeneous multi- and manycore systems have recently demonstrated the opportunities and challenges of these systems.[Hejazialhosseini et al., 2012] presented an object-oriented software for simulating compressible two-phase flows on 47,000 CPU cores achieving 30% of the nominal peak performance. Winkel et al. [2012] extended the PEPC library, a MPI Barnes-Hut tree code, using pthreads to scale up to almost 300,000 cores. Using 4,000 GPUs and 16,000 CPU cores Shimokawabe et al. [2011] achieved a 1 PFlop/s simulation of metal alloy solidification. Finally, Bernaschi et al. [2011] performed a biofluidics simulation at nearly 1 PFlop/s of blood flow through the human coronary arteries at the resolution of single red blood cells.

6.1. A pthreads wrapper for Fortran 2003

A common approach to writing scalable software for heterogeneous hardware platforms is to combine a distributed-memory parallelization library, such as MPI, with a threading library, like POSIX threads (pthreads) or OpenMP. MPI is then used to parallelize the application on the level of networked hosts, while the thread library is used to parallelize within each MPI process. The processor cores can thus be used to execute multiple threads in parallel. This strategy has been successfully used for example by Winkel et al. [2012]. It has several advantages over executing one MPI process per core. First, running several threads per process instead of running several processes results in a smaller overall memory footprint. This is not only due to the overhead incurred by process management, but also due to increased memory requirements for data replications, such as halo layers for decomposed domains [Rabenseifner et al., 2009, Winkel et al., 2012]. The problem is further aggravated since the size and bandwidth of main memory are not scaling with the number of cores [Dubey et al., 2010]. Second, using shared memory parallelism allows for improved dynamic memory access on NUMA architectures. Finally, using hybrid threads-MPI programming models allows the programmer to delegate tasks such as internode communication [Winkel et al., 2012, Song et al., 2009], job management, and job monitoring to be executed within one thread and depending on the workload, to be dedicated to one core. This is harder to achieve in a pure SPMD programming model.

Pthreads is a POSIX standard for threads that is implemented in many POSIX-compliant operating systems, ranging from BSD derivatives and Linux to MacOS X and Solaris. Microsoft Windows offers an implementation too, albeit not natively. The original standard was published in 1995, but a number of threads implementations predate POSIX threads [Stein and Shah Sunsoft, 1992, Powell et al., 1991]. In fact, the POSIX threads standard was created in an effort to consolidate the number of existing libraries into one common interface allowing programmers to write threaded applications that are portable across many operating systems.

Pthreads provides an API for the C programming language, offering functions to manage threads, mutexes, condition variables, and thread-specific data, and allowing for synchronization between threads using locks and

barriers. This API has been ported (i.e., wrapped) to several other programming languages giving a wide audience access to the threading programming model. Unfortunately, however, support for pthreads has been lacking in Fortran. Fortran has a long-standing history being used in many scientific computing and high performance computing applications. he Fortran 2003 standard includes many desirable extensions and also supports object-oriented programming. Over the last 5 decades a large number of libraries and applications has been created for Fortran and is today actively maintained and used. BLAS, Netlib, Lapack, FFTW, PETSc, PEPC, and NAG Fortran Library are examples of widely used libraries written in Fortran or providing a native Fortran interface. To date, two non-proprietary, albeit partial implementations of Fortran pthreads wrappers exist [Hanson et al., 2002, Nagle, 2005]. Furthermore, IBM provides a proprietary pthreads Fortran interface for its AIX platform. It is possible to bind from Fortran directly to C routines, but this is often difficult and sometimes impractical and hence not easily accessible to most Fortran programmers.

6.1.1. Features and limitations

The present implementation provides routines and derived types covering almost all POSIX threads [POSIX, 2004] capabilities, including optional specifications implemented in Linux and Linux-specific extensions. The provided Fortran interfaces cover:

- **Thread creation, joining, cancellation and basic management** providing basic threading functionality, such as creating and initializing new threads, calling of initialization routines, determining thread IDs, and comparing threads.
- **Mutexes**, or mutual exclusions, provide multiple threads with exclusive access to shared resources. Forthreads exposes all functions provided by pthreads mutex handling.
- **Conditional variables** are used in conjunction with mutexes, allowing threads to atomically check the state of a condition.
- **Barriers** are synchronization points at which participating threads must wait until all their peers have called the wait function.

- **Spin locks** provide a busy-wait type of locking for threads. A thread trying to acquire a spin-lock that is already locked by a peer checks in a loop for the availability of the lock and returns as soon as its peer has returned. Spin locks are more expensive in terms of resources than conventional locks based on the process or kernel scheduler, but offer a superior reaction time.

- **Readers-writer locks**, also known as shared exclusive locks, allow multiple threads acting as readers to acquire the lock at the same time in order to read a shared resource, while only one thread acting as a writer is allowed to acquire the lock for the shared resource. Forthreads offers wrappers for all RW lock pthread functions.

- **Thread attribute objects** are provided by pthreads to allow reading and modifying miscellaneous options, such as scheduling policy, stack size or scheduling priorities.

Our current implementation expands upon the one by Hanson et al. [2002] in particular by providing wrappers for barriers, spin locks, and readers-writer locks. These constructs are useful additions to mutexes and conditional variables and offer the programmer a set of flexible tools for thread synchronization.

The only pthreads API functions that could not be wrapped in the present Fortran implementation are:

- `pthread_cleanup_push` and `pthread_cleanup_pop`: These functions allow the programmer to register callback functions into a calling thread's cancellation cleanup stack that will be popped and executed in order whenever the thread exits, is cancelled, or calls `pthread_cleanup_pop` itself. These functions cannot be wrapped, as push and pop must be called in pairs in the same scope. Hence, the POSIX standard foresees their implementation to be done using C macros [POSIX, 2004].

- Pthread thread-specific data management routines (`pthread_key_*` and `pthread_getspecific` / `pthread_setspecific`): These routines heavily rely on the C programming language's `void` pointers. Unfortunately such pointers are not available in Fortran. Therefore, it seems

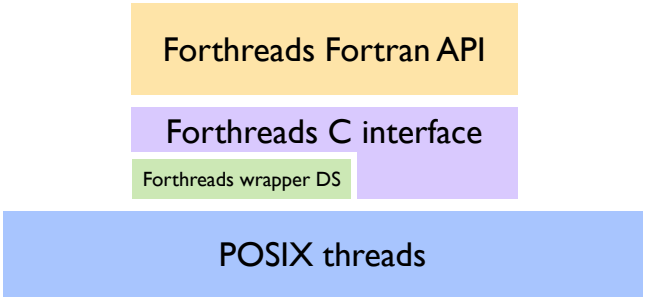


Figure 6.2. The forthreads library consists of two parts. First, the pthreads functions and opaque types are wrapped with C code that exposes Fortran-friendly data types and function interfaces. Then, a set of Fortran routines provides the user with a seamless Fortran-native interface calling internally the forthreads C functions and passing all necessary pointers and data.

difficult to provide portable and safe wrappers to these functions.

Finally, in contrast to pthreads, the current implementation only allows `INTEGER` pointers to be passed to the thread-start routine. This is for the same reason as the above-mentioned limitations on thread-specific data management routines.

The POSIX threads standard states that all pthread-specific types are opaque and that their specification should be treated as unknown to the user. Because of this limitation we chose to implement forthreads in two layers (figure 6.2). C language functions and data structures are first used to manage and store all pthreads objects and expose only indexes, primitive type variables, and types defined in forthreads itself. A set of Fortran 2003 derived types and routines wrapping the forthreads C routines define the actual forthreads API. The routines make heavy use of Fortran ISO C bindings introduced in Fortran 2003. They allow seamless interaction with the library without any knowledge of C/Fortran interoperability.

To illustrate our approach we provide in listings 6.1 and 6.2 the code required to wrap the `pthread_mutex_lock` function. This function locks the mutex with the given ID. If it is already locked by another thread, then the calling thread blocks until it can acquire the lock on the mutex. Listing

```
typedef struct array_tag {
  void **data;
  int size;
  int after;
  pthread_mutex_t mutex;
} array_t;

array_t *mutexes;

void thread_mutex_lock(int *mutex_id, int *info) {
  *info = FT_OK;
  if (!is_initialized) {
    *info = FT_EINIT;
    return;
  }
  if (!is_valid(mutexes,*mutex_id)) {
    *info = FT_EINVALID;
    return;
  }
  *info = pthread_mutex_lock((pthread_mutex_t*)
                  (mutexes->data[*mutex_id]));

}
```

Listing 6.1 The forthreads C wrapper code for `pthread_mutex_lock`

6.1 shows the forthreads C interface and the required data structures to wrap pthreads' opaque `pthread_mutex_t` identifiers. The user passes the previously obtained mutex identifier (`mutex_id`) to the function. Forthreads in turn passes to `pthread_mutex_lock` the `pthread_mutex_t` object that had previously been stored in the `mutexes` array. Listing 6.2 shows the ISO C binding interface to Fortran 2003, and the implementation of the Fortran wrapping routine. It is not strictly necessary to use these Fortran routines as interface, but they free the user of dealing with the intricacies of dealing with Fortran-C interfaces.

```
! ------ ciface.h ------

interface
  subroutine thread_mutex_lock(mutex_id,info) bind(c)
  use iso_c_binding

  integer(c_int), intent(in)  :: mutex_id
  integer(c_int), intent(out) :: info

  end subroutine thread_mutex_lock
end interface

! ------ forthread.f03 ------

subroutine forthread_mutex_lock(mutex_id,info)
implicit none

include 'ciface.h'

integer, intent(in)  :: mutex_id
integer, intent(out) :: info

call thread_mutex_lock(mutex_id,info)

end subroutine forthread_mutex_lock
```

Listing 6.2 The forthreads Fortran 2003 wrappers built on top of the C interface shown in listing6.1.

6.1.2. Using forthreads in hybrid MPI/pthread programs

Different design patterns exist for combining distributed- and shared-memory parallelism.

The *SIMD* pattern uses multiple threads to distribute a large number of identical (and preferably independent) work items. Each thread executes the same subprogram on different data. This pattern is most prominently used in OpenMP, which employs preprocessor directives placed around sections of the code to be executed in parallel. The compiler then generates additional instructions to spawn and execute the threads. The same can also be achieved using pthreads (and hence forthreads). MPI is then used to parallelize the computation across multiple memory address spaces.

The *task parallelism* pattern assigns different tasks to different threads, executing possibly different code. A thread could for example be tasked with performing inter-process communication or message passing (e.g., using MPI) while other threads can run the program's main computations. Task-parallel threads can also be used to compute real-time in-situ visualizations, or to allow user interaction of a running program. Pthreads and forthreads offer the full flexibility required for task-parallel applications through their various interfaces for thread management and synchronization. Also OpenMP has in its recent versions gained support for task-level parallelism through `task` constructs, which continue to be improved.

In the *thread pool* pattern, finally a number of worker threads are typically created that receive work tasks through a queue. A master thread manages the creation and destruction of worker threads based on the workload and interprocess communication. Such systems are particularly useful when the workload of each (MPI) process may vary during the computation. OpenMP internally uses the thread pool pattern, but gives the programmer only limited freedom in adjusting the mode of operation through optional clauses to its preprocessor directives.

We demonstrate the use of forthreads by adding three threading extensions to the PPM library: multi-threaded particle-mesh interpolation, a multigrid Poisson solver with computation-communication overlap using a dedicated communication thread, and interactive computing with PPM using a runtime socket server running in a separate thread. These extensions allow

PPM to make better use of multi-core architectures, offering opportunities for improved scalability and usability.

6.1.2.1. Particle-mesh interpolation using forthreads

The current parallelization model of PPM foresees that one MPI process holds several subdomains of the decomposed computational domain. This can conveniently be taken advantage of to execute independent operations on the different subdomains in parallel using threads. The particle-mesh interpolation routines have hence been modified to execute on a single subdomain. The main interpolation routine spawns one thread per subdomain and executes the interpolations on the different subdomains in parallel (algorithm 6.1). The arguments to the interpolation routines must be passed in heap memory, instead of the call stack, because of forthreads' restriction to allow only one `INTEGER` pointer to be passed as an argument to the thread start routine. Since all subroutine arguments are identical for the different subdomains, and they are only *read* by the interpolation routine, it is sufficient to store only one set of arguments. The subdomain ID is passed as the sole argument to the thread. The threads are created and afterwards immediately joined, which amounts to a barrier at the end of the interpolation, ensuring all subdomains have been completely interpolated before the simulation proceeds.

We compare our forthreads approach with an OpenMP implementation. OpenMP offers a simple and efficient solution to this specific problem since we employed SIMD-type parallelization (listing 6.3). Both approaches are comparable in terms of the required code modifications. However, OpenMP allows for a more condensed syntax. More importantly, both implementations perform similarly well, as shown in figure 6.3. Furthermore, we compare the two shared-memory based approaches with a distributed-memory MPI implementation of particle-mesh interpolation. The MPI implementation is faster for linear and, to a lesser extent, $M'4$ interpolation. We ascribe the superior runtime to the fact that the operating system is able to optimize memory allocation to the MPI processes' processor-core assignment. The reduced memory-access times are particularly have a greater effect for operations with low computational intensity, such as linear interpolation. For multithreaded applications currently the thread-core assignment is not

Algorithm 6.1 Multi-threaded PPM particle-mesh interpolation routine.

1. Allocate derived type object for routine arguments
2. Allocate a threads array containing the IDs of created threads
3. Copy pointers to particle positions and properties, meshes, and parameters to routine argument object
4. For each subdomain
   a) Create a new thread passing the interpolation subroutine as start routine and the subdomain ID as routine argument
   b) store the thread ID in the threads array
5. For each subdomain
   a) Retrieve the associated thread ID and join the thread

taken into account for memory allocation. All timings were performed on an AMD Opteron 8380 using 8 threads on 8 cores and a problem size of $512 \times 512 \times 512$ with 1 particle per mesh cell. All benchmark code was compiled using GCC 4.6.2 and the `-O3` flag.

6.1.2.2. Multigrid Poisson solver with computation-communication overlap

We demonstrate forthreads' use as a Fortran library for shared-memory task parallelism by porting PPM's multigrid Poisson solver for heterogeneous platforms. Multigrid (MG) methods use a hierarchy of successively coarsened meshes in order to efficiently invert a matrix, resulting, e.g., from spatial discretization of a partial differential equation. The numerics part of the PPM library includes a MG solver for finite-difference discretizations of the Poisson equation. The solution is found by iteratively applying a linear system solver, such as the Gauss-Seidel method or Successive Over-Relaxation, restricting the error of the solution onto a coarser mesh, possibly applying more solver iterations, and finally interpolating back onto the orig-

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) &
!$OMP& FIRSTPRIVATE(lda,dxxi,dxyi,dxzi) &
!$OMP& SHARED(topo,store_info,list_sub,xp,up,&
!$OMP&        min_sub,max_sub,field_up)
  DO isub = 1,nsubs
    ! perform interpolation for subdomain isub
  END DO
!$OMP END PARALLEL DO
```

Listing 6.3 Listing showing the code required to shared-memory parallelize the particle-mesh subroutine in PPM using OpenMP.

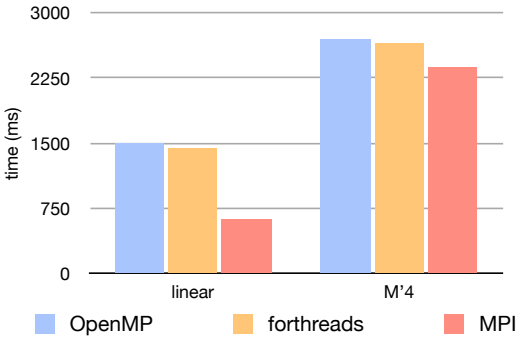


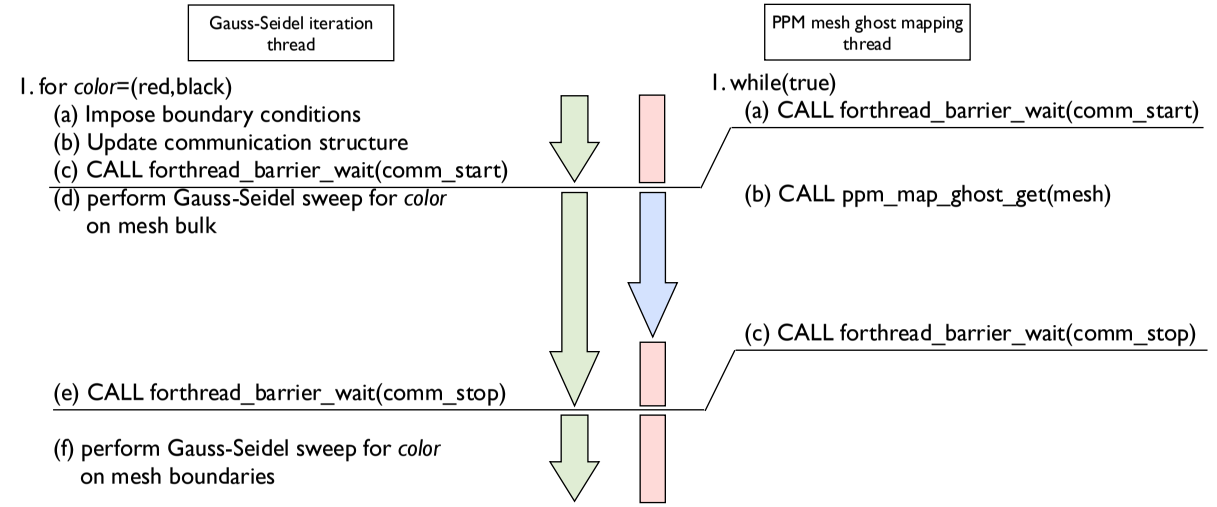
Figure 6.3. Timings for particle-mesh linear and $M'4$ interpolation using OpenMP (blue), forthreads (orange) and MPI (red). All timings were performed on an AMD Opteron 8380 using 8 threads on 8 cores. The used problem size is $512 \times 512 \times 512$ with 1 particle per mesh cell.

inal mesh. The MG method has its roots in the 1960s, but was popularized byBrandt [1977]. Today, MG solvers are widespread computational tools in science and engineering.

To make use of multicore platforms, we extend the current MG implementation in PPM by encapsulating the calls to PPM mesh-ghost mapping routines in a separate, concurrent thread. This enables overlapping computation and communication. Algorithm 6.2 describes the forthreads MG implementation. Apart from thread creation in the MG initialization routine, we only need to adapt the iterative solver (in this case a Gauss-Seidel method) and add a new routine encapsulating the calls to the PPM communication abstractions in a concurrent thread. The communication thread is executed in an infinite loop, waiting at the `comm_start` barrier. Then, the main computation thread enters the Gauss-Seidel routine and reaches the `comm_start` barrier, the communication thread becomes active. It updates the mesh ghost layers using communication from neighboring processes (using the PPM mapping abstraction). At the same time, the next solver iteration is performed on the bulk mesh, i.e., away from the boundaries that are currently being communicated. Both threads have to subsequently synchronize at the `comm_stop` barrier before the computation thread continues updating the mesh boundaries, based on the new ghost values.

Even though our implementation successfully overlaps MPI communication with computation, its runtime is higher than that of the MPI-only implementation. We believe the reason for this is two-fold: First, the newly added iteration index calculations that are necessary to filter boundary mesh points from the main bulk iterations incur a large overhead. The red-black Gauss-Seidel iterations on mesh bulks require initializing the mesh indices according to a number of state variables, which prevented them from being vectorized by the compiler. Pre-computing these mesh indices could potentially alleviate the problem and improve the efficiency of the multi-threaded solver routine. Second, PPM's ghost mapping communication schedule requires multiple communication rounds in order to prevent network conflicts and deadlocks. A 3D Cartesian topology on 8 processors, for example, requires 8 communication rounds. On 64 processors, 27 communication rounds are required. Consequently the volume-to-surface ratio of the sub-domains should be increased in order to mask the increased ghost mapping time with a matching amount of bulk-mesh computation

Algorithm 6.2 PPM numerics multigrid red-black Gauss-Seidel routine with forthreads. The arrows visualize the control flow and thread states (green/blue: run, red: wait). The communication thread (right) is executed in an infinite loop. The computation thread (left) performs a given number of iterations; we show here one iteration.



time.

6.1.2.3. Interactive computing with the PPM library and forthreads

Many modern applications use task parallelism and threads to allow for quick, responsive interaction with the user. We use forthreads together with a POSIX internet sockets Fortran wrapper to provide a prototypic server instance allowing remote clients to connect and to control a running PPM simulation. In particular, this server is capable of handling an arbitrary number of concurrent client connections. Such server extensions allow PPM to be not only directly controlled by the user, but also to communicate with other applications, such as visualization tools, cluster management, databases, or web browsers.

In order to extend PPM with an internet server thread, we first build a simple wrapper for the POSIX internet socket API for Fortran (fsocket), abstracting some of the intricacies of the sockets API. This wrapper is by no means a complete wrapper implementation, as it is specifically geared toward providing the necessary functionality for building TCP internet servers. It provides the following functions:

- `fsocket_init()` must be called before any other fsocket routine. It creates and initializes an internal data structure for mainting the open connections and file descriptors
- `fsocket_server_create()` is a shortcut for the `socket()` and `bind()` functions. It creates a socket address structure and requests a file descriptor.
- `fsocket_listen()` wraps the `listen()` function indicating that the caller is ready to accept incoming connections.
- `fsocket_accept()` creates a new client address object, then calls the `accept()` function, which returns as soon as an incoming connection is to be established. Once the connection is established, the client address and file descriptor are stored in the internal data structure and a unique ID is returned to the caller.

- `fsocket_read()` and `fsocket_write()` are simple wrappers for the `read()` and `write()` functions. They allow reading and writing character buffers from/to the socket.
- `fsocket_close_conn()` and `fsocket_close_server()` wrap the `close()` function either passing the client connection file descriptor or the server file descriptor.

In order to extend this library into a more general-purpose Fortran sockets interface, one should at least separate the `socket()` and `bind()` functions, provide generic interfaces for building socket address structures, and extend the `fsocket_read()` and `fsocket_write()` routines with a type argument allowing arbitrary Fortran primitive types, similar to MPI's communication routines.

We extend the PPM core library by adding a simple Fortran module providing a single user-facing subroutine. This routine is responsible for spawning a new thread (using the forthreads library) that creates an internet server socket and enters the main server listen loop. Whenever a new client connection is established, this loop advances by one iteration and creates an additional thread for handling the new client connection. This mechanism ensures that the running PPM application remains responsive to all connecting clients while at the same time continuing its normal operation. The mode of operation of this PPM server is summarized in (figure 6.4).

Since the PPM server is executed per process, it can be made available in every MPI process of a parallel PPM instance, allowing the end-user to address and manipulate specific MPI processes of a running PPM simulation. An example is shown in (6.5), connection to a PPM process running on the local host and asking what the dimensionality of the currently solved problem is.

6.1.3. Summary and Conclusion

We have developed a comprehensive binding of the POSIX threads API to Fortran 2003 that gives the programmer access to almost all thread management functions and all thread synchronization constructs. Using this library the programmer is only exposed to native Fortran interfaces.

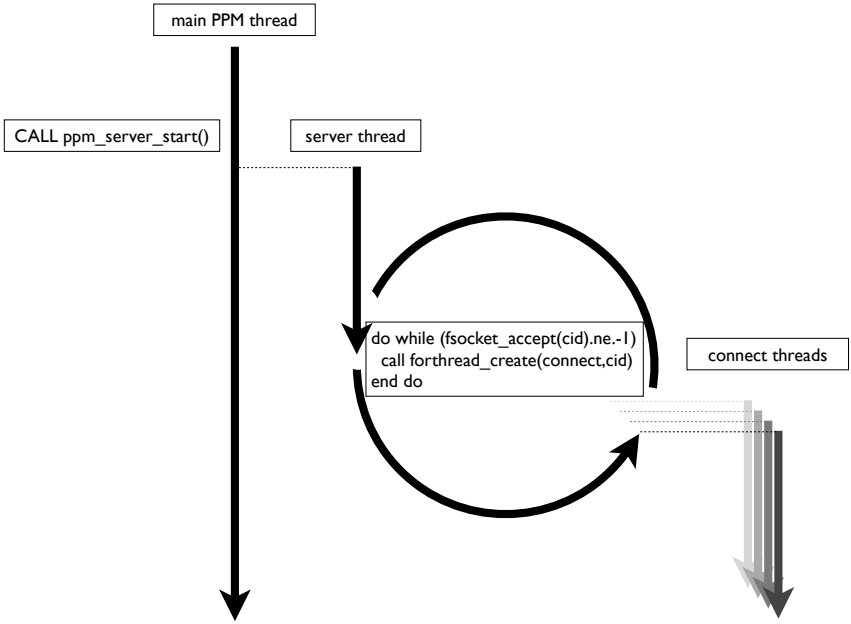


Figure 6.4. PPM server. The arrows visualize the thread control flow; dashed lines are thread spawns. The PPM server creates a thread for the server listen loop and for each newly established connection.

```
$ telnet 127.0.0.1 1337
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
PPM command server 0.1 - Welcome
> hello
Hi there!
> dim
ppm_dim is: 2
> exit
bye
Connection closed by foreign host.
```

Figure 6.5. Log of example PPM command server session.

Forthreads extends previous implementations, most notably by Hanson et al. [2002], Nagle [2005]. We showed the versatility of forthreads in three examples using different design patterns. All examples extended the PPM library using forthreads in order to provide new functionality and multicore support. First, we have extended the existing particle-mesh interpolation routines to spawn one thread per subdomain to be executed on separate processor cores. The benchmarks showed that our implementation yields a comparable performance to OpenMP, while offering additional control over the threads. This comes, however, at the cost of a slight increase in code complexity. Second, we have redesigned and ported the existing implementation of the multigrid Poisson solver of the PPM library to use threads. The thread-enabled multigrid solver maintains a separate communication thread, allowing overlapping computation with communication. The current implementation, however, has significant shortcomings over the original implementation in terms of time efficiency. We do, however, suspect that this is not due to the use of forthreads, but is caused by the loss of code vectorization due to the index algebra needed to separate bulk mesh nodes from boundary nodes. Third, we have implemented a Fortran wrapper for the POSIX internet socket API and a new PPM module providing a control server for PPM simulations capable of handling several simultaneous client connections. Mixed shared and distributed-memory parallel programming has in several instances shown significant improvements over pure distributed-memory parallelizations [Winkel et al., 2012, Song et al., 2009, Madduri et al., 2009]. The forthreads library is intended to offer a simple yet powerful alternative to existing parallelization frameworks for shared-memory parallelism in Fortran 2003. Forthreads is complete in the sense that it provides native Fortran 2003 interfaces to all POSIX threads routines where this is possible. Forthreads also maintains the opacity of the internal pthreads types and data structures, as required by the POSIX standard. Together with the fsocket wrapper for the POSIX internet socket API, we believe that forthreads will be a useful tool for developing numerical software for multicore platforms.

6.2. An OpenCL implementation of particle-to-mesh and mesh-to-particle interpolation in 2D and 3D

[1]Due to their regular computational structure and fine granularity, particle-to-mesh and mesh-to-particle interpolations admit SIMD (Single Instruction Multiple Data [Hockney and Jesshope, 1981]) parallelism on streaming multiprocessors, such as GPUs. Different approaches to particle-mesh interpolation on GPUs were discussed and compared by Stantchev et al. [2008] in the context of a 2D plasma physics PIC code using CUDA. A very efficient implementation of 2D particle-to-mesh interpolation on a GPU was presented by Rossinelli and Koumoutsakos [2008]. Using OpenGL, they reported real-time incompressible fluid mechanics simulations using vortex methods [Koumoutsakos, 1997, Cottet and Koumoutsakos, 2000] at 25 frames per second on a 1024×1024 mesh, representing a 26-fold speedup over the highly optimized reference CPU code. The same authors later extended their work to simulate flows in bounded, complex geometries in 2D [Rossinelli et al., 2010] and reported a 100-fold speedup of OpenGL over CUDA when using the OpenGL point-sprite primitives and blending techniques. A GPU implementation of 2D mesh-to-particle interpolation has also been provided by Rossinelli et al. [2011] in both OpenCL and CUDA. For double-precision arithmetics, they reported 2 to 3-fold speedups of the GPU implementation over a 16-threaded CPU implementation and a 20 to 45-fold speedup over a single-thread CPU implementation. Madduri et al. presented another 2D implementation of particle-mesh interpolation in a particle-in-cell (PIC) code [Madduri et al., 2011]. Their implementation uses CUDA, particle binning, grid replication, and texture memory, but barely provides any speedup over an optimized multi-threaded CPU Implementation, demonstrating the limits of GPU-accelerated interpolation. Recently, Conti et al. presented a complete OpenCL implementation of a 2D finite-time Lyapunov exponent computation with an up to 30-fold speedup over their single-thread CPU reference code; They also compared the performance of a GPU and an APU [Conti et al., 2012].

[1]This work has been done together with Ferit Buyukkececi, who has been involved in design, implementation and benchmarking of the code.

While purpose-made GPU implementations can offer impressive speedups, they typically suffer from low programmer productivity and poor performance portability. Libraries and generic algorithms are hence a recent trend in the field [Hönig et al., 2010, Du et al., 2012], even though they normally entail a performance toll when compared with specialized solutions. Here we follow this trend and build on the above-mentioned prior works in GPU-accelerated particle-mesh interpolation in order to present a portable OpenCL [Du et al., 2012, Khronos, 2009] implementation of a generic algorithm for particle-to-mesh and mesh-to-particle interpolation in both 2D and 3D. Our algorithm is generic in the following ways, without implying superior performance in all cases: (1) It is free of assumptions about the (typical or maximal) number of particles per mesh cell. (2) It does not expect the input data to be stored or sorted in any particular way. (3) It works with arbitrary numbers of particle properties and vector-valued fields. (4) It works in 2D and 3D, single precision and double precision. (5) It works with arbitrary mesh sizes that do not have to be powers of two. Moreover, the OpenCL implementation is portable across hardware platforms (multi-core CPUs and GPUs from different vendors). Parallelism is achieved by first reordering the particle data according to access patterns of threads in workgroups. Moreover, particle and mesh values are stored such as to distribute particles in the same mesh cell among different buffer frames. Together with a mesh-padding technique, this avoids race conditions and provides coalesced global memory access in strides. The mesh is additionally decomposed into blocks to achieve data locality for high cache hit rates. Our approach avoids atomic operations, which have been reported to harm performance on the GPU [Madduri et al., 2011]. We show that a common parallelization strategy can be used for both particle-to-mesh and mesh-to-particle interpolation, albeit not matching the performances of the respective specialized implementations by Rossinelli et al. [2010, 2011]. Our approach, however, naturally extends to 3D and we present and benchmark a fully 3D implementation of particle-to-mesh and mesh-to-particle interpolation on the GPU. The implementation is integrated and available in the PPM library for parallel hybrid particle-mesh simulations.

6.2.1. GPU Programming with OpenCL

The Open Computing Language (OpenCL) is a programming framework for heterogeneous multi-core computing devices, including GPUs, APUs, and CPUs [Khronos, 2009]. It provides a higher level of hardware abstraction than OpenGL, while still providing enough hardware control to allow for efficient implementations. OpenCL does not expose low-level graphics primitives, but instead provides data structures and operations suitable for general-purpose programs. OpenCL can be used on a wide variety of operating systems and hardware platforms, also extending beyond GPUs. OpenCL achieves portability through a hierarchy of abstraction layers: the platform model, the execution model, the programming model, and the memory model. The execution model describes the computing platform as a host and a collection of devices. In our case, the host is the CPU and the devices are the streaming multiprocessors of the GPU. The host executes the *host program*, which creates a context for the devices and manages the execution of *kernels* on the devices. When a kernel is launched by the host program, the OpenCL devices execute many instances of this kernel, called *work items*. Each work item performs a set of instructions specified by the kernel at one point in index space, thus processing different data items in parallel. Work items that execute on the same set of processing elements, *the compute unit*, form a *workgroup*. The work items of a workgroup share resources, such as the on-chip memory of the compute unit.

Like most parallel computing environments, OpenCL provides synchronization support in the form of *barriers*. There are two types of barriers: the command-queue barrier and the workgroup barrier. Here, we only use workgroup barriers in order to synchronize the work items within a workgroup. A workgroup barrier dictates that no work item must leave the barrier before all work items have entered it. Synchronization of work items belonging to different workgroups is not possible using workgroup barriers. This global synchronization of work items can be achieved by atomic operations or semaphores, or by orchestration of the workgroups from the host code. Such global synchronization, however, usually leads to significant performance losses.

GPUs are many-core architectures consisting of collections of identical Streaming Multi-Processors (SMP) that execute many instances of the

same kernel in a SIMD control structure. They are massively parallel, but with limited resources per core [Kirk and Hwu, 2010]. Mapping the OpenCL execution model onto a GPU, each work item is executed in a separate *thread*, and all threads hosting work items from the same workgroup execute on the same SMP. Therefore, work items must perform simple tasks on small amounts of data in order to provide sufficient granularity. Threads are executed in *warps* (called *wave fronts* for ATI GPUs), which are collections of threads that are executed simultaneously. Control flow divergence among threads within a warp results in serialization [NVIDIA, 2010] and has to be avoided. Hence, work items should be homogeneous, i.e., perform the same tasks. Homogeneous, fine-grained work items enable the GPU to apply fast *context switching*. GPUs with hardware support for context switching change between threads in a single clock cycle in order to hide data-access latency of a thread by computation of another. One of the most important performance determinants on GPUs is the memory access pattern of the threads within a warp. A memory transaction to or from the global device memory is a multi-word burst transaction. Offsets and non-unit stride access of a warp result in additional transactions and degrade the memory bandwidth. Thus, work items and the data they access must be aligned and coalesced.

6.2.2. Method

In order for an algorithm to perform well on a GPU, it has to meet a number of requirements. First, it should rely on fine-grained kernels that can be parallelized over a large number of small work items. Second, it has to avoid race conditions that would require synchronization. Third, it has to avoid conditional statements that lead to control flow divergence. Fourth, the algorithm and the data structures must guarantee coalesced and aligned global memory access and use local memory for frequent read/write operations within the same memory region.

In the following, we outline the design of a generic streaming-parallel particle-to-mesh/mesh-to-particle interpolation algorithm in 2D and 3D as guided by these design principles. Particles are assigned to work items that loop over mesh nodes within the support of the interpolation function. In particle-to-mesh interpolation, work items scatter the particles' contribu-

tions onto the mesh nodes, whereas in mesh-to-particle interpolation they gather contributions from the mesh nodes. As work items iterate over mesh nodes, the interpolation weights are only recomputed along dimensions that do change. Different parallelization strategies are discussed in the following subsection.

6.2.2.1. Strategies for interpolation on the GPU

Stantchev et al. [2008] distinguish two strategies for particle-mesh interpolation on the GPU: the *particle-push* and *particle-pull* strategies. In the particle-push strategy, particles are assigned to work items that scatter the particle contributions onto the mesh nodes. In the particle-pull strategy, mesh nodes are assigned to work items that gather particle contributions. Both strategies have advantages and disadvantages: Particle-push allows the work item to dynamically compute a fixed-length $\mathcal{M}(p)$ (set of mesh nodes contributing to a particle p, c.f. Fig. 1.2 and Eq. 1.15) from the particle coordinates. Moreover, the work item can reuse some interpolation weights that were already computed earlier when traversing mesh nodes along the same dimension. However, the particle-push strategy causes memory collisions as concurrent work items may attempt to write to the same mesh node simultaneously. This is avoided in the particle-pull strategy, where each mesh node can be updated by only one work item. The disadvantage of the particle-pull strategy is that it is costly to compute $\mathcal{P}(m)$ (the set of particles contributing to a mesh node m, c.f. Fig. 1.2 and Eq. 1.14) unless the particles are arranged in a regular spatial pattern (e.g., on a grid). Furthermore, the interpolation weights always have to be recomputed for each particle.

Similar trade-offs also exist in mesh-to-particle interpolation. This can be seen by analogously defining *mesh-push* and *mesh-pull* strategies: In the mesh-push strategy, each work item scatters the contributions of mesh nodes onto the particles within the support of the interpolation weights. A clear disadvantage of this strategy is the high cost of computing $\mathcal{P}(m)$ if particles are not organized in a regular spatial pattern. More importantly, a particle p might simultaneously be updated by concurrent work items from m and $\widetilde{m}$ if p resides both in $\mathcal{P}(m)$ and $\mathcal{P}(\widetilde{m})$. In the mesh-pull strategy, all work items are launched over particles and visit the mesh nodes in $\mathcal{M}(p)$,

	particle-		mesh-	
	pull	*push*	*pull*	*push*
No memory collisions	$\surd$	$\times$	$\surd$	$\times$
Fast computation of $\mathcal{P}(m)$ and $\mathcal{M}(p)$	$\times$	$\surd$	$\surd$	$\times$
Re-using interpolation weights	$\times$	$\surd$	$\surd$	$\times$

Table 6.1. Qualitative comparison of parallelization strategies for particle-to-mesh (columns 2 and 3) and mesh-to-particle (columns 4 and 5) interpolation.

which is easily computed thanks to the regular geometry of the mesh. Each work item then accumulates the contributions of the mesh nodes in $\mathcal{M}(p)$ to $\omega(p)$, which is free of memory collisions.

The advantages and disadvantages of these four parallelization strategies are qualitatively summarized in Table 6.1. From this, we conclude that the particle-push and mesh-pull strategies are preferable for particle-to-mesh and mesh-to-particle interpolation, respectively. A common feature of both strategies is that work items are defined over particles. This allows using the same algorithm for both interpolations by storing the particle positions and properties in the private memories of the work items, reserving the shared memory for the mesh data. Since mesh node positions do not need to be stored, this is an additional advantage given the limited size of the shared memory. In the following, we present data structures and algorithms for generic particle-push/mesh-pull interpolation on the GPU in 2D and 3D.

6.2.2.2. Data structures

The main goal in designing the data structures is to guarantee coalesced and aligned global memory access and data locality. At the same time, race conditions and atomic operations are to be avoided.

Particle data. The particle positions and strengths are stored in two linear buffers, named `particle_pos` and `particle_str`, respectively. Particles in the same mesh cell are distributed across different frames of these buffers, which we call *domain copies* (Fig. 6.6). We then decompose each domain

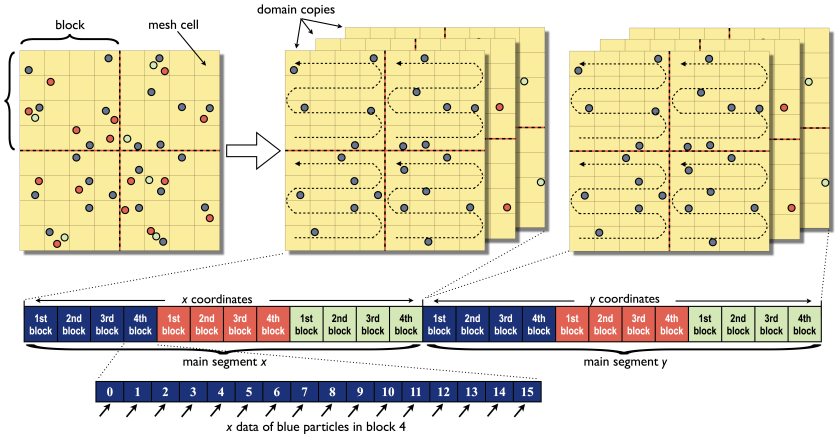


Figure 6.6. Particle data are stored in buffer segments, blocks, and domain copies according to their dimension, mesh cell index, and number of fellow particles within the mesh cell. Particles within the same mesh cell are stored in different domain copies as highlighted by color. Each domain copy is subdivided into blocks (red dashed lines) that are stored consecutively in the buffer. Within each block, particles are numbered in the order shown by the dashed arrows. Data along different space dimensions are stored in separate main segments of the buffer.

copy into blocks (red lines in Figure 6.6) and store the particle data of each block consecutively with an ordering as indicated by the dashed arrows in Figure 6.6. Taking this decomposition approach one step further, we also store the particle data along different dimensions in separate segments of the buffer, called *main segments*. This renders our strategy dimension-oblivious and guarantees memory access in unit stride.

The buffers are constructed as follows: We first count the number of particles in each mesh cell and store it in an `np_cell` buffer. Counting is done by atomically incrementing the elements of `np_cell` using the `atom_inc` instruction of OpenCL, which ensures sequential access to `np_cell`. This is the only time we use atomic operations in the whole workflow, and it enables us to directly assign indices to the particles according to the new ordering. Using these indices, we calculate and store the index of the mesh cell within

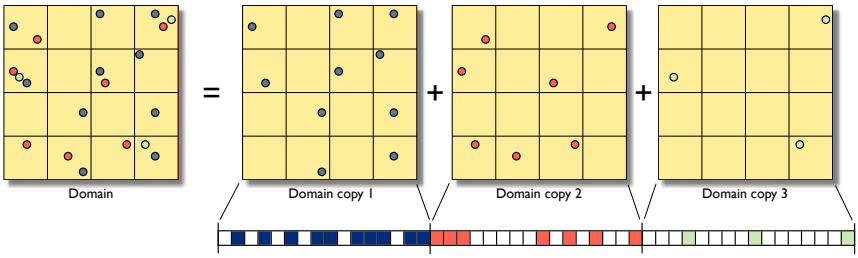


Figure 6.7. Illustration of the use of domain copies and dummy particles in order to avoid conditional statements and guarantee coalesced memory access. See main text for details.

which each particle resides in the *cell list* `p_cell`. Then, the `np_cell` buffer is reduced in parallel to find the maximum number of particles in any mesh cell, $p_{\max}$. The required size of the particle data buffers is then given by $p_{\max} \times d \times n_{\text{cell}}$, where d is the number of dimensions and n_{cell} is the total number of mesh cells in the domain. The buffers then contain d main segments, each holding the particle data along one dimension. Each main segment further contains $p_{\max}$ domain copies (Fig. 6.7), which are in turn decomposed into blocks. Each mesh cell corresponds to one memory location in the buffer; empty mesh cells are represented as "dummy particles" of zero strength, as shown in Figure 6.7. This guarantees coalesced memory access in unit strides since the particles within each block are consecutively numbered. In this scheme, inhomogeneities in the particle distribution lead to memory and compute overhead. In our experience, however, this overhead is amortized by the performance gain from the resulting regular, coalesced memory access pattern. Moreover, adaptive meshes and remeshing are used in practical simulations to limit particle inhomogeneity [Bergdorf et al., 2005].

Mesh data. Mesh data are stored in a linear buffer `mesh_prop`, which is again decomposed into *main segments* by dimension and mesh cell *blocks*, as described above. In order to avoid the memory collisions that are possible in a particle-push strategy, and in order not to introduce sequential parts into the algorithm, we replicate mesh nodes that are closer to any block boundary than the support radius of the interpolation function. This is

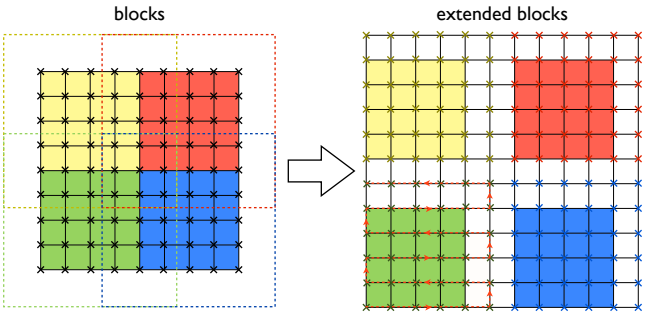


Figure 6.8. Mesh nodes close to the boundary of a block are replicated in order to avoid concurrent writes and race conditions. **Left:** the dashed squares indicate the areas within which mesh nodes are influenced by particles in the block of the same color. **Right:** extended blocks are defined by replicating mesh nodes in overlapping influence regions. The red dashed arrow shows the storage order of the mesh nodes inside the green extended block.

illustrated in 2D in Figure 6.8. A block including its ghost layer of replicated nodes is called an *extended block*. Mesh nodes are traversed in the same order as mesh cells, as shown by the red arrow in the green block in Figure 6.8. After a completed particle-to-mesh interpolation, the contributions on replicated mesh nodes are aggregated by stitching the mesh back together in a post-processing step.

The total numbers of mesh cells in the computational domain without and with ghost layers are termed `cell_size_int` and `cell_size_ext`, respectively. The total problem size is given by `cell_size_ext`. This size can be arbitrary, as it depends on the domain decomposition done by the PPM library. OpenCL, however, requires that the domain size be an integer multiple of the extended block size. We hence pad the domain as illustrated in Figure 6.9.

6.2.2.3. Mapping of the data structures into OpenCL

The above-defined data structures are mapped onto the OpenCL execution model by assigning one mesh cell per work item. The data of all $p_{\max}$

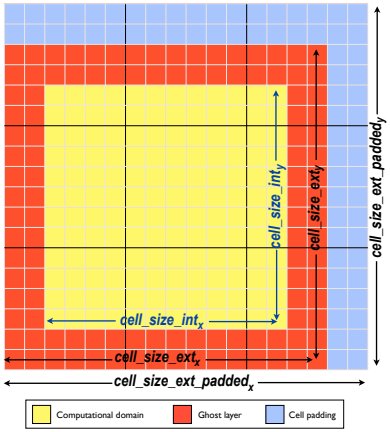


Figure 6.9. Sizes of the computational domain (`cell_size_int`), including ghost layers (`cell_size_ext`), and after padding with extra cells (`cell_size_ext_padded`) for consecutive memory access. Extended blocks are shown by black lines.

particles (some of them potentially dummy particles) inside that mesh cell are stored in the private memory of the work item; the mesh data are kept in shared memory. A workgroup is then defined as the collection of all work items having mesh cells that belong to the same block. The data structure and storage order introduced in Figure 6.6 then guarantee that work items always access the global memory in a coalesced fashion. The mesh padding shown in Figure 6.8 avoids race conditions across workgroups and dispenses with the need for inter-workgroup synchronization and atomic operations.

The block size is chosen according to the GPU hardware. It depends on the number of shared memory banks and SMPs the GPU has, and it represents a memory trade-off. On the one hand, the blocks should be small enough for the data to fit into the shared memory of each workgroup. On the other hand, they should not be too small in order to limit the memory overhead stemming from the ghost layers around the extended blocks (Fig. 6.8). To avoid shared memory bank conflicts, the blocks should moreover contain at least as many cells along the x-direction (i.e., the leading dimension of the loop) as the GPU has shared memory banks. The specific settings for

the hardware used here are described in section 6.2.4.

6.2.2.4. Interpolation algorithms for the GPU

In particle-mesh interpolation, each workgroup is assigned a block of mesh cells with one work item per cell. Every work item then loops over the $p_{\max}$ particles (i.e., the domain copies) in its cell and scatters their strength onto the mesh nodes within the support radius of the interpolation function, i.e., within the extended block. The redundant computation of dummy particles turns out to be more efficient than conditional statements on the GPU. Mesh-to-particle interpolation uses an analogous parallelization strategy, where dummy particles are purged from the `particle_str` buffer before reading the data back from the device.

Particle-to-mesh interpolation. In particle-to-mesh interpolation, each work item loops over the particles in one mesh cell and scatters their strengths onto the mesh nodes around in a nested loop over dimensions. Since domain copies and main segments are stored in different frames of the buffer, memory conflicts are avoided. The number of inner-loop iterations required is given by the size of the support of the interpolation weights W, $\mathrm{supp}(W)$, i.e., the number of mesh nodes where $W \neq 0$. In each iteration, all work items within a workgroup assign particle contributions onto mesh nodes in the same direction. A workgroup barrier is then used to synchronize all work items before assigning into the next direction. This synchronization ensures that no concurrent writes onto the same mesh node occur. Figure 6.10 illustrates this "synchronous swimming" of work items for the example of a linear interpolation function (support: 4 mesh nodes). This is repeated $p_{\max}$ times until all particles have been assigned. The general kernel is given in Algorithm 6.3, the complete workflow for particle-to-mesh interpolation in Algorithm 6.4.

Mesh-to-particle interpolation Mesh-to-particle interpolation follows an analogous parallelization strategy with workgroups being assigned mesh cell blocks, and work items particles within single mesh cells. The work items then gather in parallel the contributions from the mesh nodes within

Algorithm 6.3 Particle-to-mesh interpolation kernel

1. $w \leftarrow$ work item ID
2. $\chi \leftarrow$ indices of mesh cells assigned to w
3. **for** i from 1 to $p_{\max}$
   a) $p \leftarrow$ index of the i^{th} particle in χ
   b) $\mathbf{x} \leftarrow$ coordinates of p
   c) **for** $\kappa \in \text{supp}(W)$
      i. $\mu \leftarrow \chi + \kappa$ (shift of mesh node in target)
      ii. $M \leftarrow$ index of the mesh node pointed to by μ
      iii. **for** each particle property ω_i
         A. $\omega_i(M) \leftarrow \omega_i(M) + W(\mu, \mathbf{x})\omega_i(p)$
      iv. workgroup barrier

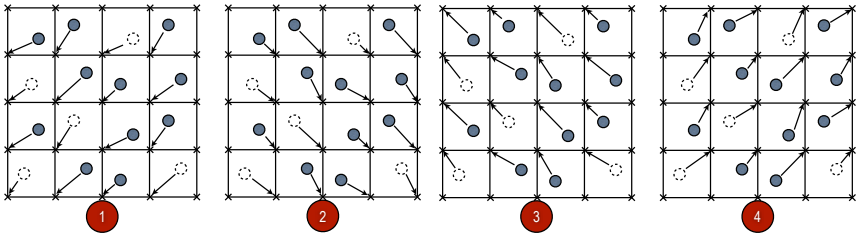


Figure 6.10. Particle-to-mesh interpolation example in 2D with a linear interpolation function (support size 4). In each of the 4 iterations, the work items in the same workgroup (block) update the mesh nodes in the same direction in order to avoid concurrent writes. Dummy particles (dashed circles) avoid conditional statements. The 4 stages are repeated $p_{\max}$ times until all particles have been assigned.

Algorithm 6.4 Overall particle-to-mesh interpolation workflow

1. Copy particle positions and strengths from host to device
2. Initialize `np_cell`
3. Determine cell indices of particles and store them in the cell list `p_cell`
4. Reduce `np_cell` to compute $p_{\max}$
5. Store particle coordinates in `particle_pos`
6. Store particle strengths in `particle_str`
7. Allocate and initialize extended mesh with replicated mesh nodes
8. Launch particle-mesh interpolation kernel
9. Stitch duplicated mesh nodes
10. Read back mesh nodes from device

Algorithm 6.5 Mesh-to-particle interpolation kernel

1. $w \leftarrow$ work item ID
2. $\chi \leftarrow$ indices of mesh cells assigned to w
3. **for** i from 1 to $p_{\max}$
   a) $p \leftarrow$ index of the i^{th} particle in χ
   b) $\mathbf{x} \leftarrow$ coordinates of p
   c) **for** $\kappa \in \text{supp}(W)$
      i. $\mu \leftarrow \chi + \kappa$ (shift of mesh node in target)
      ii. $M \leftarrow$ index of the mesh node pointed to by μ
      iii. **for** each particle property ω_i
         A. $\omega_i(p) \leftarrow \omega_i(p) + W(\mu, \mathbf{x}) \cdot \omega_i(M)$

the support of the interpolation weights W. Unlike in particle-mesh interpolation, however, we do not need to synchronize the work items since particle strengths are stored in the private memory of the respective work item, and mesh nodes have been replicated as outlined in Sec. 6.2.2.2. Algorithm 6.5 shows the mesh-to-particle interpolation kernel. In each iteration, the particles receive contributions from mesh nodes in the same direction. This is illustrated in Figure 6.11 for the example of a linear interpolation function with 4 mesh nodes in its support. It is repeated $p_{\max}$ times until all particles have been considered. The complete workflow of mesh-to-particle interpolation is given in Algorithm 6.6.

6.2.3. Integration in the PPM Library

Using the present OpenCL implementation of particle-to-mesh and mesh-to-particle interpolation, we extend the PPM library to multiple levels of parallelism. On the coarsest level, sub-domains are assigned to MPI processes that operate in separate memory address spaces. On the finest level, the light-weight GPU threads parallelize over the individual mesh cells

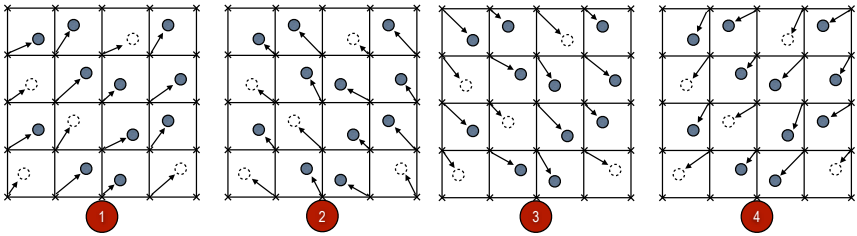


Figure 6.11. Mesh-to-particle interpolation example in 2D with a linear interpolation function (support size 4). In each of the 4 iterations, the work items in the same workgroup (block) accumulate on their particles the contributions from the mesh nodes in the same direction. Dummy particles (dashed circles) avoid conditional statements. The 4 stages are repeated $p_{\max}$ times until all particles have been considered.

Algorithm 6.6 Overall mesh-to-particle interpolation workflow

1. Copy particle positions and mesh node values from host to device
2. Initialize `np_cell`
3. Determine cell indices of particles and store them in the cell list `p_cell`
4. Reduce `np_cell` to compute $p_{\max}$
5. Store particle coordinates in `particle_pos`
6. Construct extended mesh with replicated mesh nodes (for memory stride and coalesced access)
7. Launch mesh-to-particle interpolation kernel
8. Collect strengths of real (non-dummy) particles
9. Read back particle strengths from device

within a sub-domain. Due to the presence of ghost particles, all information is locally available and no additional MPI communication is incurred by using the present OpenCL implementation. This extends the present implementation to multi-GPU settings.

Since PPM is a general-purpose library, the particle data cannot be assumed to be sorted or arranged in any specific way when entering the interpolation routines. Also, PPM's internally used object-oriented data structures do not directly map onto the OpenCL memory model. This requires a number of "wrapper" routines, as illustrated in Figs. 6.12 and 6.13 in yellow, and several pre- and post-processing kernels (blue boxes *other than* the actual interpolation kernels in Figs. 6.12 and 6.13).

Particle-to-mesh interpolation (Fig. 6.12) starts by re-numbering the particles such that consecutively indexed particles are located in the same sub-domain. This is necessary because a process can be assigned multiple sub-domains. The re-numbering allows applying the GPU kernels locally per sub-domain on contiguous chunks of memory. The second step consists of allocating and populating the particle and mesh data buffers as described in Sec. 6.2.2.2. For each sub-domain in an MPI process, the OpenCL kernels are then run (potentially in parallel if multiple sub-domains and multiple GPUs are available on a compute node). This entails first copying the buffer data from the host memory to the device memory. Once on the GPU, the particles are sorted into mesh cells as described in Sec. 6.2.2.2. Then, Algorithm 6.3 is used before the replicated mesh nodes are stitched back together and the results copied back to the host memory. Finally, the results are copied from the flat buffers into the PPM mesh objects.

Mesh-to-particle interpolation (Fig. 6.13) proceeds analogously. The main difference is that the duplicated mesh nodes don't need to be stitched together after interpolation. Instead, the dummy particles need to be removed from the GPU buffer before the results are transferred back to the host memory.

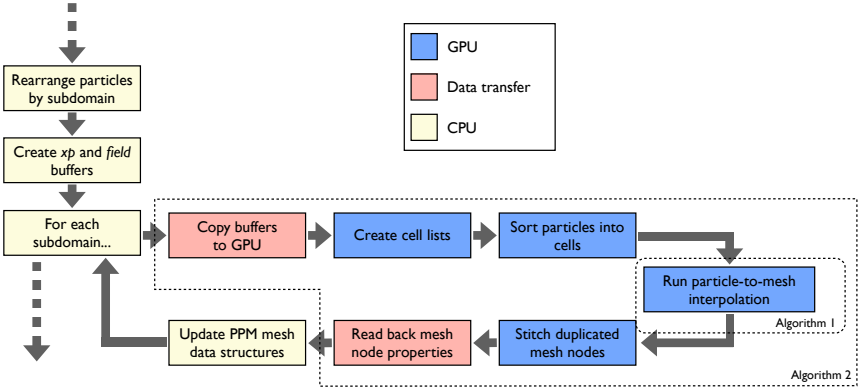


Figure 6.12. Workflow scheme for GPU-accelerated particle-to-mesh interpolation in the PPM library.

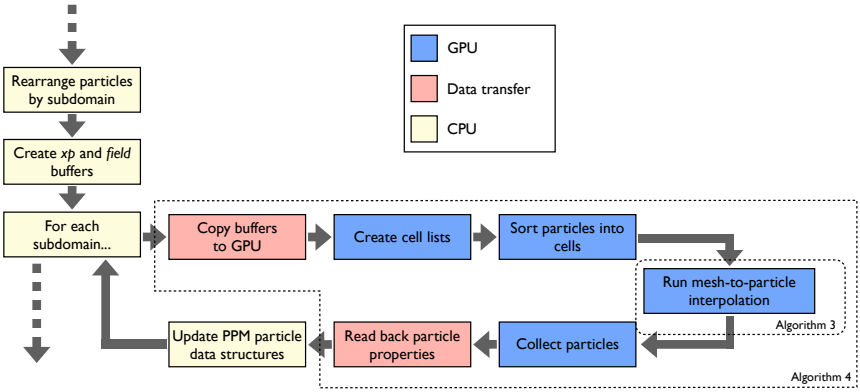


Figure 6.13. Workflow scheme for GPU-accelerated mesh-to-particle interpolation in the PPM library.

6.2.4. Benchmarks

We benchmark the accuracy and runtime of the presented algorithms in 2D and 3D using both single and double-precision floating-point numbers. For the benchmarks we initialize one particle on each mesh node in the unit square and perturb the particle positions by uniform random numbers in $[-2\mathbf{h}, 2\mathbf{h}]$. This leads to a quasi-random particle distribution with 0 to 16 particles per mesh cell. For particle-to-mesh interpolation we sample the function $f(\mathbf{x}) = \exp(-\|\mathbf{x} - 1/2\|_2^2/15)$ at the particle positions and interpolate the resulting particle strengths to the mesh nodes. The error is defined at each mesh node as the difference between the interpolation result and the exact value of g at the location of that mesh node. For mesh-to-particle interpolation the same function is sampled at the mesh nodes and the error after interpolation is analogously defined on the particles.

We benchmark the OpenCL implementation on a NVIDIA Tesla C2050 GPU consisting of 448 CUDA cores organized into 14 SMPs with 1030 GFLOP/s single-precision and 515 GFLOP/s double-precision peak performance and 3 GB GDDR5 memory with ECC disabled for the benchmarks. The peak memory bandwidth of this GPGPU card is 144 GB/s. For comparison, and to demonstrate the portability of the OpenCL implementation, we also benchmark it on an ATI "Cayman" Radeon HD 6970 GPU featuring 1536 stream processors with 2.7 TFLOP/s single-precision and 683 GFLOP/s double-precision peak performance and 1 GB GDDR5 memory with a peak bandwidth of 176 GB/s. As a baseline we use both the optimized sequential Fortran 90 implementation available in the PPM library, as well as an OpenMP-parallelized version of the same PPM routines. All CPU code is compiled with the GCC Fortran compiler version 4.6.2 using the `-O3` optimization flag. Both CPU versions (sequential and multi-threaded) are run on an 8-core AMD FX 8150 at 4.2 GHz with 16 GB DDR3 SDRAM.

In order to minimize bank conflicts in the GPU's shared memory, and hence maximize the effective bandwidth, the block sizes (cf. Sec. 6.2.2.2) are set according to the number of shared memory banks and the maximum number of work items per compute unit. Both GPU devices tested have 32 shared memory banks. We hence always use 32 mesh cells along the x-direction.

The NVIDIA Tesla C2050 GPU can process up to 1024 work items in a workgroup, but can launch 1536 threads per compute unit. We hence choose 512 work items per workgroup, in order to launch three workgroups with 100% thread utilization. Thus, the block size is set to 32×16 in 2D and $32 \times 4 \times 4$ in 3D. The maximum workgroup size for the ATI "Cayman" Radeon HD 6970 GPU is 256 and we can launch 1536 threads per compute unit. Since 256 is a divisor of 1536, 100% occupancy is always guaranteed. Setting the number of work items in the x-direction to 32, we use block sizes of 32×8 in 2D and $32 \times 4 \times 2$ in 3D.

For the timings, we measure the runtime of GPU-based interpolation, consisting of all GPU (blue) stages of the workflows shown in Figs. 6.12 and figure 6.13. Additionally, we measure the time to copy the data to and from the device memory (red). The sum of these times is compared with the runtime of the sequential and multi-threaded PPM implementations running on the CPU, where sorting of particles and re-ordering of data is not necessary. The *speedup* is defined as the ratio between the CPU wall-clock time and the GPU wall-clock time. For each interpolation kernel and GPU platform, we also measure the sustained performance in GFLOP/s, only counting floating-point multiply and add operations, and use this to evaluate the *efficiency* of the implementations, defined as the fraction of the theoretical peak performance of the respective GPU that is actually sustained by the interpolation kernel. We only provide GFLOP/s rates for the actual interpolation kernels (i.e., Algorithms 6.3 and 6.5), but not for the other modules run on the GPU (i.e., the pre- and post-processing kernels). The reason is that the latter perform virtually no floating-point multiply or add operations.

6.2.4.1. Accuracy

We benchmark the accuracy and correctness of the OpenCL implementations using both the M_4' and the linear interpolation functions on the above-described test case on the NVIDIA Tesla C2050 GPGPU. Figure 6.14 shows the convergence plots for 2D (blue) and 3D (red). When using single precision, the implementations do not converge below machine precision of 10^{-6} (Fig. 6.14a and b). The convergence plots also show the expected third-order convergence when using the M_4' scheme and second-order con-

vergence for the linear interpolation scheme, in both the ℓ^2 and the ℓ^∞ norms of the error. In all cases, the errors on the CPU (gray squares) and on the GPU (crosses) are identical, demonstrating the correctness of the GPU implementations.

6.2.4.2. Runtime

We compare the runtimes of the OpenCL implementations on the NVIDIA and ATI GPUs with the baselines provided by the sequential and multi-threaded OpenMP codes in the PPM library run on the 8-core AMD CPU. For the OpenCL implementations we also take into account the time needed to sort the particles into the data structures as outlined in Sec. 6.2.2.2, and to transfer the data to and from the device memory. Running both implementations for various problem sizes (i.e., numbers of particles and mesh nodes), we check how the implementations scale with problem size and how the speedups evolve. All OpenMP benchmarks are performed using 8 threads distributed over the 8 cores of the benchmark CPU. The speedups (OpenCL GPU vs. multi-threaded CPU) and sustained performances (interpolation kernels only) are measured in all cases. The values obtained for the NVIDIA Tesla are summarized in Table 6.2, those for the ATI Cayman in Table 6.3.

Figure 6.15 shows the wall-clock times for particle-to-mesh interpolation on all four benchmark platforms. Bars for single-precision arithmetics point downward, for double-precision arithmetics upward. The numbers above the bars give the speedups with respect to the single-thread CPU implementation when taking communication overhead into account. The speedup of the GPU over the CPU grows with increasing problem size. The speedups when using single-precision arithmetics are about twice larger that when using double-precision arithmetics. This is due to the fact that the GPUs used in our benchmarks require several clock cycles per double-precision instruction. On the CPU, however, single and double precision operations require approximately the same amount of time. The speedup increases with dimensionality of the space and with order (FLOP/byte) of the interpolation scheme, which can be ascribed to the increasing number of operations per particle. While the speedup of the GPU over the sequential CPU code, excluding transfer times, reaches up to 23, the speedups over

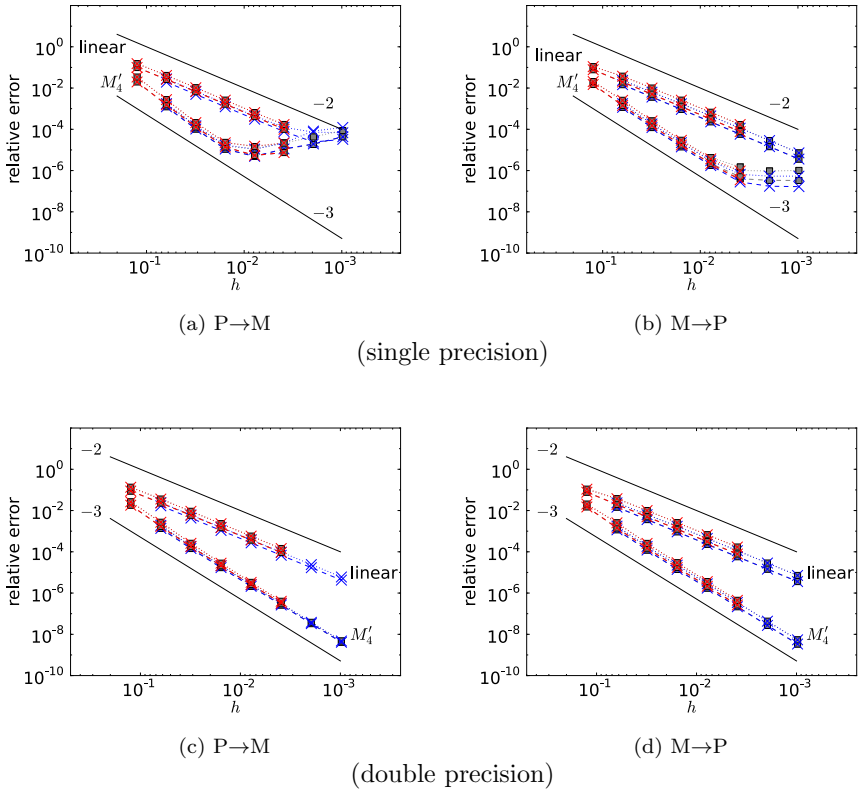


Figure 6.14. Convergence plots for M_4' and linear interpolation in single precision (a/b) and double precision (c/d). We plot the ℓ^2 (dashed lines) and ℓ^∞ (dotted lines) norms of the relative interpolation errors for both the GPU (crosses) and CPU (gray squares) implementations in 2D (blue) and 3D (red). The solid black lines indicate the slopes of orders 2 and 3, respectively.

type	ϕ	dim.	prec.	#part.	speedup w/o comm	speedup w comm	GFLOP/s
$P \to M$	linear	2D	32 bit	4096k	3.0	0.6	124.2 (12.1%)
			64 bit	4096k	1.8	0.3	
		3D	32 bit	2048k	2.3	0.6	112.6 (10.9%)
			64 bit	2048k	1.8	0.4	
	M_4'	2D	32 bit	4096k	4.3	1.7	180.8 (17.6%)
			64 bit	4096k	2.9	0.6	
		3D	32 bit	2048k	4.8	1.7	432.6 (42.0%)
			64 bit	2048k	3.1	1.0	
$M \to P$	linear	2D	32 bit	4096k	1.8	0.4	64.3 (6.2%)
			64 bit	4096k	1.8	0.3	
		3D	32 bit	2048k	1.9	0.6	199.9 (19.4%)
			64 bit	2048k	1.5	0.4	
	M_4'	2D	32 bit	4096k	4.8	1.3	273.1 (26.5%)
			64 bit	4096k	3.6	0.8	
		3D	32 bit	2048k	13.0	5.1	648.1 (62.9%)
			64 bit	2048k	8.5	2.8	

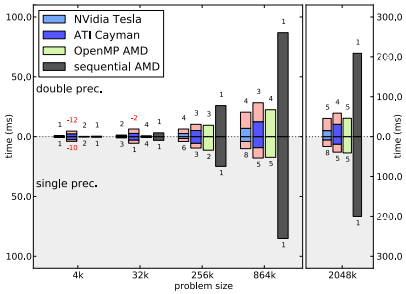
Table 6.2. Summary of the overall speedups (with and without the time required for data transfer to the device memory) measured for the OpenCL implementation on the **NVIDIA Tesla C2050** over the OpenMP reference implementation in the PPM library on the 8-core AMD FX 8150 CPU. The last column reports the sustained GFLOP/s rate for the interpolation kernels alone, i.e. Algorithms 6.3 and 6.5, and in parentheses the efficiency as the fraction of the theoretical peak performance (1030 GFLOP/s) sustained. Only the efficiency results for single precision are shown. For the double-precision kernels the GFLOP/s rates are half of those for the single-precision kernels, but the efficiencies remain the same (double-precision peak performance is 515 GFLOP/s). We only show the largest problems tested. Notice that speedups mentioned in the main text may be for other problem sizes.

type	ϕ	dim.	prec.	#part.	speedup w/o comm	speedup w comm	GFLOP/s
$P \to M$	linear	2D	32 bit	4096k	0.7	0.3	95.9 (3.5%)
			64 bit	4096k	0.5	0.2	
		3D	32 bit	2048k	0.5	0.3	83.8 (3.1%)
			64 bit	2048k	0.5	0.2	
	M'_4	2D	32 bit	4096k	1.7	0.8	154.1 (5.7%)
			64 bit	4096k	1.3	0.5	
		3D	32 bit	2048k	2.1	1.1	184.2 (6.8%)
			64 bit	2048k	1.5	0.8	
$M \to P$	linear	2D	32 bit	4096k	0.4	0.2	57.0 (2.1%)
			64 bit	4096k	0.4	0.2	
		3D	32 bit	2048k	0.4	0.2	242.8 (9.0%)
			64 bit	2048k	0.4	0.2	
	M'_4	2D	32 bit	4096k	2.0	0.8	414.5 (15.3%)
			64 bit	4096k	1.6	0.7	
		3D	32 bit	2048k	5.7	2.8	889.0 (32.9%)
			64 bit	2048k	3.9	1.8	

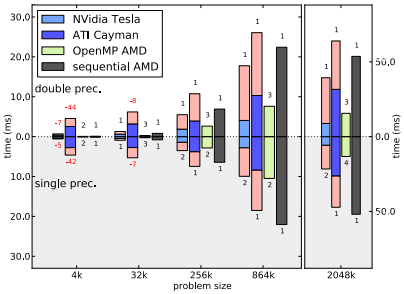
Table 6.3. Summary of the overall speedups (with and without the time required for data transfer to the device memory) measured for the OpenCL implementation on the **ATI Cayman Radeon HD 6970** over the OpenMP reference implementation in the PPM library on the 8-core AMD FX 8150 CPU. We only show the largest problems tested. Notice that speedups mentioned in the main text may be for other problem sizes.

the OpenMP-parallel CPU code show mixed results. The largest speedup over the parallel CPU code is observed when using M_4' interpolation in 3D with single-precision arithmetics and 256k particles. In this case, the NVIDIA Tesla C2050 solves the problem up to 7 times faster than the CPU employing all 8 cores. This speedup reduces to 3-fold when also accounting for the data-transfer time. The OpenCL code on the ATI Cayman only outperformed the parallel CPU code when using M_4' interpolation. In all other cases, the 8-thread CPU code was faster. In the worst case (linear interpolation, 2D, double precision), both GPUs perform significantly worse than the 8-core CPU. This suggests that the most efficient implementation should be chosen based on the order of the interpolation scheme (support size and compute intensity), problem dimensionality, and problem size. The efficiency on the NVIDIA Tesla ranges from 10% for linear interpolation (equal for both 2D and 3D, 2D data not shown) to 42% for M_4' interpolation in 3D. As expected, computationally more intense kernels (3D and M_4') lead to higher efficiencies. Comparing the sustained performance on the GPU to the sustained performance on the CPU reveals the overhead imparted by building the auxiliary data structures and sorting the particles.

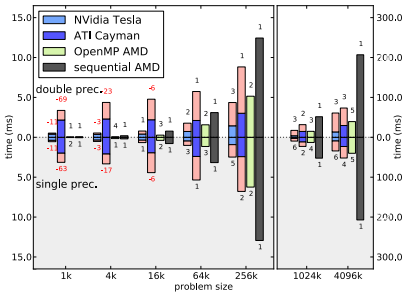
Figure 6.17 shows the results for mesh-to-particle interpolation. On the GPU, mesh-to-particle interpolation is faster than particle-to-mesh interpolation, since the former does not use any synchronization barriers and also has a higher compute intensity (FLOP/byte). This is also reflected in the generally higher efficiencies in this case. As in the particle-to-mesh case, the speedups become smaller with decreasing dimensionality, decreasing problem size, increasing numerical precision, and decreasing order of accuracy of the interpolation scheme. The largest observed speedup over the parallel CPU code is 15-fold (7-fold when accounting for data-transfer time) when using M_4' interpolation in 3D with single precision on the NVIDIA Tesla. On the same device, the speedups with respect to the sequential CPU code are 57-fold and 22-fold without and with data-transfer time, respectively. A breakdown of the computational time by kernels for this case is shown in Figure 6.16(a). The actual interpolation (Algorithm 6.5) only amounts to 14% of the the total time; the rest is consumed by pre- and post-processing kernels and by data transfer. The case that delivered the smallest speedup is shown in Fiugre 6.16(b). There, the actual interpolation kernel only accounts for 8% of the computer time, and 80% of the time are used for data transfer.



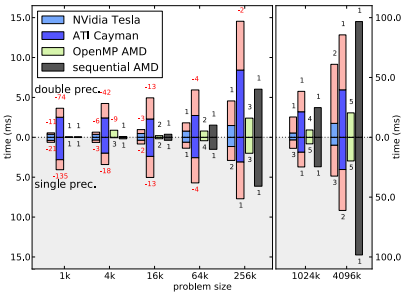
(a) 3D M_4' interpolation



(b) 3D linear interpolation

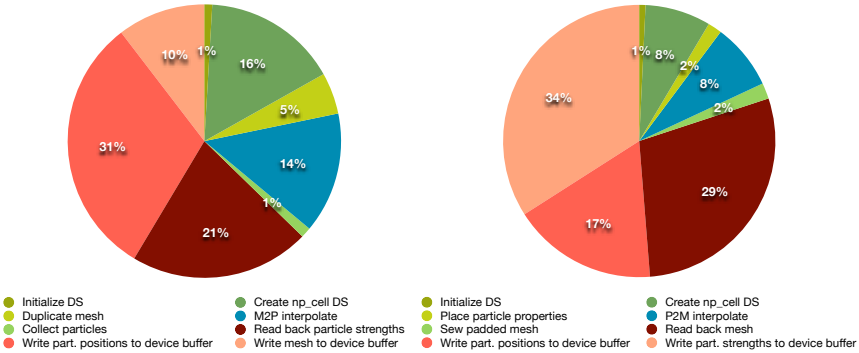


(c) 2D M_4' interpolation



(d) 2D linear interpolation

Figure 6.15. Timings for **particle-to-mesh** interpolation. We compare the wall-clock times of the OpenCL implementation run on the NVIDIA Tesla GPU (light-blue bars) and on the ATI Cayman GPU (dark-blue bars) with those of the Fortran90-OpenMP implementation running on an 8-core AMD FX 8150 CPU at 4.2 GHz (8-thread: green bars, sequential: gray bars). The times needed to transfer the data to and from the GPU device memory are added in red. Bars pointing downwards refer to single-precision runs, bars pointing upwards to double-precision runs. The speedups with respect to the sequential CPU implementation are given above/below the bars. Speedups $1/x < 1$ are given as $-x$ in red. Notice the different time axes for the two parts of each plot. The CPU implementation does not make use of the processor's vector units and therefor has similar runtimes for single-precision and double-precision runs. A 2D CPU implementation for linear particle-to-mesh interpolation is not available in the PPM library.



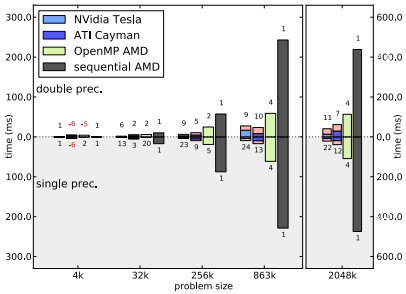
(a) 3D mesh-to-particle M_4', single prec. (b) 2D, particle-to-mesh linear, double prec.

Figure 6.16. Breakdown of the computational time on the NVIDIA Tesla C2050 GPU by kernel. We show the case (a) that delivered the largest speedup, i.e., 3D mesh-to-particle interpolation using the M_4' scheme in single precision and (b) the case that delivered the smallest speedup, i.e., 2D particle-to-mesh interpolation using the linear scheme in double precision.

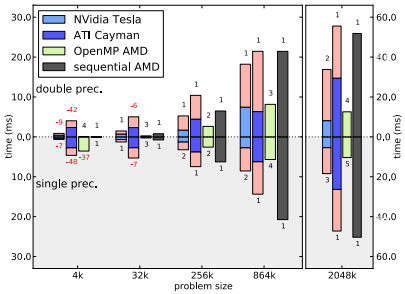
The ATI Cayman only marginally outperforms the 8-core CPU when using M_4' interpolation. Its best case, M_4' interpolation in 3D with single precision, shows a 6-fold speedup (4-fold when account for data-transfer time). In the worst case (linear interpolation, 2D, double precision) both GPUs perform significantly worse than the 8-core CPU (the NVIDIA Tesla is 3.2 times slower than the 8-thread CPU, the ATI Cayman 5.8 times). The efficiency again grows for more compute-intense kernels and reaches 63% for M_4' interpolation in 3D on the NVIDIA Tesla. The fact that the speedup in the same case is not very large can again be attributed to the overhead of building the auxiliary data structures and reducing the results. The linear kernels are memory-limited, and in 3D 70% of the theoretical peak memory bandwidth are sustainably utilized.

6.2.5. Conclusions and Discussion

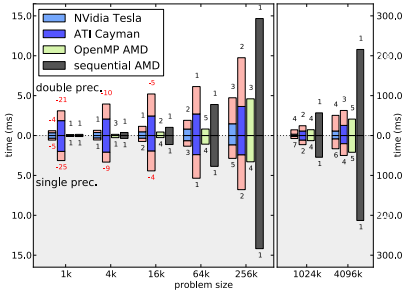
We have presented a portable OpenCL implementation of generic algorithms for SIMD-parallel particle-to-mesh and mesh-to-particle interpola-



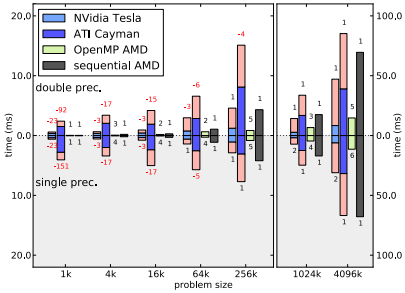
(a) 3D M_4' interpolation



(b) 3D linear interpolation



(c) 2D M_4' interpolation



(d) 2D linear interpolation

Figure 6.17. Timings for **mesh-to-particle** interpolation. We compare the wall-clock times of the OpenCL implementation run on the NVIDIA Tesla GPU (light-blue bars) and on the ATI Cayman GPU (dark-blue bars) with those of the Fortran90-OpenMP implementation running on an 8-core AMD FX 8150 CPU at 4.2 GHz (8-thread: green bars, sequential: gray bars). The times needed to transfer the data to and from the GPU device memory are added in red. Bars pointing downwards refer to single-precision runs, bars pointing upwards to double-precision runs. The speedups with respect to the sequential CPU implementation are given above/below the bars. Speedups $1/x < 1$ are given as $-x$ in red. Notice the different time axes for the two parts of each plot. The CPU implementation does not make use of the processor's vector units and therefor has similar runtimes for single-precision and double-precision runs.

tion on GPUs. The same parallelization strategy has been used for both mesh-to-particle and particle-to-mesh interpolation, and it also readily generalized to 3D. Our algorithm is generic with respect to the mesh size, the input representation, the number of particles per mesh cell, and the number of particle properties that are to be interpolated. Large inhomogeneities in the particle density, however, lead to load imbalance and lower performance. In these cases, Adaptive Mesh Refinement (AMR) strategies are available to restore a more uniform particle distribution across mesh cells [Bergdorf et al., 2005]. AMR codes have been parallelized on GPUs with about 10-fold speedup [Schive et al., 2010].

The present parallelization strategy uses threads (work items) over mesh cells and a particle-push, mesh-pull paradigm [Stantchev et al., 2008]. Particle data are reordered according to the memory access patterns of the work items in a workgroup. Additionally distributing multiple particles in a mesh cell across domain copies, and decomposing domain copies into blocks, leads to efficient use of the global memory bandwidth. Particle reordering also allows us to benefit from local memory and high cache hit rates. We avoid race conditions in particle-mesh interpolation by workgroup barriers and by replicating mesh nodes in ghost layers between neighboring blocks (workgroups).

We benchmarked both the accuracy and the computational cost of the presented OpenCL implementation on two GPU platforms against the existing, highly optimized single-thread CPU implementation in the PPM library and a shared-memory-parallel OpenMP version thereof. The benchmarks have shown that speedups of up to 22-fold (57-fold when not accounting for data transfer) over the sequential CPU code are possible with a generic OpenCL algorithm running on a GPU. This, however, depends on using an interpolation kernel with a high computation-to-memory ratio and rapidly diminishes when using double-precision arithmetics (due to the emulated double-precision support on the GPUs used). Also, the speedups observed for the present general-purpose implementation are below those reported for specialized codes in 2D [Rossinelli et al., 2010, 2011]. This is mainly due to the time needed in the present implementation to sort the particle data into GPU-suited data structures. Without these pre- and post-processing kernels, i.e., if the calling program would already store the particle and mesh data in a GPU-friendly ordering, higher overall speedups could be re-

alized, as can be extrapolated from the pie charts in Figure 6.16. Rossinelli et al. reported a 35-fold speedup of their OpenCL implementation over their single-thread CPU implementation of mesh-to-particle interpolation using the M_4' function in 2D with double precision [Rossinelli et al., 2011]. For the same case, the present OpenCL implementation shows a 19-fold speedup over the single-thread CPU implementation of the PPM library. Rossinelli et al.'s OpenGL implementation of 2D particle-mesh interpolation displayed a 26-fold speedup [Rossinelli and Koumoutsakos, 2008]. In single precision, Rossinelli et al.'s mesh-to-particle interpolation showed a 145-fold speedup [Rossinelli et al., 2011], which is significantly larger than the 24-fold speedup of the present implementation in the same case.

From our efficiency measurements we also see that more compute-intense kernels with a larger number of floating-point operations per load-store operation allow higher efficiencies (up to 63% for M_4' interpolation in 3D). Hence, 3D interpolation kernels use the hardware more efficiently than 2D ones, and M_4' kernels have a higher efficiency than linear ones. The linear kernels were memory-limited in all cases.

Our results also confirm previous reports that linear interpolation schemes achieve smaller speedups on a GPU than higher-order schemes [Madduri et al., 2011, Rossinelli et al., 2011]. We believe that this is due to the lower FLOP/byte ratio of the linear kernels. The present results also show that on the hardware used, double-precision computations take about twice longer than single-precision ones. When using higher-order interpolation schemes, such as M_4', this could be an issue since the interpolation error quickly drops below the single-precision machine epsilon. However, if solution accuracies around 10^{-5} to 10^{-6} are sufficient, single-precision arithmetic provides better speedups, especially for higher-order interpolation schemes with more compute-intense kernels. With the advent of APUs and better double-precision support on GPUs, however, this could be a temporary limitation.

Compared with the multi-threaded OpenMP reference implementation running on an 8-core CPU, the OpenCL code was in most cases slower (always slower when using linear interpolation). This is in agreement with previous reports [Madduri et al., 2011, Rossinelli et al., 2011]. For particle-to-mesh interpolation, the OpenMP implementation does not incur any thread synchronization and hence does not require atomic operations. This leads to

an almost perfect scaling of the OpenMP-Fortran90 code with the number of CPU cores. When taking data transfer times into account, significant speedups over the 8-core CPU could only be achieved for 3D mesh-to-particle interpolation using the M'_4 scheme in single precision.

Despite the modest speedups of the GPU implementation, outsourcing interpolation to the GPU may free CPU resources to meanwhile perform other tasks. Furthermore, GPUs may provide interesting options for real-time *in-situ* visualization of running simulations [Mei et al., 2007, Fraedrich et al., 2010, Goswami et al., 2010]. A bit more speed could probably be gained on the NVIDIA GPU when using CUDA instead of OpenCL [Du et al., 2012, Rossinelli et al., 2011], albeit at the expense of reduced portability. Aiming at a generic, portable code that could be integrated into the general-purpose PPM library, we chose not to do so.

OpenCL is a portable parallel computing language, and OpenCL code can also run on multi-core CPUs. For the present implementation, however, running the OpenCL code on the 8-core AMD CPU was slower than using the OpenMP code on the same CPU. We hence chose the OpenMP code as the baseline for the benchmarks presented here.

The presented particle-to-mesh and mesh-to-particle interpolations extend the PPM library with OpenMP parallelism and transparent GPU support on distributed-memory parallel computers. The presented algorithms are applied locally per sub-domain (i.e., per processor) of a domain decomposition. They thus have no influence on the network communication overhead of a distributed parallel simulation, assuming that the ghost layers are populated beforehand. If several nodes of a compute cluster are equipped with GPUs, this enables distributed-memory multi-GPU interpolation, offering a simple hybrid MPI-OpenCL platform for large particle-mesh simulations.

Part III.

A domain-specific language for particle methods

Abstraction

CHAPTER 7

The Parallel Particle Mesh Language

[1]The abundant proliferation of computing power has helped establishing computer simulations as an integral part of research and engineering. In recent years, computational tools have become important instruments in virtually all sciences. At the same time, hardware platforms in general and parallel high-performance computers in particular have become increasingly sophisticated, featuring multiple levels of parallelism and memory. Optimal use of such hardware requires in-depth knowledge of the different technologies.

Many approaches have been taken since the early days of computing to render hardware 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

[1]This work has been done in collaboration with Milan Mitrović, who helped design and implement the software.

library. An alternative approach, however, is to provide a domain-specific language (DSL).

Particle methods have a broad spectrum of applications. In discrete models particles can be used to represent the individual actors of a system (e.g., atoms, cars, or cells). Continuous systems can be discretized using particles that represent Lagrangian tracer points and evolve through their pairwise interactions.

We present a domain-specific language that allows the programmer to effectively write hybrid particle-mesh-based simulations. This language is derived from the parallel particle-mesh abstractions described in Sbalzarini [2010] and uses the PPM library's abstract data types introduced in chapter 3.

7.1. Domain-Specific Languages

Domain-specific languages offer unique opportunities for hiding the increased domain and hardware complexity while offering the programmer (or user) a simple and expressive interface. Although DSLs have only become popular in recent years, and are now found in many applications, they have a long-standing history. For example, BNF [Backus, 1959], a formalism for describing the syntax of programming languages, document formats, communication protocols, and much more, can be considered one of the oldest and most widely used DSLs. DSLs differ from general-purpose languages in several aspects:

- Domain-specific languages are typically smaller than general-purpose languages and do not offer the full capabilities of languages like Fortran, C++, or Java. One way around this possible limitation is to embed DSL code into its target language. Such languages are generally referred to as embedded DSLs (eDSL or sometimes DSEL). An example of an eDSL is OpenMP. OpenMP implementations use compiler directives in the source code to generate the necessary instructions and function calls to produce a thread-parallel version of the source code.

- Domain-specific notations and formulations are usually not found in general-purpose languages, but they can be part of the requirements for a DSL, since its purpose serves a particular domain. Computer algebra systems, such as SAGE [Stein and Joyner, 2005], often provide the user with a DSL including mathematical symbols and notations. With this in mind, a DSL does not necessarily have to be implemented as a language, but its functionality may also be provided through an application library.

- Domain-specific data structures and algorithms can directly be expressed in a DSL through its API (application programming interface), keywords, or notations.

- A DSL provides additional knowledge that can be used for code analysis, verification, semantic checks, and optimizations that would be harder or impossible to achieve using an application library, such as PPM [Mernik et al., 2005, Andova et al., 2011, DeVito et al., 2011].

An important aspect of any abstraction is the granularity at which it is are defined. On the one hand, the message passing interface (MPI), OpenMP, OpenCL, and Linda [Carriero and Gelernter, 1989] are examples of fine-grained abstractions. Even though they provide an abstract and portable interface to parallel systems through their API (MPI) or a DSL (OpenMP), they still give the programmer full control over the parallel execution of an application. On the other hand, libraries such as FFTW, PETSc [Balay et al., 2004], and PARTI [Moon and Saltz, 1994] offer coarse-grained abstractions to numerical methods, internally handling most or all implementation details (including parallelism) of these methods. DOLFIN (FEniCS) [Logg, 2007, Logg et al., 2010, Logg and Wells, 2010] offers a high-level abstraction layer for solving PDEs using the Finite Element Method (FEM). In particular, it offers a DSL implemented as a Python API for directly expressing the governing equations of a model and solving them using FEM. It follows a multi-layered approach in which the executed Python code dynamically generates C++ relying on a lower-level library for the numerical solvers. PELLPACK [Houstis et al., 1998] is a problem-solving environment [Houstis et al., 1997] for mesh-based PDE solvers. It is another example of a software stack offering several layers of abstraction from numerical solver routines up to a graphical user interface for specifying

the problem and analyzing the solution. Uintah [de St. Germain et al., 2000] is a massivly-parallel problem-solving environment that offers a high-level visual programming interface for PDE solvers. Uintah's design is in part facilitated by the common component architecture (CCA) [McInnes et al., 2006].

Another recent example of abstraction languages for parallel high performance computing applications is provided by DeVito et al. [2011]. Liszt is a DSL based on the Scala programming language that allows building portable mesh-based PDE solvers. The language provides high level types and statements that are parsed and compiled into C++ or CUDA code, employing pthreads for shared-memory parallelism and MPI for distributed memory parallelism.

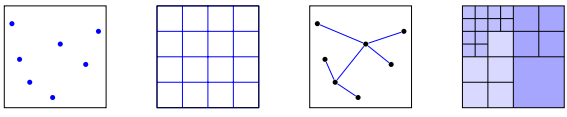
Here we present a domain-specific language for hybrid particle-mesh methods. This simplifies the development of PPM clients by introducing domain-specific data types and operations for particle-mesh methods, along with language elements that provide a concise notation for specifying PPM clients.

The present approach is similar to Liszt or FEniCS in that it offers a high-level domain-specific language that is compiled to standard Fortran code linking against the PPM library.

7.2. PPM abstractions

The PPM core library is a middleware providing a number of abstractions that allow the programmer to write 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. Specifically, the PPM library provides the following abstractions, which can be divided into three groups:

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



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

Figure 7.1. PPM provides four types of data abstractions: particles (a), meshes (b), connections (c), and toplogies (d). Hybrid particle-mesh simulations can be expressed in terms of particles and meshes. Connections allow particle-particle associations, representing for example covalent bonds in molecular dynamics, contact lists in discrete element methods, or graph edges in social networks.

pointers.

- *Particles* and *mesh* are abstractions that represent the corresponding discretization elements as defined in chapter 1. Particles are defined by their positions and properties, while a mesh is defined by its resolution, offset and a set of patch origins and extents. Mesh patches allow for local mesh refinements as used in adaptive mesh refinement (AMR) methods. One simulation may contain several sets of particles and meshes that interact with each other (Fig. 7.1 (a) and (b)).

- *Connections* (Fig. 7.1 (c)) are particle-particle associations that can be used to model e.g., unstructured grids, graphs, or chemical bonds between atoms represented by particles.

- A *topology* is defined by a domain decomposition and a subdomain-to-processor assignment (Figs. 7.1 (d) and 7.2, upper-right panel). During a simulation, one or several topologies can be created (or destroyed), and each particle set, mesh, and connection set is mapped to at most one topology at a time. The choice of decomposition largely depends on the compute operations to be executed on the data abstractions and the distribution of the particles, mesh patches and connections. For example, fast Fourier transforms are most efficiently executed on slab decompositions (c.f. section 2.1.1) due to the structure of this computation.

2. **Communication abstractions** provide transparent inter-process

communication for the data abstractions. They make explicit the incurred communication overhead, while hiding the intricacies of programming a parallel application. Making the communication overhead explicit to the programmer, helps assessing the expected parallel scalability.

- In order to associate a particle set or mesh with a topology, a *global mapping* (Fig. 7.2, lower-left panel) operation has to be executed. This abstraction encapsulates the necessary communication to move the particles, mesh patches, and connections to their target processor as defined by the respective topology.

- When particle positions are updated, moving them no more than the immediate neighborhood of the current processor they are assigned to, *local mappings* can be used to migrate the particle data to their new target processor. This mapping abstraction incurs a smaller overhead than global mappings, since it can be implemented using only local or shared-memory communication.

- In order to locally evaluate interactions between particles or mesh nodes, PPM allocates ghost (halo) layers around processor boundaries. These ghost layers are populated with ghosts (copies) of the particles/mesh nodes from the neighboring processors. *Ghost mappings* (Fig. 7.2, lower-right panel) populate these ghost layers and allow the user to send back contributions from the ghosts to their real counterparts.

3. **Compute abstractions** are abstractions encapsulate computations performed on the data abstractions. For particle-particle interactions they internally use cell lists [Hockney and Eastwood, 1988], Verlet lists [Verlet, 1967], or adaptive-resolution neighbor lists (c.f. chapter 4) to efficiently find interaction partners. The PPM numerics library provides a number of high-level abstractions encapsulating frequently used numerical solvers, including multigrid and FFT-based Poisson solvers.

All abstractions have been implemented as Fortran 2003 derived types and subroutines, as described in chapters 2 and 3. They are accessible through the library's API and offer the programmer an abstraction layer

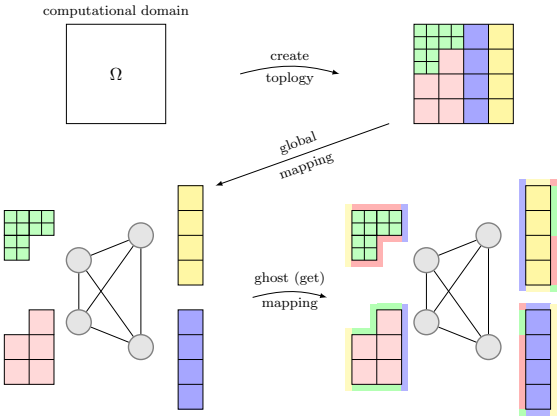


Figure 7.2. Using the topology abstraction, the computational domain Ω is decomposed and the sub-domains assigned to processors (upper panels). The global mapping abstraction distributes particles, meshes, and connections across the processors (represented by the graph) as specified by to the topology (lower-left panel). The ghost get mapping abstraction creates ghost layers around subdomains according to the topology and the boundary conditions (in the example shown here, periodic boundary conditions).

of intermediate granularity.

7.3. PPML syntax and features

Owing to the names and calling conventions used, every library's API can be interpreted as a DSL, albeit without the expressiveness of a language [Mernik et al., 2005]. PPML is an eDSL that is embedded into Fortran code and supplements it with types and operations needed to compactly express parallel hybrid particle-mesh simulations. PPML's abstraction types are directly derived from the PPM abstractions described by Sbalzarini [2010]. Besides particles, meshes, and connections, PPML also provides following types:

- A *field* is an abstract type that represents a *continuous* physical quantity 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 the model to be simulated directly within PPML. These types can also be used to provide the user with contextual feedback during execution, and with annotated outputs
- *Neighbor lists* can either be traditional cell or Verlet lists, or adaptive-resolution neighbor lists. Furthermore, they can be built as cross-set neighbor lists, allowing the programmer to associate two sets of particles through a neighborhood relationship.
- *Operator discretizations* are defined by an operator and a particle or mesh discretization. They amount to implementations of numerical methods for the respective abstract operator.
- The *ODE* type encapsulates the functionality needed for solving ordinary differential equations using time integration.

In addition to types and operators, PPML also supports `foreach` control flow structures (Table 7.1) for intuitive access to discretization elements. `foreach` loops are highly customizable iterator templates. PPML provides `foreach` loops for particles, particle neighbors, and meshes, but new iterators can easily be added using PPML macros (c.f. 7.4.3). The loops allow iterating through the discretization elements and accessing their properties and positions. Furthermore, the loops provide special clauses for accessing only parts of a mesh (i.e., its bulk or its halo layers) or only a subset of particles (i.e., only real particles or only ghost particles). This is particularly useful when writing operator stencils. Naturally, `foreach` loops can also be nested and composed inside macros, providing great flexibility. Finally, PPML extends Fortran's array index operator `()` with an array index offset operator `[]`. This operator simplifies writing finite-difference stencils on meshes.

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.

Extending Fortran 2003 with PPM-specific language elements has the advantage that many of the basic syntactic and semantic features of a programming language are already provided by Fortran itself and can be reused at no additional cost. It also allows us to focus on developing the tools required for processing PPML's extensions to Fortran, while relying on the wealth of compilers and development tools that already exist.

7.4. Implementation

In order to define and process PPML code, we build a flexible source-to-source compiler framework allowing us to embed arbitrary code into Fortran 2003 source code. The framework is composed of three parts. The *parser*, the *macro collection,* and the *generator*. The *parser* reads PPML/Fortran code and generates an intermediate representation generally known as abstract syntax tree (AST). The parser largely ignores Fortran, but is aware of scope declarations, such as modules, subroutines, functions, and derived types. It also keeps track of the scope hierarchy and various

Control flow structure

```
foreach p in particles(pset) with fields(f) ! [options]
   ! iterator body
   f_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
      ! ...
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 7.1. Examples of PPML control-flow structures.

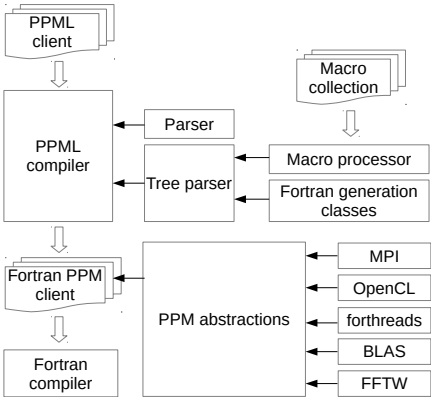


Figure 7.3. The PPML framework. Line arrows represent "used by" relationships, hollow arrows show the processing order. The PPML compiler is supported by the parser and the tree parser, which in turn uses the PPML macro collection and the Fortran generation classes to expand PPML operations to Fortran source code.

elements declared within the current scope. Using this data structure it is possible to provide simple implicit type declarations for PPML variables. 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 operation or language element is encountered, the pre-processor looks up the appropriate macro and injects the resulting Fortran code into the target code. The basic operation and component structure of the PPML framework is shown in Fig. 7.3. The PPML framework is implemented using the Ruby programming language, which provides a number of libraries and tools for parsing, code generation, and command-line handling. Furthermore, Ruby provides the eRuby templating framework, a useful tool for the macro collection component of PPML. eRuby is Ruby's built-in templating language that allows regular text files to be augmented with Ruby code that can be evaluated by the Ruby interpreter and replaced by its result (similar to how PPML code is evaluated to Fortran code).

7.4.1. Parsing

We build the PPML parser using the ANTLR parser generator. This parser and lexer generator allows us to define PPML's syntax and grammar using an extended BNF-like language (c.f. Appendix A). ANTLR uses this grammar to produce the necessary Ruby code for lexing a source file, parsing it, and generating an intermediate abstract syntax tree data structure. The PPML grammar consists of a number of terminal symbols, lexer rules, and higher-level parser rules. The lexer symbols and rules accept all valid tokens in Fortran and PPML. Since PPML is an embedded language, it is not necessary to create a complete parser for Fortran 2003. Instead, it is sufficient 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. Hence, large parts of the input source code are inserted verbatim into the final, generated code.

7.4.2. Code generation

A second ANTLR tree grammar is used to validate and traverse the generated abstract syntax trees from the preceding parsing step. The tree parser traverses the AST during the code generation stage, generating Fortran code at each tree node. The grammar leaves Fortran code unmodified and emits it without further processing. The syntax trees of PPML expressions, however, are traversed using the tree grammar. Each grammar rule declares an output template that takes the parsed source code as an argument and returns Fortran code. Whenever a PPML operation or control-flow structure is encountered, its name is looked up in the macro collection, the arguments and options are passed to the macro, and the macro output is inserted into in the generated code. Macro code may recursively call the PPML parser and generator, as it may itself contain PPML statements.

7.4.3. PPML Macro collection

PPML macros are eRuby Fortran templates that are evaluated at code-generation time. eRuby is a templating system that allows embedding Ruby

```
1  macro create_mesh(topology,offset)
   % scope.var({result.to_sym => "type(ppm_mesh),pointer::#{result}"},
   %           :ppm_mesh)

   allocate(<%= result %>,stat=info)
6  or_fail_alloc("Could not allocate <%= result %>")

   call <%= result %>%create(<%= topology %>,<%= offset %>,info)

   end macro
```

Listing 7.1 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.

code into text documents that are expanded to clear text at execution-time. The macro is first processed by Ruby in the scope of the current parser instance, interpreting eRuby macro elements. It is then parsed using a child PPML parser instance and macro calls within the macro are processed recursively. This allows flexible composition of macros. Listing 7.1 shows an example PPML macro. The `scope` variable is provided in all macros and allows the programmer to access information about the Fortran code currently being processed. During the parsing and generation phases the scope (and its enclosing scopes) are tracked to allow the user to manipulate the scope by adding local variables, `use` declarations, includes and procedure interfaces in Fortran. The `result` variable refers to the left-hand side variable of the statement currently being evaluated in the Fortran code.

Most language features of PPML are implemented as macros. This includes the `foreach` iterators. We provide three generic types of macros: function macros, include macros, and `foreach` macros. Function macros are the basic building blocks for all PPML operations they are evaluated inside procedure or program scopes and allow modifying the current scope by adding new variables and `use` clauses. Include macros allow verbatim code inclusion, similar to CPP `#include` statements, and can be used almost everywhere in the user code. Lastly, `foreach` macros are used to specify iterators. They admit multiple named-argument and loop-body clauses.

Besides PPM abstractions and iterator macros, we provide a number of convenience macros for frequently used tasks. These macros range 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 in order to adapt its semantics to the programmer's needs. Furthermore, new macros can easily be defined and are imported into the macro collection whenever the PPML generator is executed.

7.5. The PPM client generator

PPML provides two language features in addition to the ones described in section 7.3: `client` and `rhs`. Both keywords are scope declarations similar to Fortran's `program` or `subroutine` scopes. The `client` scope is similar to the Fortran `program` scope, but indicates to the PPML generator that this scope encloses a PPM client. In this case, the code generator additionally produces two Fortran modules that provide initialization and utility functions, global access to parameters, and a container for user-defined procedures. The `rhs` scope is used in conjunction with PPML's `ode` type. This scope is essentially a Fortran function, but provides a convenient shorthand for programmers to specify the right-hand side of the equations to be solved. Using PPML's `ode` type and `rhs` scopes much of the code required to access to discretizations of fields that are used on the right-hand side is auto-generated in a way that is transparent to the programmer.

Using PPM and PPML as the center pieces, we build an environment for hybrid particle-mesh simulations that covers a large part of the development cycle. This environment is used via command-line utilities that offer the programmer a number of tools for the initialization, construction, and testing of PPM simulation:

- newproject: This tool generates a PPM project directory and a code scaffold allowing the programmer to work within a well-defined environment.

- build: In order to generate the final simulation binary all the PPML source files are collected, parsed, and Fortran code is generated. All output files are placed in a project sub-folder together with an auto-generated Makefile. Finally, the make utility is invoked to compile the generated sources to the target binary.

- pp: to pre-process a single PPML file into compilable Fortran sources, the `pp` tool is used. This tool is also used for pre-processing the PPM library's code itself.

- bench: Benchmarking the efficiency and scalability of a parallel simulation is frequently required during the development of a parallel application. Therefore, we provide a simple benchmarking framework for PPM simulations. Together with timing routines used inside the simulation code, this tool allows the programmer to submit a number of cluster jobs, measuring the execution time, and determining the parallel efficiency.

7.5.1. A minimal PPML example

In order to illustrate PPML and the client generator, we present two minimal examples simulating diffusion in the domain $(0,1) \times (0,1)$. Diffusion of a scalar quantity $U(\mathbf{x}, t)$ is described by

$$\frac{\partial U}{\partial t} = \Delta U. \tag{7.1}$$

Listing 7.2 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 11-20 initialize PPM, create a field, a topology, creating a mesh and discretize the field onto the mesh. Lines 21-27 specify the initial condition. In this example this is done by discretizing a Gaussian onto the mesh using the mesh-`foreach` loop. Line 29 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 31-36 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. Line 39 finalizes the client by calling the macro for PPM finalization. Lines 41-54 show a simple `rhs` code implementing the right-hand side of equation 7.1 using a mesh-`foreach` loop over all real (i.e., non-ghost) mesh nodes.

In listing 7.3 we solve the same governing equation with the same initial and boundary conditions, but using a particle discretization instead of a mesh discretization (c.f. chapter 1). In lines 32/33 we use a discretization-corrected particle strength exchange (DC-PSE) operator [Schrader et al., 2010] as a particle approximation to the Laplacian (c.f. section 1.1.2). The main differences with the mesh-based implementation are in the setup, where we create a particle set instead of a mesh (line 13), and in the implementation of the right-hand side. Since PPM already provides routines for defining and applying DC-PSE operators, we only need to call these routines. Of course, similar implementations for mesh-based operators could also be added to PPM and provided to the programmer using the PPML operator type. For comparison, we show the full auto-generated Fortran 2003 source code for this PPML client in Appendix B.

7.6. A visual programming interface for PPM

[2]In order to further reduce the knowledge barrier for writing parallel hybrid particle-mesh simulations, we build on top of the PPML language and the PPM library a web-based visual development environment. Various visual programming environments exist and are used in applications ranging from engineering to computer science education. MathWorks' Simulink, e.g., offers a block-diagram-based graphical environment for simulating multidomain dynamic systems. It has recently been extended to run on parallel platforms [Canedo et al., 2010]. Another recent example of a visual programming language is Blockly. Blockly is a simple open-source web-based editor allowing the programmer to visually combine small program elements to larger programs. It is mainly intended for educational use, but could also be used as an editor component in other visual development

[2]This work has been done together with Joël Bohnes, who helped designing the software and led the implementation of the client side.

```
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 laplaceian
  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 7.2 PPML specification of a PPM client to simulate diffusion on a Cartesian mesh.

```
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
    if (ppm_rank.eq.0) print *,st
  end timeloop

  ppm_finalize()
end client

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

Listing 7.3 PPML specification of a PPM client to simulate diffusion using particles and the DC-PSE method [Schrader et al., 2010].

environments or expert systems.

We present a web-based visual programming environment for PPML, which we call webCG. WebCG uses a client-server architecture, where the client is implemented in the web browser using various web technologies, while the server is implemented using the Ruby Sinatra web application framework. The server uses the PPM client generator to build and execute PPM clients and returns the results to the web front-end.

WebCG enables the user to model a PPML client in a web browser using a block-diagram representation. The blocks represent different PPML objects and operations. It is possible to extend the block diagram with arbitrary Fortran code-snippet blocks. The user draws connections between blocks according to the desired data flow. Before submission to the server, PPML code is generated by traversing the block diagram's graph data structure. This PPML code is then submitted to the server and processed by the PPM client generator. Upon successful completion of the build process, the user can execute the resulting PPM client on the server. The run can be monitored in real time and results are returned to the web interface. Appendix C shows an example webCG client simulating the diffusion model presented in section 7.5.1 using the DC-PSE method.

7.6.1. Architecture

7.6.1.1. Client-side architecture

The client is implemented using Javascript/ECMAScript 5, making use of a number of web technologies, such as SVG for rendering the visual programming elements, HTML/CSS for stylizing the user interface, DOM for manipulating the underlying data model, and AJAX for asynchronous communication with the application server.

User interface All user-interface elements except the block-diagram editor are implemented using the Yahoo User Interface (YUI) toolkit. The user interface consists of a project selection view, a property editor, a block selection, and a number of pop-up views for building, running, and monitor-

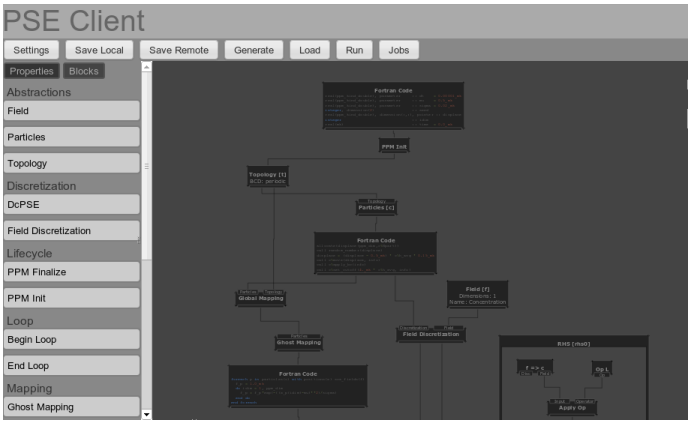
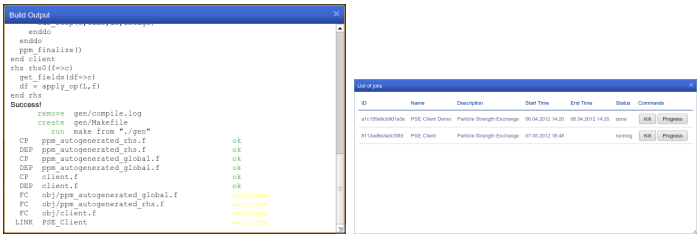


Figure 7.4. The main webCG interface. On the left, the block list and property editor view are seen. On the top, the main menu bar and the project title are placed. The interface is dominated by the SVG block-diagram editor.

ing simulations (Figs. 7.4 and 7.5). The block-diagram editor is rendered using SVG. SVG has so far predominantly been used for visualizations, and not for interactive user interfaces, only few libraries exist for manipulating SVG. We therefore extend YUI's DOM manipulation framework to also handle the SVG object model. This allows us to capture and process user interactions with the diagram editor. Furthermore, we implement a custom SVG widget-rendering engine for updating the diagram's block and wiring widgets. The wires connecting the blocks are rendered by a simple auto-routing routine so that they don't cross other widgets.

Data model The client maintains a graph data structure that is updated according to the manipulations performed on the block diagram. Each PPML type instance or operation is represented by a graph vertex that carries attributes. Attributes are set by the user through the user interface and parameterize the code generation of the associated vertex. The vertices are connected by directed edges that represent the data-flow wires. An edge $e(v_i, v_j)$ establishes a "happens before" relationship between vertices v_i and v_j. In order to generate PPML code, this graph is traversed in topological order. Whenever the order of two vettices cannot be resolved, the position



(a) **(b)**

Figure 7.5. WebCG views showing the build process of a PPML client (a) and the status of running PPM simulation jobs (b).

of the corresponding diagram blocks along the vertical axis of the canvas is used.

Server communication The web client communicates with the server via a stateless and asynchronous HTTP protocol offered through the server API. This allows the web client to remain responsive when waiting for the server to complete an operation. The data returned from the server are checked for errors, and invalid server responses are reported to the user.

7.6.1.2. Server-side architecture

The server is implemented in Ruby using the Sinatra web-application framework. PPML clients and PPM jobs are stored and handled using the server file system based storage. No additional overhead for setting up and maintaining a database is hence required. The server's HTTP API covers all expected operations from project listing, loading, and deleting, to building and executing PPM clients. Each PPM client is executed in a separate thread in order to ensure that the server remains responsive. Even though the current implementation directly executes PPM clients, it can be extended to use a cluster job scheduler.

7.7. Summary and Conclusions

PPML is a domain-specific language (DSL) for parallel hybrid particle-mesh methods: it is based on the Parallel Particle Mesh (PPM) library abstractions introduced by Sbalzarini [2010]. The language formalizes these abstractions and offers iterators for particles and meshes, a templating mechanism for Fortran, and program scopes for PPM clients. Based on the language specification we have created an implementation including a parser, code generator, and PPM client toolkit. PPML types, operations, and iterators are implemented using macros that can be modified and extended by the programmer. We have showcased PPML in two minimal client applications simulating diffusion on both a mesh and a particle discretization. Finally, we have presented a web-based visual programming environment for PPML that allows the user to quickly prototype hybrid particle-mesh simulations using a block-diagram interface. The environment is powered by a simple application server that can build and execute the generated PPML clients. Taken together, the presented DSL, source-to-source compiler, and prototyping environment offer the user an end-to-end solution for building parallel hybrid particle-mesh codes using the PPM library.

Thanks to the intermediate granularity of the PPM abstractions, PPML is able to offer a simple, yet expressive notation for particle-mesh methods that is compiled to portable distributed-memory code. Extending the PPM library to heterogeneous hardware platforms, as described in chapter 6, further improves the portability of PPML.

The PPML webCG visual programming environment builds onto the described DSL and provides a user-friendly interface for prototyping particle-mesh simulations without compromising the ability to run large-scale parallel jobs.

The presented software stack represents one possible approach to narrowing the knowledge gap [Sbalzarini, 2010]. In addition, PPML and webCG also provide opportunities for narrowing the reliability and performance gaps.

CHAPTER 8

PPML benchmark clients

[1]We showcase the PPML language and benchmark the new implementation of the PPM library using two example PPML clients. The first client is a simple but complete reaction-diffusion simulation of the Gray-Scott model [Gray and Scott, 1984]. The code is inherently parallel and can be executed on shared- and distributed-memory machines without any modifications. The second client implements a molecular dynamics simulation of a Lennard-Jones gas [Jones, 1924]. We highlight the use of PPML's `foreach` iterators and PPM's mapping abstractions. We benchmark both clients on 256 four-way AMD Opteron 8380 nodes and measure the runtime, and parallel efficiencies.

[1]Thanks to Olivier Byrde and the Brutus cluster team for technical assistance and helpful discussions.

8.1. The benchmark system

All benchmarks are performed on the ETH Brutus cluster. Brutus is a heterogeneous cluster with compute nodes of several CPU generations and architectures, some also equipped with GPGPUs. For the sake of simplicity and reproducibility, we restrict the benchmarks to one type of node. All nodes used have four AMD Opteron 8380 quad-core processors and 32 GB of main memory. The Opteron 8380 processor has a 4.0 GB/s 16-bit point-to-point link connecting each core with two others. Each core is directly connected to the main memory via two memory channels. The nodes are connected with an InfiniBand 4X QDR network and have access to a fast cluster file system. For the present benchmarks we have compiled the PPM library and the benchmark clients using the Intel Fortran compiler 13.0.0 with the `-O3` flag. We used OpenMPI 1.6.2, which has native support for InfiniBand interconnects. We were able to reserve 256 nodes (4096 cores) for exclusive access. This minimizes external effects on timing and parallel scaling. For all benchmarks we use OpenMPI's `-bysocket -bind-to-core` core affinity settings. These settings instruct MPI to bind processes to successive sockets and to individual cores. We used the `numactl` utility allowing us to control the memory allocation. By instructing `numactl` to allocate all process memory locally to the process core, we optimally utilize the processor architecture's per-core memory channels.

8.2. 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 Turing patterns [Turing, 1952] that spontaneously emerge from a homogeneous concentration distribution. In the macroscopic continuum limit reaction-diffusion systems can be described by a system of coupled partial differential equations. Each equation describes the evolution of the concentration field of one species. It's right-hand side contains a term describing the diffusion process and one or several terms describing the

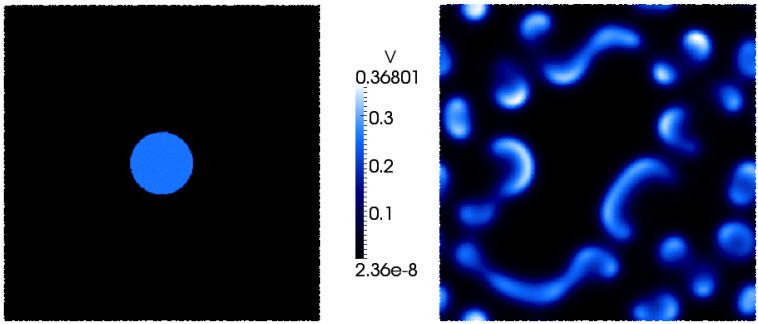


Figure 8.1. A PPML Gray-Scott reaction-diffusion particle simulation using the discretization-corrected PSE method [Schrader et al., 2010] with 10^5 particles. (a) Initial condition (b) concentration field of V after $4 \cdot 10^4$ time steps.

reactions the species undergoes.

We implement a PPML program simulating the Gray-Scott reaction-diffusion model. The Gray-Scott model produces Turing patterns in the concentration fields of the reactants U and V. The model is described by the two coupled partial differential equations

$$\begin{aligned} \frac{\partial U}{\partial t} &= D_U \nabla^2 U - UV^2 + f(1-U) \\ \frac{\partial V}{\partial t} &= D_V \nabla^2 V + UV^2 - (f+k)V, \end{aligned} \tag{8.1}$$

where D_U and D_V are the diffusion constants of species U and V, and f and k are reaction rates. We simulate reactions between the two species U and V using an ordinary differential equation, and the diffusion of U and V using discretization-corrected particle strength exchange (DC-PSE) operators [Schrader et al., 2010]. An implementation of DC-PSE is provided by the PPM operator framework (c.f. chapter 3).

Listing 8.1 shows the complete PPML implementation of the DC-PSE Gray-Scott reaction-diffusion simulation. Lines 1-10 are variable declarations and parameter setup. The `add_arg` operation provides a concise interface for command-line and configuration-file parameters. We declare runtime configuration parameters for the diffusion constants and reaction rates. The `ppm_init` operation initializes the PPM library and auxiliary libraries such as MPI. Most PPML operations may take a number of optional parameters for further customization. The programmer can, for example, pass an optional debug level argument to `ppm_init` in order to control log output. Lines 14 and 15 declare the two scalar concentration fields U and V. Lines 17-25 create the PPM topology and the particles. Since PPML is an eDSL, we can use standard Fortran code to implement the particles' off-grid displacement. The particles are then mapped onto the topology using a global mapping, and the fields are discretized onto the particles in lines 26-31. Lines 33-42 specify the initial condition of the system. Here we use a `foreach` loop to access the particle properties of U and V. We then create Verlet lists [Verlet, 1967] for the particles, and define and discretize the Laplace operator (lines 44-47). Before entering the main time loop of the simulation, the ODE module is initialized. This is done with one call, passing the fields to be used in the right-hand side of the governing equations, the particle discretization, the right-hand side function, and the time integration scheme. PPM only supports explicit time integration schemes but the ODE module is designed to be easily extensible with new schemes. Lines 50-60 implement the main time loop, including time stepping and ghost mapping. The PPML `print` operation offers a simple interface to the PPM VTK module. This call causes the simulation to output VTK files of the current fields. The right hand side of the governing equations is implemented in the `rhs` scope (lines 62-70). We first apply the diffusion operator to U and V and then iterate over the particles to update the concentrations according to the reactions. The complete client is implemented in merely 70 lines of code. This includes all necessary functionality for a parallel implementation with control file and command-line argument handling, as well as VTK output. This is made possible by PPML's abstract types and operations, iterators, and the client generator framework.

Fig. 8.1 shows the concentration field of V at time zero and after $4 \cdot 10^4$ time steps of size $\Delta t = 0.05$ for $k = 0.051$, $f = 0.015$, $D_U = 2 \cdot 10^{-5}$, $D_V = 10^{-5}$,

```
client grayscott
  integer, dimension(6) :: bcdef = ppm_param_bcdef_periodic
  integer, dimension(2) :: seed
  real(ppm_kind_double), dimension(:,:), pointer :: displace
  real(ppm_kind_double) :: noise = 0.0_mk
  integer               :: istage = 1
  add_arg(k_rate,"real(mk)",1.0_mk,0.0_mk,'k_rate','Reaction rate')
  add_arg(F,"real(mk)",1.0_mk,0.0_mk,'F_param','Reaction param F')
  add_arg(D_u,"real",1.0,0.0,'Du_param','Diffusion const U')
  add_arg(D_v,"real",1.0,0.0,'Dv_param','Diffusion const V')

  ppm_init()

  U = create_field(1, "U")
  V = create_field(1, "V")

  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)
  call c%set_cutoff(4._mk * c%h_avg, info)

  global_mapping(c, topo)

  discretize(U,c)
  discretize(V,c)
  map_ghost_get(c)

  foreach p in particles(c) with positions(x) sca_fields(U,V)
    U_p = 1.0_mk
    V_p = 0.0_mk
    if (((x_p(1) - 0.5_mk)**2 + (x_p(2) - 0.5_mk)**2) .lt. 0.01) then
      call random_number(noise)
      U_p = 0.5_mk  + 0.01_mk*noise
      call random_number(noise)
      V_p = 0.25_mk + 0.01_mk*noise
    end if
  end foreach

  create_neighlist(c)

  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,V], grayscott_rhs, [U=>c,V], rk4)
  t = timeloop()
    do istage=1,nstages
      map_ghost_get(c,psp=true)
      ode_step(o, t, time_step, istage)
    end do
    print([U=>c, V=>c],100)
  end timeloop

  ppm_finalize()
end client

rhs grayscott_rhs(U=>parts,V)
  get_fields(dU,dV)

  dU = apply_op(L, U)
  dV = apply_op(L, V)
  foreach p in particles(parts) with sca_fields(U,V,dU,dV)
    dU_p = D_u*dU_p - U_p*(V_p**2) + F*(1.0_mk-U_p)
    dV_p = D_v*dV_p + U_p*(V_p**2) - (F+k_rate)*V_p
  end foreach
end rhs
```

Listing 8.1 A complete PPML client simulating the Gray-Scott reaction-diffusion system in 2D.

#procs	4 cores per node			16 cores per node		
	DC-PSE	map ghost get	iteration	DC-PSE	map ghost get	iteration
1	0.207s	0.001s	0.229s	0.207s	0.001s	0.228s
4	0.217s	0.054s	0.240s	0.217s	0.054s	0.241s
16	0.218s	0.041s	0.434s	0.365s	0.065s	0.251s

Table 8.1. Detailed timings of the PPML reaction-diffusion client for 1, 4, and 16 processes. When using all 16 cores in a node the operator evaluation (DC-PSE) slows down by over 70%.

and 10^5 particles. As expected for this set of parameters, Turing 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 the Brutus cluster on up to 1936 processors. All timings are measured per iteration and averaged over 100 iterations. Since no global communication or barrier is included in the time loop, we report the average processor time. For a weak scaling (i.e., fixed problem size per processor) the parallel efficiency is defined as $E_p = T_1/T_p$, where T_1 is the time to execute the sequential problem and T_p is the time to execute a p-times larger problem on p processors. Fig. 8.2 shows the parallel efficiency of a weak scaling when using all cores on each node. When increasing the number of processes from 4 to 16 the efficiency drops by more than 40%. Table 8.1 shows the detailed timings for the time loop. The measurements show calculating the discretized Laplace operator is more than 70% slower when using all 16 cores of one node instead of only four cores per node. Since the DC-PSE operator does not incur any communication, we conclude that this drop in efficiency is likely due to a bottleneck in memory access. Therefore, we also test the client's parallel efficiency using at most four processes per node (allocating one core per processor). The results are summarized in Figure 8.3. In this case, the parallel efficiency only drops at 1024 processors.

Figure 8.4 shows the parallel efficiencies of fastest (blue) and slowest (red) processor. Despite the regular distribution of the particles and the regular structure of the PPM topology, we notice a significant load-imbalance. We suspect that this imbalance might stem from the computer hardware, but

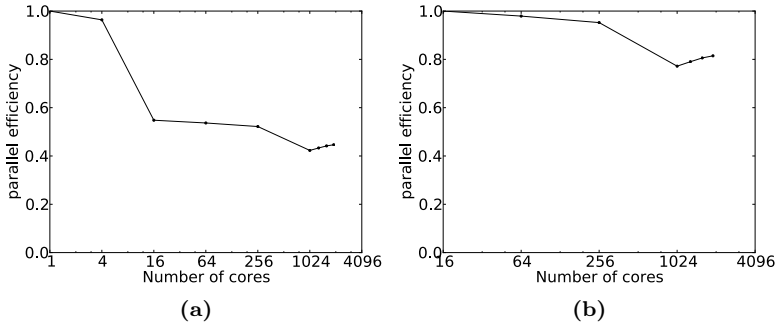


Figure 8.2. Parallel efficiency of a weak scaling of the PPML reaction-diffusion client. The largest cores of processors used was 1936. All cores per node are used. This leads to a performance drop at 16 cores (a). (b) The parallel efficiency when starting from 16 processes.

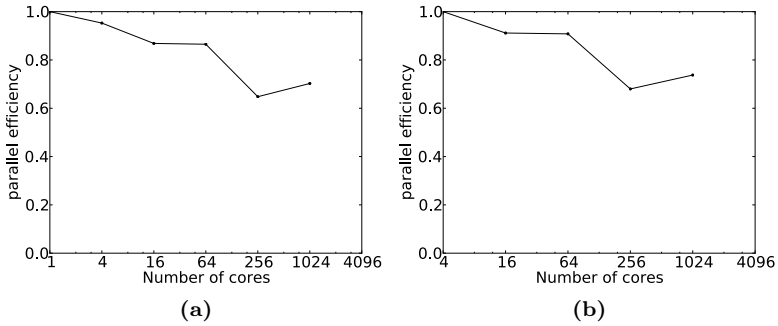


Figure 8.3. Parallel efficiency of a weak scaling of the PPML reaction-diffusion client on up to 1024 processes. In order to avoid memory congestion, we use only 1 core per processor. (a) Parallel efficiency with respect to the sequential run. (b) Parallel efficiency relative to the largest run using only one node.

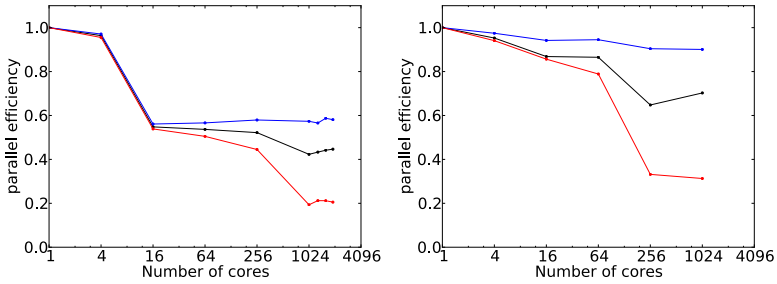


Figure 8.4. Parallel efficiency of a weak scaling of the PPML reaction-diffusion client using 16 cores per node (a) and four processors per node (b). In addition to the average efficiency, we show the efficiencies of the fastest processor in blue, and of the slowest processor in red.

further investigation may be needed.

8.3. Simulating molecular dynamics using PPML

We show the use of PPML to simulate a discrete particle system modeling the dynamics of atoms in a gas. The interactions between neutral atoms are approximated by the Lennard-Jones potential

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right], \tag{8.2}$$

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 (Fig. 8.5). We truncate the Lennard-Jones potential at $r_c = 2.5\sigma$ for computational efficiency, since not all atom pairs need to be considered anymore. The truncated Lennard-

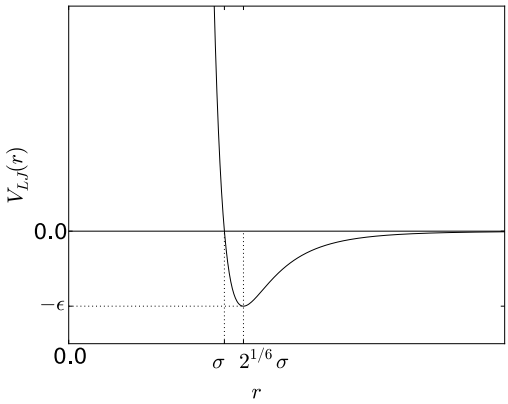


Figure 8.5. The Lennard-Jones potential function. ϵ characterizes the depth of the potential well, while σ is the distance at which the potential is zero. The potential well's minimum can be found at $2^{1/6}\sigma$.

Jones potential is formulated as:

$$V_{LJ_{trunc}}(r) = \begin{cases} V_{LJ}(r) - V_{LJ}(r_c) & , r \leq r_c \\ 0 & , r > r_c. \end{cases} \tag{8.3}$$

We use Verlet lists with a cutoff radius equal to the potential's truncation radius r_c. As the atoms are moving, the Verlet list need to be recomputed. We hence add a 10% thickness "safety-margin" (skin) to the cutoff radius when computing the Verlet list. The Verlet list then only need to be recomputed once any particle has moved more than half the skin thickness.

We use PPML to write a 3D parallel Lennard-Jones simulation. The code computes the forces, potential and kinetic energies, and motion of all atoms (particles). Time integration is done using the velocity Verlet algorithm [Verlet, 1967, Swope et al., 1982]. Listing 8.2 shows the main loop of the simulation computing the forces based on the updated particle positions. Once any particle has moved by more than half the skin thickness, the particles are remapped using a local mapping, and the Verlet list are rebuilt. This is done in lines 11-27. Lines 29-31 show nested `foreach` loops. In this case, the outer loop iterates over all particles, while the inner loop iterates

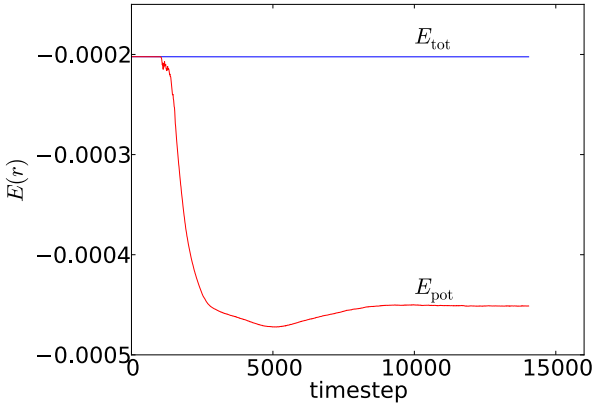


Figure 8.6. Total (blue) and potential (red) energy of 1 million particles over 14,000 time steps in a PPML Lennard-Jones simulation.

over all interaction partners of each particle.

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, kinetic, and potential energies $(E_{\text{tot}}, E_{\text{kin}}, E_{\text{pot}})$ of the system. Figure 8.6 shows the results. As expected, the total energy is conserved, while the potential energy decreases as the particles equilibrate. The potential energy stabilizes at the equilibrium of the system.

We benchmark the PPML Lennard-Jones client on the cluster described in section 8.1. Figure 8.7 shows the results of a weak scaling with 1 million particles per processor. All timings are measured per iteration and averaged over 100 iterations. We show the average time per process, but the minimum and maximum times per process were identical, indicating perfect load balance. For this benchmark, we allocate 8 MPI processes per node, using only two cores of each processor. Using all cores per node leads to loss of performance when scaling within one node, similar to what we observed for the reaction-diffusion client. We attribute this to memory bottlenecks in the force-calculation loop. The benchmarks shows that the PPML client runs on 1728 processor cores (with $1.728 \cdot 10^9$ particles) at 77.5% parallel

```
t = timeloop(tstart=0.0_mk,deltat=dt,tend=stop_time)
  maxdisp = 0.0_mk
  allmaxdisp = 0.0_mk
  foreach p in particles(parts) with positions(x,writex=true) &
  &                                  sca_props(E) vec_props(F,a,v,dx)
    a_p(:) = F_p(:)/m
    x_p(:) = x_p(:) + v_p(:)*dt + 0.5_mk*a_p(:)*dt**2
    F_p(:) = 0.0_mk
    E_p = 0.0_mk
    dx_p(:) = dx_p(:) + v_p(:)*dt + 0.5_mk*a_p(:)*dt**2
    disp = dx_p(1)**2 + dx_p(2)**2 + dx_p(3)**2
    if (disp.gt.maxdisp) maxdisp = disp
  end foreach

  pmax(madisp,allmaxdisp)

  if (4.0_mk*allmaxdisp.ge.skin**2) then
    call parts%apply_bc(info)
    partial_mapping(parts)
    foreach p in particles(parts) with vec_props(dx)
      dx_p(:) = 0.0_mk
    end foreach
    map_ghost_get(parts)
    comp_neighlist(parts)
  else
    map_ghost_get(parts,psp=true)
  end if

  foreach p in particles(parts) with positions(x) ghosts(true) &
  &                                  sca_props(E) vec_props(F)
    foreach q in neighbors(p,nlist) with positions(x)
      r_pq(:) = x_p(:) - x_q(:)
      r_s_pq2 = r_pq(1)**2 + r_pq(2)**2 + r_pq(3)**2
      if (r_s_pq2.le.cutoff**2) then
        scaldF = (24.0_mk*eps)*(2.0_mk*(sigma12/r_s_pq2**7) &
        &                          - (sigma6/r_s_pq2**4))
        F_p(:) = F_p(:) + r_pq(:)*scaldF
        E_p = E_p + 4.0_mk*eps*((sigma12/r_s_pq2**6) &
        &                          - (sigma6/r_s_pq2**3)) - E_prc
      endif
    end foreach
  end foreach
  foreach p in particles(parts) with positions(x) vec_props(F,a,v)
    v_p(:) = v_p(:) + 0.5_mk*(a_p(:) + (F_p(:)/m))*dt
  end foreach
  t = t + dt
end timeloop
```

Listing 8.2 The main time loop of a PPML Lennard-Jones simulation client.

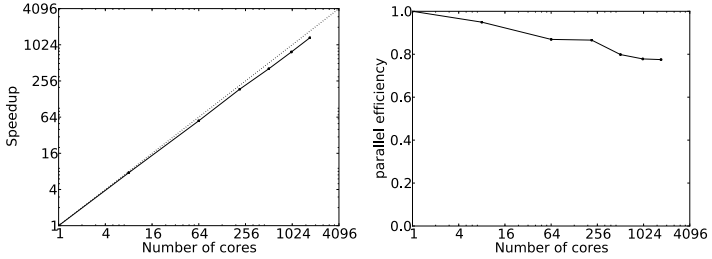


Figure 8.7. Speedup and parallel efficiency of the PPM Lennard-Jones benchmark client. The parallel efficiency remains at 77.5% on 1728 processes.

efficiency.

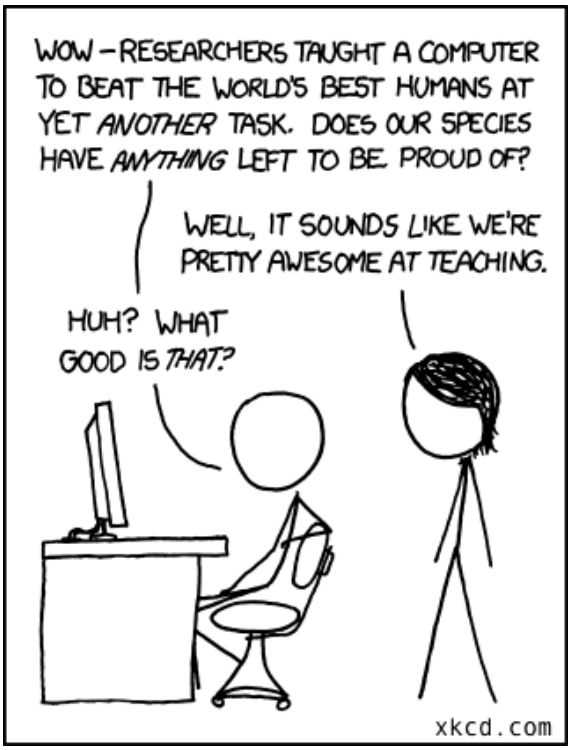
8.4. Conclusion

We have demonstrated the efficiency of the PPML language and the redesigned PPM library in two client applications. Both applications were written in PPML and used most of the introduced language features (c.f. section 7.3), as well as the client generator framework. Furthermore, we used PPM's new object-oriented interface as introduced in chapter 3. A DC-PSE reaction-diffusion client was written in 70 lines and a Lennard-Jones molecular dynamics client in 140 lines. Both clients natively support shared- and distributed-memory parallelism. Our benchmarks have shown parallel efficiencies of 70% to 80% for both clients 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 computing to quickly develop particle-mesh simulations for parallel hardware platforms.

Part IV.

Conclusions

Progeny


WOW – RESEARCHERS TAUGHT A COMPUTER TO BEAT THE WORLD'S BEST HUMANS AT YET *ANOTHER* TASK. DOES OUR SPECIES HAVE *ANYTHING* LEFT TO BE PROUD OF?
WELL, IT SOUNDS LIKE WE'RE PRETTY AWESOME AT TEACHING.
HUH? WHAT GOOD IS *THAT?*
xkcd.com

CHAPTER 9

Conclusions and future work

We have developed a domain-specific language for parallel hybrid particle-mesh methods using the PPM library [Sbalzarini et al., 2006a, Sbalzarini, 2010]. We redesigned PPM to be object-oriented, and extended the library with algorithms and data structures for adaptive-resolution particle methods on heterogeneous platforms. We provided support for multi- and manycore platforms through a new Fortran/POSIX threads library and OpenCL particle-to-mesh and mesh-to-particle interpolation routines. We tested and benchmarked the new version of PPM and the domain-specific language PPML in two example applications. Below, we summarize our main conclusions, highlight current limitations, and point to possible future improvements.

Toward an object-oriented PPM core. We presented an object-oriented redesign of the PPM library. We split the library into a *core* and a *nu-*

merics part. The core library provides abstract data types for topologies, particles, meshes, fields, and operators, while the numerics library provides implementations for numerical methods using the abstract data types from the core library. We have benchmarked an intermediate, modular and the final, object-oriented version of the redesigned library. We found that while modularizing the PPM library comes at no significant performance toll our object-oriented design does have a negative impact on the runtime and the parallel scaling of the particular client applications tested. The present implementation supports the described types, but the mesh type misses some high-level convenience functions for mapping and requires more systematic testing. Further analysis and tuning of the present PPM core implementation is needed to improve its performance to the level of previous versions. Furthermore, most solvers in PPM numerics do not yet use the new PPM core data structures and should be updated.

Fast neighbor lists for adaptive-resolution particle simulations. We described data structures and algorithms for efficiently finding interaction partners of particles with varying cutoff radii. Adaptive-resolution (AR) neighbor lists enable efficient computation of particle–particle interactions in adaptive-resolution simulations with a possibly continuous spectrum of cutoff radii. We benchmarked the construction and use of AR neighbor lists and compared them with cell lists and Verlet lists. The gained adaptivity comes at the cost of increased asymptotic runtime compared with conventional cell lists. If the number of interaction partners of each particle is bounded by a constant, the overall runtime of the algorithm for N particles is $O(\text{maxlevel} \times N \log N)$, instead of $O(N)$ for cell lists. The overhead in construction, however, is quickly amortized by the more efficient evaluation of particle–particle interactions or construction of Verlet lists. For realistic adaptive-resolution simulations, the presented AR neighbor list data structures can be several orders of magnitude faster than conventional cell lists. The AR neighbor lists are built per subdomain and do hence not create additional communication overhead in a parallel simulation, assuming that the ghost layers have been populated before.

Since memory access typically is the bottleneck, we believe that the runtime of AR cell lists could be improved by using a locality-preserving ordering of the particles, such as Hilbert-Peano ordering.

A new edge-coloring-based communication scheduler. We implemented a new communication scheduler for synchronous message passing in high-performance computing. Our algorithm is based on the DSATUR vertex-coloring algorithm [Brélaz, 1979], but operates on the line graph of the communication graph and uses a list of heaps for fast access to candidate vertices. We hence expect an asymptotic runtime of $O((n+m)\log n)$. We benchmarked the present implementation on a number of different graphs and compared it with the previous communication scheduler in PPM, which was based on Vizing's algorithm [Vizing, 1964, Misra and Gries, 1992]. We compared the two implementations on random graphs of increasing sizes and vertex degrees and on 2D and 3D Cartesian grids. The new implementation outperformed the previous one in all test cases. However, we noticed that the present algorithm (as well as Vizing's algorithm) is unsuitable for graphs with high adjacency degrees, since it assumes that all neighbors of a vertex can be traversed in $O(1)$ time.

The presented results are encouraging, but more tests on realistic and larger graphs from PPM domain decompositions are needed to further study the present implementation. In the context of communication-scheduling for high-performance computing, a distributed graph coloring algorithm such as the one presented by Boman et al. [2005] may also be a desirable alternative.

A pthreads wrapper for Fortran 2003. We presented a comprehensive Fortran 2003 wrapper (forthreads) for the POSIX threads library. The forthreads library provides interfaces to almost all thread management functions and synchronization constructs. It extends previous implementations [Hanson et al., 2002]. We demonstrated the versatility of forthreads in three examples, all implemented in the PPM library. First, we accelerated the existing particle-mesh interpolation routines in PPM using forthreads and OpenMP, and compared these two implementations in terms of code complexity and parallel efficiency. The forthreads version had a performance comparable to that of the OpenMP version. However, the gained flexibility of POSIX threads comes at the cost of a slight increase in code complexity. Second, we ported the existing multigrid Poisson solver of PPM to make use of a dedicated communication thread in order to overlap communication and computation. We benchmarked the new implementation

and compared it with the original non-overlapped version of this solver. The threaded implementation shows a decreased time efficiency over the pure MPI implementation. This, however, is likely not due to the use of forthreads, but the result of more complex index calculations that prevent the main loops from vectorizing. Third, we implemented a Fortran wrapper for the POSIX sockets library and used the sockets and pthreads wrappers in PPM to build a control server for real-time monitoring of PPM client applications over the internet.

The present implementation does not support pthreads functions that are implemented as C language macros or that heavily rely on C's `void` pointers. Future Fortran/C binding extensions might enable complete wrapping of the POSIX threads interface. One non-standard extension of the library that could be particularly useful in the context of heterogeneous distributed- and shared-memory parallelism is *thread pinning*. It instructs the operating system scheduler to allocate a thread onto a particular core of a processor and prevents it from migrating to other cores, reducing the associated overhead. A more extensive study of heterogeneous distributed- and shared-memory parallel implementations should assess what improvements can be gained in particular for particle-mesh methods.

An OpenCL implementation of particle-to-mesh and mesh-to-particle interpolation in 2D and 3D. We designed and implemented GPU-suited algorithms and data structures for particle-to-mesh and mesh-to-particle interpolation in 2D and 3D. Our implementation is portable, allowing execution on any multi- or manycore platform supporting OpenCL. It is also generic with respect to the mesh size, the input representation, the number of particles per mesh cell, and the number of particle properties that are to be interpolated. We benchmarked both accuracy and runtime of the presented implementation and compare it with a sequential shared-memory-parallel OpenMP version. The benchmarks show speedups of up to 22-fold over the sequential CPU code. When comparing with the OpenMP multi-threaded implementation executed on an 8-core CPU, the OpenCL code was in most cases slower but could yield speedups of up to 7-fold in the best case.

Despite the moderate speedups on the GPU, we believe that outsourcing

particle-to-mesh and mesh-to-particle interpolation may have the advantage that CPU resources are freed up to perform other tasks. In order to support this, our current implementation could be extended to allow GPU-CPU overlapping. As GPUs are becoming commonplace in supercomputers (Oak Ridge National Laboratory's Titan supercomputer features one GPU per compute node), we believe that the PPM library should make stronger use of GPUs. Due to their regular compute-structure, mesh operations are especially suited for SIMD computing. Therefore, it may be desirable to provide GPU implementations of mesh operations that could directly be applied to mesh data already residing on the GPU, reducing the data-transfer overhead.

A domain-specific language for particle methods. We have presented PPML, an embedded domain-specific language (eDSL) for parallel hybrid particle-mesh methods. We first reviewed existing DSLs for numerical and parallel applications and described the PPM abstractions introduced by Sbalzarini [2010]. Based on the PPM abstractions, we outlined the types and operations of PPML. Furthermore, we introduced particle and mesh iterators, and PPML-specific program scopes. We demonstrated PPML in two minimal applications simulating diffusion using both a mesh and a particle discretization.

The presented approach is analogous to Liszt [DeVito et al., 2011], a DSL for distributed-memory mesh-based PDE solvers, and DOLFIN [Logg and Wells, 2010], an abstraction layer and Python DSL for finite-element methods. Both Liszt and DOLFIN use code analysis and optimization techniques on the intermediate representation of the DSL code. Although this is currently not done in PPML, both code checking and optimization are in principle possible on the AST representation. Future work will also explore the possibility of using automatic model checkers for parallel programs, such as [Siegel et al., 2006], to provide the user with feedback on correctness already at an early stage of code development.

A current limitation of the PPML code generator is that it cannot inject code in arbitrary locations of the processed program scope. This currently prevents auto-generating initialization statements in PPM clients. Initialization statements, such as `ppm_init`, should be placed after the last variable

declaration, but before any user-defined code. To enable this feature, the PPML parser must be extended with rules for recognizing variable declarations. The more Fortran code is recognized by the parser, the more flexible the code generator will become in its output.

The current macro collection covers most aspects of implementing a PPML application. In the future, this collection should become larger, handling more special cases and providing more utilities. Thanks to the flexible design of PPML, macros can be added to the collection whenever needed.

Based on PPML, we developed a web-based visual programming environment enabling simple prototyping of PPML programs without sacrificing the ability to run large-scale simulations. The current implementation supports only basic PPML and Fortran features and provides only a limited set of blocks to the user. Future work will provide a generic mechanism for recognizing new PPML macros and representing them in the web-interface. Also, as cloud-based high-performance computing services are becoming more abundant (e.g., Amazon EC2, Google Compute Engine), an integration with automated deployment platforms such as Neptune [Bunch et al., 2011] could help make the present framework more accessible to the scientific community.

PPML benchmark clients. We demonstrated the usability of PPML and the scalability of the redesigned PPM library in two example simulations on up to 1936 cores. The PPML reaction-diffusion client consists of 70 lines of code and had a parallel efficiency of 75% on 1024 cores with wall-clock times of less than half a second per time step. The Lennard-Jones molecular dynamics simulation consists of 140 lines of code and had a parallel efficiency of 77.5% on 1728 cores. PPML significantly reduces code development times for mid-sized parallel simulations, such as the ones presented. This should, however, be further investigated using more complex clients containing both particles and meshes.

Lastly, larger benchmarks using the multi- and manycore extensions of PPM should be conducted in order to determine how the present framework performs in real-world applications.

APPENDIX A

The PPML ANTLR grammar

Listing A.1 We provide an abbreviated version of the PPML ANTLR grammer. The grammar syntax is derived from an Extended Backus-Naur Form, but is specialized to the ANTLR parser generator. We show here only the parser rules, without the production rules and the embedded Ruby language code. The complete PPML ANTLR grammar can be found in the `lib/grammar/CG.g` file in the PPML client generator software package.

```
grammar CG ;

options {
language = Ruby;
output = AST;
}

tokens {
PROCEDURE;
TYPE;
SCOPE;
SCOPE_START;
SCOPE_END;
INNER_STUFF;
```

```
    TYPE_BODY;
    USE;
    IMPORT;
    GENERIC;
    IMPLICIT;
    CONTAINS;
    FMACRO;
    IMACRO;
23  FOREACH;
    MODIFIERS;
    BODY;
    FLINE;
    TEXT;
    ARGS;
    RETARGS;
    NAMEDARGS;
    FDARG;
    RHS_SCOPE;
33  RHS_START;
    RHS_INNER;
    TIMELOOP;
    TEMPLATE;
    VLIST;
    VPAIR;
    BIND;
    }


43  prog
        : (naked_code)=>naked_code
        | scope_statement*
        ;

    // Scope detection - top level statements

    scope_statement
        : t=template?
          open=scope_start
53          i=inner_stuff
          close=scope_end
        | rhs_statement
        ;

    template
        : TEMPLATE_T s=STAR_T?
        LT_T
          v+=template_var (COMMA_T v+=template_var)*
        GT_T
63        n=NOINTERFACE_T? (SUFFIXES_T f=value_list)? NEWLINE_T
        ;

    template_var
        : n=ID_T COLON_T
        LEFT_SQUARE_T
          t+=template_type (COMMA_T t+=template_type)*
        RIGHT_SQUARE_T
```

```
    ;

template_type
    : (ID_T|TYPE_T) (LEFT_PAREN_T (ID_T|NUMBER_T) RIGHT_PAREN_T)?
    ;

scope_start
    :
  (kind=PROGRAM_T name=ID_T NEWLINE_T
  |kind=CLIENT_T name=ID_T NEWLINE_T
  |kind=MODULE_T name=ID_T NEWLINE_T
  |ABSTRACT_T? kind=INTERFACE_T name=ID_T? NEWLINE_T
  |RECURSIVE_T? kind=SUBROUTINE_T name=ID_T arglist? bind? NEWLINE_T
  |(ID_T (LEFT_PAREN_T (ID_T | NUMBER_T)
      RIGHT_PAREN_T)?)?
            kind=FUNCTION_T   name=ID_T   arglist?
  (RESULT_T LEFT_PAREN_T ID_T RIGHT_PAREN_T)? bind?  NEWLINE_T
    )
    ;

scope_end
    :
    ( {@context.last==:program}?=>
    (ENDPROGRAM_T | END_T PROGRAM_T) ID_T? NEWLINE_T
  | {@context.last==:client}?=>
    (ENDPROGRAM_T | END_T CLIENT_T) ID_T? NEWLINE_T
  | {@context.last==:module}?=>
    (ENDMODULE_T | END_T MODULE_T) ID_T? NEWLINE_T
  | {@context.last==:interface}?=>
    (ENDINTERFACE_T | END_T INTERFACE_T) ID_T? NEWLINE_T
  | {@context.last==:subroutine}?=>
    (ENDSUBROUTINE_T | END_T SUBROUTINE_T) ID_T? NEWLINE_T
  | {@context.last==:function}?=>
    (ENDFUNCTION_T | END_T FUNCTION_T) ID_T? NEWLINE_T
  )
    ;

rhs_statement
    : s=rhs_start
        i=rhs_inner_stuff
      e=rhs_end
    ;

rhs_start
    : RHS_T name=ID_T args=rhs_arglist NEWLINE_T
    ;

rhs_end
    : (END_T RHS_T | ENDRHS_T) NEWLINE_T
    ;

rhs_inner_stuff
    : pre+=line*
      GET_FIELDS_T ret=rhs_arglist NEWLINE_T
      post+=line*
    ;
```

```
    rhs_arglist
        : LEFT_PAREN_T
            ( args+=fd_arg
             (COMMA_T args+=fd_arg)*
            )?
133         RIGHT_PAREN_T
        ;

    // field and discretization
    fd_arg : field=ID_T (ARROW_T disc=ID_T)?
      ;

    type_statement
        : open=type_start
            i=type_body
143         close=type_end
        ;


    naked_code : line* ;

    // Scope detecion - start and end lines

    type_start : kind=TYPE_T
                ( (COMMA_T EXTENDS_T LEFT_PAREN_T ID_T RIGHT_PAREN_T)
153             | (COMMA_T ABSTRACT_T) )*
               (DOUBLE_COLON_T)? name=ID_T NEWLINE_T
            { @context << :type }
      ;
    type_end    : ( ENDTYPE_T | END_T TYPE_T ) ID_T? NEWLINE_T
            { @context.pop }
      ;

    // Scope detection - body

163 inner_stuff
        : (use+=use_statement)*
          (imp+=import_statement)*
          (implicit=implicit_none)?
          ({@input.peek(2) != PROGRAM_T}? body+=line)*
          (con=contains
                (sub+=scope_statement
                |sub+=imacro
                )+ )?
      ;
173
    type_body
        : ({@input.peek(2) != TYPE_T}? body+=line)*
          (con=contains
            ((procedure_statement)=> sub+=procedure_statement
            |sub+=generic_statement
            |{@input.peek(2) != TYPE_T}? body+=line)+ )?
            //|sub+=imacro)+ )?
      ;
```

```
183
    implicit_none
         : IMPLICIT_T NONE_T NEWLINE_T
         ;
    contains
         : CONTAINS_T NEWLINE_T
         ;
    use_statement
         : USE_T allowed* NEWLINE_T
         ;
193
    import_statement
         : IMPORT_T allowed* NEWLINE_T
         ;

    procedure_statement
         : PROCEDURE_T allowed* NEWLINE_T
         ;

    generic_statement
203      : GENERIC_T allowed* NEWLINE_T
         ;


    line
         : {fmacro_call?}?=> fcmacro
         | {imacro_call?}?=> imacro
         | scope_statement
         | (type_statement)=>type_statement
         | foreach
213      | timeloop
         | fline
         ;

    foreach
         : FOREACH_T it=ID_T IN_T name=ID_T a=arglist?
             (WITH_T (modifiers+=ID_T arglists+=arglist?)*)?
           NEWLINE_T
             ((loop_body)=>bodies+=loop_body
             |(bodies+=qualified_body)*
223          )
           e=foreach_end
         ;

    foreach_end: (ENDFOREACH_T | END_T FOREACH_T) NEWLINE_T
      ;


    loop_body
         : ({@input.peek(2) != FOREACH_T and @input.peek(2) != TIMELOOP_T}?
233        body+=line)*
         ;

    qualified_body
         : FOR_T name=ID_T NEWLINE_T
           ({@input.peek(2) != FOREACH_T}?
```

```
        body+=line)*
    ;


timeloop
    : t=ID_T EQUALS_T TIMELOOP_T tp=arglist NEWLINE_T
        body=loop_body
      e=timeloop_end
    ;

timeloop_end : (ENDTIMELOOP_T | END_T TIMELOOP_T) NEWLINE_T
  ;

fcmacro
    :  results=return_args name=ID_T args=arglist NEWLINE_T
  ;

return_args
    : (results+=ID_T (COMMA_T results+=ID_T)* EQUALS_T)?
    ;

imacro
    : MINCLUDE_T (name=ID_T)  args=arglist NEWLINE_T
    ;

bind
   : BIND_T LEFT_PAREN_T allowed* RIGHT_PAREN_T
   ;

arglist
    : LEFT_PAREN_T
  ( (args+=value | args+=value_list | names+=ID_T EQUALS_T
    (values+=value | values+=value_list))
  (COMMA_T (args+=value | args+=value_list | names+=ID_T EQUALS_T
    (values+=value | values+=value_list)))*)?
  RIGHT_PAREN_T
  ;

// Catchall

fline
    : (allowed*
  | MODULE_T PROCEDURE_T ID_T
  | PROCEDURE_T allowed*
  ) NEWLINE_T
  ;

allowed
    : ID_T
    | ANY_CHAR_T | DOT_T | STAR_T
    | NUMBER_T | STRING_T
    | LEFT_PAREN_T | RIGHT_PAREN_T | ARROW_T
    | LEFT_SQUARE_T | RIGHT_SQUARE_T
    | COMMA_T | EQUALS_T | DOUBLE_COLON_T | COLON_T | AMPERSAND_T
    | boolean | logical | comparison
    | END_T | IN_T | TYPE_T | ABSTRACT_T
```

```
    ;

value_list
    : LEFT_SQUARE_T (vals+=value | vals+=value_pair)
    (COMMA_T (vals+=value | vals+=value_pair))* RIGHT_SQUARE_T
    ;

value_pair : v1=ID_T ARROW_T v2=value -> ^(VPAIR $v1 $v2) ;
303
value : ID_T | NUMBER_T | STRING_T | CODE_T;


////////////////////////////////////////
// Lexer Rules
////////////////////////////////////////

// PPM Keywords

313 FOREACH_T       : 'FOREACH'       | 'foreach'       ;
    ENDFOREACH_T    : 'ENDFOREACH'    | 'endforeach'    ;
    FOR_T           : 'FOR'           | 'for'           ;
    IN_T            : 'IN'            | 'in'            ;
    RHS_T           : 'RHS'           | 'rhs'           ;
    ENDRHS_T        : 'ENDRHS'        | 'endrhs'        ;
    GET_FIELDS_T    : 'GET_FIELDS'    | 'get_fields'    ;
    TIMELOOP_T      : 'TIMELOOP'      | 'timeloop'      ;
    ENDTIMELOOP_T   : 'ENDTIMELOOP'   | 'endtimeloop'   ;
    WITH_T          : 'WITH'          | 'with'          ;
323 TEMPLATE_T      : 'TEMPLATE'      | 'template'      ;
    CLIENT_T        : 'CLIENT'        | 'client'        ;
    ENDCLIENT_T     : 'ENDCLIENT'     | 'endclient'     ;
    NOINTERFACE_T   : 'NOINTERFACE'   | 'nointerface'   ;
    SUFFIXES_T      : 'SUFFIXES'      | 'suffixes'      ;

// Fortran Keywords

    PROGRAM_T       : 'PROGRAM'       | 'program'       ;
    ENDPROGRAM_T    : 'ENDPROGRAM'    | 'endprogram'    ;
333 MODULE_T        : 'MODULE'        | 'module'        ;
    ENDMODULE_T     : 'ENDMODULE'     | 'endmodule'     ;
    INTERFACE_T     : 'INTERFACE'     | 'interface'     ;
    ENDINTERFACE_T  : 'ENDINTERFACE'  | 'endinterface'  ;
    SUBROUTINE_T    : 'SUBROUTINE'    | 'subroutine'    ;
    ENDSUBROUTINE_T : 'ENDSUBROUTINE' | 'endsubroutine' ;
    FUNCTION_T      : 'FUNCTION'      | 'function'      ;
    ENDFUNCTION_T   : 'ENDFUNCTION'   | 'endfunction'   ;
    END_T           : 'END'           | 'end'           ;
    USE_T           : 'USE'           | 'use'           ;
343 IMPLICIT_T      : 'IMPLICIT'      | 'implicit'      ;
    NONE_T          : 'NONE'          | 'none'          ;
    CONTAINS_T      : 'CONTAINS'      | 'contains'      ;
    PROCEDURE_T     : 'PROCEDURE'     | 'procedure'     ;
    RECURSIVE_T     : 'RECURSIVE'     | 'recursive'     ;
    RESULT_T        : 'RESULT'        | 'result'        ;
    TYPE_T          : 'TYPE'          | 'type'          ;
    ENDTYPE_T       : 'ENDTYPE'       | 'endtype'       ;
```

```
    EXTENDS_T       : 'EXTENDS'       | 'extends'       ;
    MINCLUDE_T      : 'MINCLUDE'      | 'minclude'      ;
353 ABSTRACT_T      : 'ABSTRACT'      | 'abstract'      ;
    GENERIC_T       : 'GENERIC'       | 'generic'       ;
    IMPORT_T        : 'IMPORT'        | 'import'        ;
    BIND_T          : 'BIND'          | 'bind'          ;
    // DEFAULT_T       : 'DEFAULT'       | 'default'       ;

    DOT_T
        : '.'
            (   (TRUE_T)=>  TRUE_T  {$type=TRUE_T}
            |   (FALSE_T)=> FALSE_T {$type=FALSE_T}
363         |   (AND_T)=>   AND_T   {$type=AND_T}
            |   (OR_T)=>    OR_T    {$type=OR_T}
            |   (NOT_T)=>   NOT_T   {$type=NOT_T}
            |   (EQV_T)=>   EQV_T   {$type=EQV_T}
            |   (NEQV_T)=>  NEQV_T  {$type=NEQV_T}
            |   (GT)=>      GT      {$type=GT_T}
            |   (LT)=>      LT      {$type=LT_T}
            |   (GE)=>      GE      {$type=GE_T}
            |   (LE)=>      LE      {$type=LE_T}
            |   (EQ)=>      EQ      {$type=EQ_T}
373         |   (NE)=>      NE      {$type=NE_T}
            )?
        ;

    // True/False

    boolean : TRUE_T | FALSE_T ;

    fragment
    TRUE_T  : 'TRUE.'  | 'true.'  ;
383 fragment
    FALSE_T : 'FALSE.' | 'false.' ;

    // Logical

    logical : AND_T | OR_T | NOT_T | EQV_T | NEQV_T ;

    fragment
    AND_T  : 'AND.'  | 'and.'  ;
    fragment
393 OR_T   : 'OR.'   | 'or.'   ;
    fragment
    NOT_T  : 'NOT.'  | 'not.'  ;
    fragment
    EQV_T  : 'EQV.'  | 'eqv.'  ;
    fragment
    NEQV_T : 'NEQV.' | 'neqv.' ;


    // Comparison
403
    comparison : GT_T | LT_T | GE_T | LE_T | EQ_T | NE_T ;

    fragment
```

```
    GT    : 'GT.' | 'gt.' ;
    GT_T :   '>'  ;
    fragment
    LT    : 'LT.' | 'lt.' ;
    LT_T :   '<'  ;
    fragment
413 GE    : 'GE.' | 'ge.' ;
    GE_T :   '>=' ;
    fragment
    LE    : 'LE.' | 'le.' ;
    LE_T :   '<=' ;
    fragment
    EQ    : 'EQ.' | 'eq.' ;
    EQ_T :   '==' ;
    fragment
    NE    : 'NE.' | 'ne.' ;
423 NE_T :   '/=' ;

    // Identifiers

    ID_T : (ALPHA | '_') (ALNUM | '_' | '%')* ;

    // Constants

    CODE_T
        : START_CODE .* STOP_CODE
433     ;

    fragment
    START_CODE : '<#' ;

    fragment
    STOP_CODE : '#>' ;


    STRING_T
443     : '"' ('\\"'|~'"')* '"'
        | '\'' ('\\\''|~'\'')* '\''
        ;

    NUMBER_T
        : ('-')? DIGIT+
            ((DECIMAL)=> DECIMAL ((KIND)=> KIND)?)?
        ;

    fragment
453 DECIMAL : '.' DIGIT+ ;
    fragment
    KIND : '_' (ID_T | DIGIT+) ;



    EQUALS_T        : '='  ;
    LEFT_PAREN_T    : '('  ;
    RIGHT_PAREN_T   : ')'  ;
    LEFT_SQUARE_T   : '['  ;
```

```
463 RIGHT_SQUARE_T : ']'   ;
    AMPERSAND_T    : '&'   ;
    DOUBLE_COLON_T : '::'  ;
    COLON_T        : ':'   ;
    COMMA_T        : ','   ;
    ARROW_T        : '=>'  ;
    STAR_T         : '*'   ;

    // Fragments

473 fragment
    COMMENT : '!' ~('\n'|'\r')* ;
    fragment
    WS      : (' '|'\r'|'\t'|'\u000C')* ;

    fragment
    ALNUM : ( ALPHA | DIGIT ) ;
    fragment
    ALPHA : 'a'..'z' | 'A'..'Z' ;
    fragment
483 DIGIT : '0'..'9' ;

    ANY_CHAR_T : ~( '\r' | '\n' | ' ' | '\t' ) ;
```

APPENDIX B

A minimal PPM client

The following listing shows the auto-generated main program and modules of the DC-PSE diffusion client (listing 7.3) from section 7.5.1.

```
program mini

  use ppm_autogenerated_global
  use ppm_autogenerated_rhs
  implicit none
  include 'mpif.h'
  character(len=4) :: caller = 'mini'
  integer :: info
  integer :: comm
  integer :: rank
  integer :: nproc
  class(ppm_t_field_), pointer :: U
  real(mk), dimension(:), pointer :: genjb3_cost => null()
  type(ppm_t_particles_d), pointer :: c
  integer :: genmjb_info
  INTEGER :: gen794_info
  integer :: gen794_p
  real(mk), dimension(:,:), pointer :: gen794_x => null()
  real(mk), dimension(:), pointer :: gen794_U_wp => null()
```

```
      class(ppm_t_part_prop_d_), pointer :: genpag_temp_prop => null()
      class(ppm_t_neighlist_d_), pointer :: n
      type(ppm_t_options_op) :: gen8l3_opopts
      integer, parameter :: eulerf = ppm_param_ode_scheme_eulerf
      integer, parameter :: tvdrk2 = ppm_param_ode_scheme_tvdrk2
      integer, parameter :: midrk2 = ppm_param_ode_scheme_midrk2
      integer, parameter :: rk4 = ppm_param_ode_scheme_rk4
      integer, parameter :: sts = ppm_param_ode_scheme_sts
      type(ppm_t_ode) :: o
      integer :: nstages
      class(ppm_v_main_abstr), pointer :: genn9p_vars
      procedure(ppm_p_rhsfunc_d), pointer :: genn9p_rhs_ptr
      class(ppm_v_var_discr_pair), pointer :: genn9p_rhs_vars
      class(ppm_t_var_discr_pair), pointer :: genn9p_pair
      class(ppm_t_field_discr_pair), pointer :: genn9p_fpair
      class(ppm_t_main_abstr), pointer :: genn9p_el
      class(ppm_t_part_prop_d_), pointer :: genhhl_temp_prop => null()
      real(mk) :: t
      integer :: genip6_info
      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)


      call MPI_Init(info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "MPI_Init failed.",&
          caller, 10 , info)
        GOTO 9999
      END IF

      comm = MPI_COMM_WORLD

      call MPI_Comm_Size(comm, nproc, info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "MPI_Comm_Size failed.",&
          caller, 10 , info)
        GOTO 9999
      END IF

      call MPI_Comm_Rank(comm, rank, info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "MPI_Comm_Rank failed.",&
          caller, 10 , info)
        GOTO 9999
      END IF

      call define_args
```

```
      call parse_args(info)
      if (info .eq. exit_gracefully) then
        goto 9999
      else
        IF (info.NE.0) THEN
          info = ppm_error_error
          CALL ppm_error(ppm_err_sub_failed, &
            "Parse args failed.",&
            caller, 10 , info)
          GOTO 9999
        END IF
      end if

      call ppm_init(2, ppm_kind_double, &
                    -14, comm,&
                    0, info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "ppm_init failed.",&
          caller, 10 , info)
        GOTO 9999
      END IF


      allocate(ppm_t_field::U,stat=info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_alloc, &
          "Allocating the U field",&
          caller, 12 , info)
        GOTO 9999
      END IF

      call U%create(1, info, name="U")
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "Create field failed!",&
          caller, 12 , info)
        GOTO 9999
      END IF

      call ppm_mktopo(topo,domain_decomposition,processor_assignment,&
      &                min_phys,max_phys,bcdef,ghost_size,&
      &                genjb3_cost,info)

      allocate(c,stat=info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_alloc, &
          "Could not allocate c",&
          caller, 16 , info)
        GOTO 9999
      END IF
      call c%initialize(Npart, info, topoid=topo, &
```

```
     &                    distrib=ppm_param_part_init_cartesian)
     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)
     call c%map(info,global=.true.,topoid=topo)
     IF (info.NE.0) THEN
       info = ppm_error_error
       CALL ppm_error(ppm_err_sub_failed, &
         "Global mapping failed.",&
         caller, 22 , info)
       GOTO 9999
     END IF

     call U%discretize_on(c,genmjb_info)
     call c%get_xp(gen794_x,gen794_info)
     IF (gen794_info.NE.0) THEN
       gen794_info = ppm_error_error
       CALL ppm_error(ppm_err_sub_failed, &
         "getting positions c",&
         caller, 26 , gen794_info)
       GOTO 9999
     END IF
     call c%get(U,gen794_U_wp,gen794_info)
     IF (gen794_info.NE.0) THEN
       gen794_info = ppm_error_error
       CALL ppm_error(ppm_err_sub_failed, &
         "getting field U for c",&
         caller, 26 , gen794_info)
       GOTO 9999
     END IF
     do gen794_p=1,c%Npart
       gen794_U_wp(gen794_p) = 1.0_mk/(2.0_mk*pi*sigma(1)*sigma(2))*
   &
           &      exp(-0.5_mk*(((gen794_x(1,gen794_p)-0.5_mk)**2/sigma(1)**2)+ &
           &                   ((gen794_x(2,gen794_p)-0.5_mk)**2/sigma(2)**2)))
     end do

     call c%map_ghosts(info)

     call c%comp_neighlist(info,cutoff=2.5_mk * c%h_avg)
     IF (info.NE.0) THEN
       info = ppm_error_error
       CALL ppm_error(ppm_err_sub_failed, &
         "Could not compute neighlist",&
         caller, 33 , info)
       GOTO 9999
     END IF
     n => c%get_neighlist()
     allocate(Lap)
     call Lap%create(2, (/1.0_mk,1.0_mk/), (/2,0,0,2/), info, name='"Laplacian"')
     call gen8l3_opopts%create(ppm_param_op_dcpse,info, order=2, c=1.0_mk)
     IF (info.NE.0) THEN
       info = ppm_error_error
       CALL ppm_error(ppm_err_sub_failed, &
```

```
        "failed to initialize option object for operator",&
        caller, 35 , info)
      GOTO 9999
    END IF
    call Lap%discretize_on(c, L, gen8l3_opopts, info)

    allocate(genn9p_vars,stat=info)
    IF (info.NE.0) THEN
      info = ppm_error_error
      CALL ppm_error(ppm_err_alloc, &
        "Allocating genn9p_vars vector",&
        caller, 37 , info)
      GOTO 9999
    END IF
    allocate(genn9p_rhs_vars,stat=info)
    IF (info.NE.0) THEN
      info = ppm_error_error
      CALL ppm_error(ppm_err_alloc, &
        "Allocating genn9p_rhs_vars vector",&
        caller, 37 , info)
      GOTO 9999
    END IF


    genn9p_el => U
    call genn9p_vars%push(genn9p_el,info)
    IF (info.NE.0) THEN
      info = ppm_error_error
      CALL ppm_error(ppm_err_sub_failed, &
        "Pushing element failed",&
        caller, 37 , info)
      GOTO 9999
    END IF

    allocate(genn9p_pair,stat=info)
    genn9p_pair%var => U
    genn9p_pair%discr => c
    call genn9p_rhs_vars%push(genn9p_pair,info)
    IF (info.NE.0) THEN
      info = ppm_error_error
      CALL ppm_error(ppm_err_sub_failed, &
        "Pushing genn9p_pair failed",&
        caller, 37 , info)
      GOTO 9999
    END IF

    genn9p_rhs_ptr => mini_rhs

    call o%create(eulerf,genn9p_vars,genn9p_rhs_ptr,genn9p_rhs_vars,info)

    nstages = o%integrator%scheme_nstages

    t = start_time
    do while (t .le. stop_time)
      do istage=1,nstages
      c%flags(ppm_part_partial) = .true. ! hack
```

```
        c%flags(ppm_part_areinside) = .true.
        c%flags(ppm_part_ghosts) = .true.
        genhhl_temp_prop => c%props%begin()
        do while (associated(genhhl_temp_prop))
          genhhl_temp_prop%flags(ppm_ppt_partial) = .true.
          genhhl_temp_prop => c%props%next()
        enddo
        call c%map_ghost_push_positions(info)
        call c%map_ghosts(info)
        call c%map_ghost_pop_positions(info)

        call o%step(t,time_step,istage,info)

        end do
        st = st + 1
        if (ppm_rank.eq.0) print *,st
      end do


      call ppm_finalize(genip6_info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "ppm_finalize failed.",&
          caller, 48 , info)
        GOTO 9999
      END IF

      call MPI_Finalize(genip6_info)
      IF (info.NE.0) THEN
        info = ppm_error_error
        CALL ppm_error(ppm_err_sub_failed, &
          "MPI_Finalize failed.",&
          caller, 48 , info)
        GOTO 9999
      END IF
    9999 continue
    end program

    module ppm_autogenerated_global
      use ppm_module_core
      use ppm_module_user_numerics
      implicit none
      integer, parameter :: mk = ppm_kind_double
      integer :: st = 0
      real(mk) :: D_u
      integer :: topo = 0
      real(mk), dimension(:), pointer :: min_phys
      real(mk), dimension(:), pointer :: max_phys
      integer :: domain_decomposition
      integer :: processor_assignment
      real(mk) :: ghost_size
      integer :: Npart
      type(ppm_t_operator), pointer :: Lap => null()
      class(ppm_t_operator_discr), pointer :: L => null()
      integer :: ODEscheme
```

```
  real(mk) :: start_time
  real(mk) :: time_step
  real(mk) :: stop_time
  contains
  subroutine define_args
    implicit none
    call arg(D_u, 'D_u', &
             default = 1.0_mk, &
             min = 0.0_mk, &
             ctrl_name = 'Du_param', &
             long_flag = 'Du_param', &
             help = 'Diffusion constant of U')
    call arg_group('Domain Parameters')
    allocate(min_phys(2))
    call arg(min_phys, 'min_phys', &
             default = (/0.0_mk, 0.0_mk/), &
             ctrl_name = 'min_phys', &
             long_flag = 'min_phys', &
             help = 'lower domain boundary')
    allocate(max_phys(2))
    call arg(max_phys, 'max_phys', &
             default = (/1.0_mk, 1.0_mk/), &
             ctrl_name = 'max_phys', &
             long_flag = 'max_phys', &
             help = 'upper domain boundary')
    call arg(domain_decomposition, 'domain_decomposition', &
             default = ppm_param_decomp_cuboid, &
             min = 1, &
             max = 13, &
             ctrl_name = 'domain_decomposition', &
             help = 'Domain decomposition, one of:\n'&
    &             //'* 1 - tree\n'&
    &             //'* 2 - pruned cell\n'&
    &             //'* 3 - bisection\n'&
    &             //'* 4 - x pencil\n'&
    &             //'* 5 - y pencil\n'&
    &             //'* 6 - z pencil\n'&
    &             //'* 7 - cuboid\n'&
    &             //'* 8 - user defined\n'&
    &             //'* 10 - xy slab\n'&
    &             //'* 11 - xz slab\n'&
    &             //'* 12 - yz slab\n'&
    &             //'* 13 - cartesian')
    call arg(processor_assignment, 'processor_assignment', &
             default = ppm_param_assign_internal, &
             min = 1, &
             max = 6, &
             ctrl_name = 'processor_assignment', &
             help = 'Processor assignment, one of:\n'&
    &             //'* 1 - internal\nmetis:\n'&
    &             //'* 2 - nodal cut\n'&
    &             //'* 3 - nodal comm\n'&
    &             //'* 4 - dual cut\n'&
    &             //'* 5 - dual comm\n'&
    &             //'* 6 - user defined')
    call arg(ghost_size, 'ghost_size', &
```

```
355                 default = 0.021_mk, &
                    min = 0.0_mk, &
                    ctrl_name = 'ghost_size', &
                    long_flag = 'ghost_size', &
                    help = 'Ghost layer width')
        call arg(Npart, 'Npart', &
                    default = 10000, &
                    ctrl_name = 'Npart', &
                    long_flag = 'npart', &
                    help = 'Global number of particles')
365     call arg_group('ODE Parameters')
        call arg(ODEscheme, 'ODEscheme', &
                    default = ppm_param_ode_scheme_eulerf, &
                    ctrl_name = 'ODEscheme', &
                    help = 'ODE integrator')
        call arg_group('Time Parameters')
        call arg(start_time, 'start_time', &
                    default = 0.0_mk, &
                    min = 0.0_mk, &
                    ctrl_name = 'start_time', &
375                 help = 'Start time')
        call arg(time_step, 'time_step', &
                    default = 0.1_mk, &
                    min = 0.0_mk, &
                    ctrl_name = 'time_step', &
                    help = 'Time step')
        call arg(stop_time, 'stop_time', &
                    default = 1.0_mk, &
                    min = 0.0_mk, &
                    ctrl_name = 'stop_time', &
385                 help = 'End time')
      end subroutine define_args

    end module ppm_autogenerated_global

    module ppm_autogenerated_rhs
      implicit none
      contains
      function mini_rhs(vars_discr, time, changes)
        use ppm_autogenerated_global
395     implicit none
        real(ppm_kind_double) :: mini_rhs
        class(ppm_v_var_discr_pair), pointer :: vars_discr
        real(ppm_kind_double) :: time
        class(ppm_v_main_abstr), pointer :: changes
        class(ppm_t_main_abstr), pointer :: change => null()
        class(ppm_t_var_discr_pair), pointer :: vd_pair => null()
        class(ppm_t_discr_info_), pointer :: di => null()
        character(len=21) :: caller='ppm_autogenerated_rhs'
        class(ppm_t_field_), pointer :: U
405     class(ppm_t_particles_d), pointer :: parts
        class(ppm_t_field_), pointer :: dU
        integer :: info

        mini_rhs = 0
        vd_pair => vars_discr%at(1)
```

```
      select type(vdpairvar => vd_pair%var)
      class is (ppm_t_field_)
        U => vdpairvar
      end select
      select type(vdpairdiscr => vd_pair%discr)
      class is (ppm_t_particles_d)
        parts => vdpairdiscr
      end select
      change => changes%at(1)
      select type(change)
      class is (ppm_t_field_)
        dU => change
      end select
      call L%compute(U, dU, info)
        IF (info.NE.0) THEN
          info = ppm_error_error
          CALL ppm_error(ppm_err_sub_failed, &
            "Operator computation failed",&
            caller, 53 , info)
          GOTO 9999
        END IF
    9999 continue
    end function mini_rhs

end module ppm_autogenerated_rhs
```

APPENDIX C

A webCG PPML showcase

We show the graphical webCG version of the minimal diffusion client presented in section 7.5.1. The diagram is divided into four parts with wires crossing figure boundaries labeled A to F.

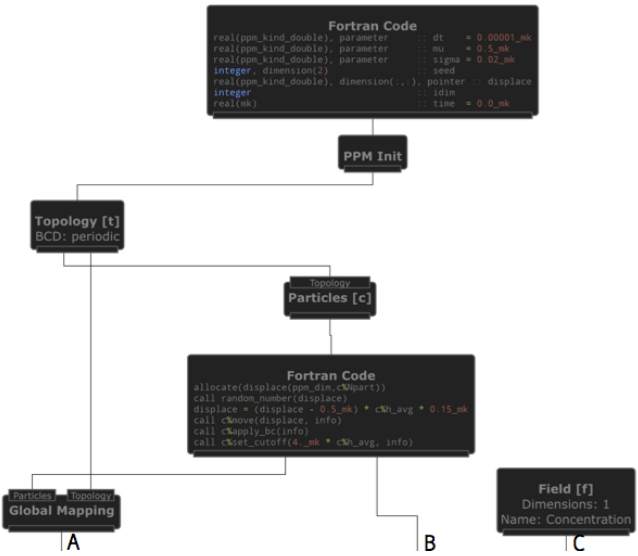


Figure C.1. The initialization part of the program. A topology and a particle set is created. The particles are mapped onto the topology and the the field f is created. The diagram also shows Fortran code blocks providing code that is not expressed in PPML.

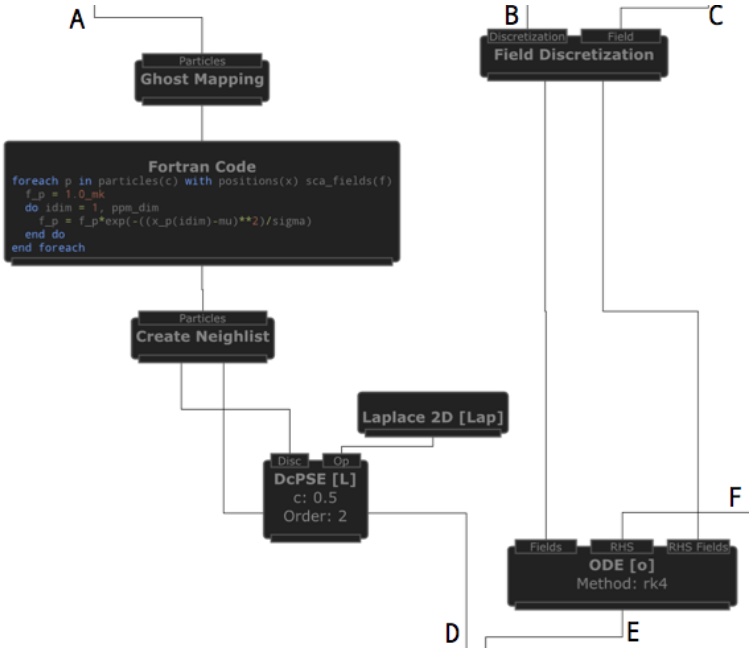


Figure C.2. The field is discretized onto the particles and a ghost-get mapping is performed. Then, we compute the Verlet list on the particle set and create the DC-PSE operators of order 2. The ODE object is created, taking the right-hand side and the field f as arguments.

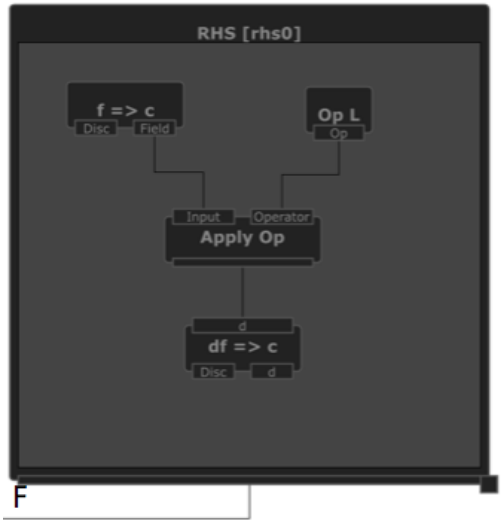


Figure C.3. Inside the right-hand side function we apply the DC-PSE discretized Laplacian operators to the particle set.

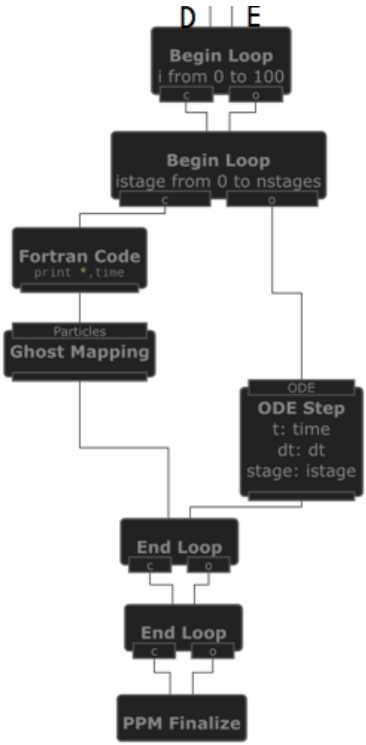


Figure C.4. A nested loop iterates through time steps and ODE integrator stages, calling the ODE step operation at each stage.

Bibliography

S. Adami, X. Y. Hu, and N. A. Adams. A new surface-tension formulation for multi-phase SPH using a reproducing divergence approximation. *J. Comput. Phys.*, 229(13):5011 – 5021, 2010.

S. Adami, X. Y. Hu, and N. A. Adams. A generalized wall boundary condition for smoothed particle hydrodynamics. *J. Comput. Phys.*, 231 (21):7057 – 7075, 2012.

M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, Oxford, 1987.

A. M. Altenhoff, J. H. Walther, and P. Koumoutsakos. A stochastic boundary forcing for dissipative particle dynamics. *J. Comput. Phys.*, 225(1): 1125 – 1136, July 2007.

S. Andova, M. van den Brand, and L. Engelen. Prototyping the semantics of a DSL using ASF+SDF: Link to formal verification of DSL models. In *Proc. 2nd Intl. Workshop on Algebraic Methods in Model-based Software Engineering,* Zurich, Switzerland, 30th June 2011, volume 56, pages 65–79. Open Publishing Association, 2011.

L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. SIGMOD, Intl. Conf. Management of Data*, pages 347–358, Paris, France, 2004. ACM.

K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10): 56–67, October 2009.

O. Awile, Ö. Demirel, and I. F. Sbalzarini. Toward an object-oriented core of the PPM library. In *Proc. ICNAAM*, pages 1313–1316. AIP, 2010.

J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. *Proc. Int. Conf. Inf. Process.*, 1959.

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. Technical Report ANL-95/11 – Revision 2.1.5, Argonne National Laboratory, 2004.

S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

J. Barnes and P. Hut. A hierarchical O(Nlog$N$) force-calculation algorithm. *Nature*, 324:446–449, 1986.

J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, 1975.

M. Bergdorf, G.-H. Cottet, and P. Koumoutsakos. Multilevel adaptive particle methods for convection-diffusion equations. *Multiscale Model. Simul.*, 4(1):328–357, 2005.

M. Bergdorf, P. Koumoutsakos, and A. Leonard. Direct numerical simulations of vortex rings at Re$\tilde{}$ γ= 7500. *J. Fluid. Mech.*, 581:495, 2007.

M. Bergdorf, I. F. Sbalzarini, and P. Koumoutsakos. A Lagrangian particle method for reaction-diffusion systems on deforming surfaces. *J. Math. Biol.*, 61:649–663, 2010.

O. Bernard, D. Friboulet, P. Thevenaz, and M. Unser. Variational b-spline level-set: A linear filtering approach for fast deformable model evolution. *IEEE Trans. Image Process.*, 18(6):1179 –1191, june 2009.

M. Bernaschi, M. Bisson, T. Endo, S. Matsuoka, M. Fatica, and S. Melchionna. Petaflop biofluidics simulations on a two million-core system. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 4:1–4:12. ACM, 2011.

X. Bian, S. Litvinov, R. Qian, M. Ellero, and N. A. Adams. Multiscale modeling of particle in suspension with smoothed dissipative particle dynamics. *Phys. Fluids*, 24(1):012002, 2012.

J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, 2005.

E.G. Boman, D. Bozdag, U. Catalyurek, A. H. Gebremedhin, and F. Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *Proc. Euro-Par 2005 Parallel Processing*, pages 241–251. Springer, 2005.

E. Bonomi, M. Flück, P. George, R. Gruber, R. Herbin, A. Perronnet, S. Merazzi, J. Rappaz, T. Richner, V. Schmid, P. Stehlin, C. Tran, M. Vidrascu, W. Voirol, and J. Vos. Astrid: Structured finite element and finite volume programs adapted to parallel vectorcomputers. *Comput. Phys. Rep.*, 11: 81–116, 1989.

A Brandt. Multi-level adaptive solutions to boundary-value problems. *Math. Comput.*, 31(138):333–390, 1977.

D Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, 01 2010.

D. Brown, G. Chesshire, W. Henshaw, and D. Quinlan. OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proc. of the SIAM Parallel Conference*. SIAM, 1997.

Chris Bunch, Navraj Chohan, Chandra Krintz, and Khawaja Shams. Neptune: a domain specific language for deploying hpc software on cloud platforms. In *Proc. 2nd Intl. Workshop on Scientific cloud computing*, ScienceCloud '11, pages 59–68. ACM, 2011.

A. Canedo, T. Yoshizawa, and H. Komatsu. Automatic parallelization of simulink applications. In *Proc. of the 8th annual IEEE/ACM Int. symposium on Code generation and optimization*, CGO '10, pages 151–159. ACM, 2010.

J. Cardinale, G. Paul, and I. F. Sbalzarini. Discrete region competition for unknown numbers of connected regions. *IEEE Trans. Image Process.*, 21(8):3531–3545, 2012.

N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

H. Chao, Z. Gengbin, and V. K. Laxmikant. Supporting adaptivity in MPI for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

Simon R. Chapple and Lyndon J. Clarke. The parallel utilities library. In *Proc. IEEE Scalable Parallel Libraries Conference*, pages 21–30. IEEE, 1994.

P. Charles, C. Grothoff, V. Saraswat, C.r Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not. Not.*, 40(10):519–538, October 2005.

P. Chatelain, G.-H. Cottet, and P. Koumoutsakos. Particle mesh hydrodynamics for astrophysics simulations. *Int. J. Mod. Phys. C.*, 18(04): 610–618, 2007.

P. Chatelain, A. Curioni, M. Bergdorf, D. Rossinelli, W. Andreoni, and P. Koumoutsakos. Billion vortex particle direct numerical simulations of aircraft wakes. *Comput. Method. Appl. Mech. Engrg.*, 197:1296–1304, 2008.

A. A. Chialvo and P. G. Debenedetti. On the use of the Verlet neighbor list in molecuar dynamics. *Comput. Phys. Commun.*, 60:215–224, 1990.

A. A. Chialvo and P. G. Debenedetti. On the performance of an automated Verlet neighbor list algorithm for large systems on a vector processor. *Comput. Phys. Commun.*, pages 15–18, 1991.

A. Clementi, P. Crescenzi, P. Penna, G. Rossi, and P. Vocca. On the complexity of computing minimum energy consumption broadcast subgraphs. *Proc. Symp. Theoretical Aspects of Computer Science*, pages 121–131, 2001.

E. Coffman, Jr., M. Garey, D. Johnson, and A. LaPaugh. Scheduling file transfers. *SIAM J. Comput.*, 14(3):744–780, 1985.

C. Conti, D. Rossinelli, and P. Koumoutsakos. GPU and APU computations of finite time Lyapunov exponent fields. *J. Comput. Phys.*, 231:2229–2244, 2012.

G.-H. Cottet and P. Koumoutsakos. *Vortex Methods – Theory and Practice*. Cambridge University Press, 2000.

G.-H. Cottet, P. Koumoutsakos, and M. L. Ould Salihi. Vortex methods with spatially varying cores. *J. Comput. Phys.*, 162(1):164 – 185, 2000.

C. De Michele. Optimizing event-driven simulations. *Comput. Phys. Commun.*, 182(9):1846 – 1850, 2011.

Davison J. de St. Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A massively parallel problem solving environment. In *Proc. HPDC'00: Ninth IEEE Intl. Symp. on High Performance and Distributed Computing*. IEEE, 2000.

V.K. Decyk, C.D. Norton, and B.K. Szymanski. How to support inheritance and run-time polymorphism in fortran 90. *Comp. Phys. Commun.*, 115 (1):9 – 17, 1998.

A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A Generic Interface for Parallel and Adaptive Scientific Computing: Abstraction Principles and the DUNE-FEM Module. *Computing*, 90(3–4):165–196, 2010.

P. Degond and S. Mas-Gallic. The weighted particle method for convection-diffusion equations. Part 2: The anisotropic case. *Math. Comput.*, 53 (188):509–525, 1989a.

P. Degond and S. Mas-Gallic. The weighted particle method for convection-diffusion equations. Part 1: The case of an isotropic viscosity. *Math. Comput.*, 53(188):485–507, 1989b.

Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12. ACM, 2011.

E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–569, September 1965.

P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programmingopencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Comput.*, 38(8):391 – 407, 2012.

A. Dubey, C. Daley, and K. Weide. Challenges of computing with flash on largest hpc platforms. *AIP Conf. Proc.*, 1281(1):1773–1776, 2010.

R. Fraedrich, S. Auer, and R. Westermann. Efficient high-quality volume rendering of SPH data. *IEEE Trans. Vis. Comput. Graphics*, 16(6): 1533–1540, 2010.

D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11 – 13, May 2005.

R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Mon. Not. R. Astron. Soc.*, 181:375–378, 1977.

P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola. Interactive SPH simulation and rendering on the GPU. In *Proc. ACM SIGGRAPH/Eurographics Symp. on Computer Animation*, pages 55–64, 2010.

C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *Image Processing, IEEE Transactions on*, 5(5):794 –797, may 1996.

P. Gray and S. K. Scott. Autocatalytic reactions in the isothermal, continuous stirred tank reactor. oscillations and instabilities in the system A + 2B → 3B; B → C. *Chem. Eng. Sci.*, 39(6):1087–1097, 1984.

L. Greengard and V. Rokhlin. The rapid evaluation of potential fields in three dimensions. *Lect. Notes Math.*, 1360:121–141, 1988.

M. Griebel and G. Zumbusch. Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization. *Contemp. Math.*, 218:271–278, 1998.

A Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD, Intl. Conf. Management of Data*, pages 47–57. ACM, 1984.

R. J. Hanson, C. P. Breshears, and H. A. Gabb. Algorithm 821: A fortran interface to posix threads. *ACM Trans. Math. Softw.s. Math. Softw.*, 28 (3):354–371, September 2002.

F. H. Harlow. Particle-in-cell computing method for fluid dynamics. *Methods Comput. Phys.*, 3:319–343, 1964.

T. N. Heinz and P. H. Hünenberger. A fast pairlist-construction algorithm for molecular simulations under periodic boundary conditions. *J. Comput. Chem. Chem.*, 25(12):1474–1486, 2004.

B. Hejazialhosseini, D. Rossinelli, C. Conti, and P. Koumoutsakos. High throughput software for direct numerical simulations of compressible two-phase flows. In *Proc. 2012 Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 16:1–16:12. IEEE Computer Society Press, 2012.

M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2): 289–300, May 1993.

L. Hernquist and N. Katz. TREESPH – a unification of SPH with the hierarchical tree method. *Astrophys. J. Suppl. Ser.*, 70:419–446, June 1989.

Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

S. E. Hieber and P. Koumoutsakos. A Lagrangian particle level set method. *J. Comput. Phys.*, 210:342–367, 2005.

C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–16, 1962.

R. W. Hockney and J. W. Eastwood. *Computer Simulation using Particles*. Institute of Physics Publishing, 1988.

R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Hilger, Bristol, 1981.

W. Hönig, F. Schmitt, R. Widera, H. Burau, G. Juckeland, MS Müller, and M. Bussmann. A generic approach for developing highly scalable particle-mesh codes for GPUs. Technical report, Forschungszentrum Dresden-Rossendorf & Technical University of Dresden, Dresden, Germany, 2010.

T. Y. Hou. Convergence of a variable blob vortex method for the Euler and Navier-Stokes equations. *SIAM J. Num. Analysis*, 27(6):1387–1404, 1990.

E. N. Houstis, E. Gallopoulos, R. Bramley, and J. Rice. Problem-solving environments for computational science. *IEEE Comput. Sci. Eng.*, 4(3): 18 –21, July-Sept. 1997.

E. N. Houstis, J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. Pellpack: a problem-solving environment

for pde-based applications on multicomputer platforms. *ACM Trans. Math. Softw.*, 24(1):30–73, March 1998.

P. J. in't Veld, S. J. Plimpton, and G. S. Grest. Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics. *Comput. Phys. Commun.*, 179(5):320 – 329, 2008.

J. E. Jones. On the determination of molecular fields. ii. from the equation of state of a gas. *Proc. Roy. Soc. Lond. Series A*, 106(738):463–477, 1924.

S. R. Karmesin, J. Crotinger, J. C. Cummings, S. Haney, W. J. Humphrey, J. V. W. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In *Proc. of the 2nd Intl. Symp. on Computing in Object-Oriented Parallel Environments*, volume 1505, pages 231–238. Springer, 1998.

R.M. Karp. Reducibility among combinatorial problems. *Complex. of Comput. Comput.*, pages 85–103, 1972.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. 20(1):359–392, 1998.

Khronos. *The OpenCL Specification, version 1.0*. Khronos OpenCL Working Group, June 2009.

D. B. Kirk and W. M. W. Hwu. *Programming massively parallel processors: A Hands-on approach*. Morgan Kaufmann, 2010.

A. Kosowski and K. Manuszewski. *Graph Colorings*, volume 352, chapter Classical coloring of graphs, pages 1–19. American Mathematical Society, 2004.

P. Koumoutsakos. Inviscid axisymmetrization of an elliptical vortex. *J. Comput. Phys.*, 138:821–857, 1997.

P. Koumoutsakos. Multiscale flow simulations using particles. *Annu. Rev. Fluid Mech.*, 37:457–487, 2005.

M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, volume 2625, pages 213–232. Springer, 2003.

J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05. ACM, 2005.

E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proc. 2001 ACM/IEEE Conf. on Supercomputing*, Supercomputing '01, pages 55–55. ACM, 2001.

V. K. Laxmikant and Z. Gengbin. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

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

A. Logg and G. N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2), 2010.

A. Logg, H. P. Langtangen, and X. Cai. Past and future perspectives on scientific software. In Aslak Tveito, Are Magnus Bruaset, and Olav Lysne, editors, *Simula Research Laboratory*, pages 321–362. Springer Berlin Heidelberg, 2010.

K. Madduri, S. Williams, S. Ethier, L. Oliker, J. Shalf, E. Strohmaier, and K. Yelicky. Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In *Proc. Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 48:1–48:12. ACM, 2009.

K. Madduri, E.-J. Im, K. Z. Ibrahim, S. Williams, S. Ethier, and L. Oliker. Gyrokinetic particle-in-cell optimization on emerging multi- and many-core platforms. *Parallel Comput.*, 37(9):501–520, 2011.

A. Marowka. Parallel computing on any desktop. *Commun. ACM*, 50(9): 74–78, September 2007.

W. Mattson and B. M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Comput. Phys. Commun.*, 119(2-3):135 – 148, 1999.

L. McInnes, B. Allan, R. Armstrong, S. Benson, D. Bernholdt, T. Dahlgren, L. Diachin, M. Krishnan, J. Kohl, J. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S. Parker, J. Ray, and S. Zhou. Parallel pde-based simulations using the common component architecture. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 327–381. Springer, 2006.

X. Mei, P. Decaudin, and B. G. Hu. Fast hydraulic erosion simulation and visualization on GPU. In *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, pages 47–56. IEEE, 2007.

M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0.* High-Performance Computing Center Stuttgart, September 2012.

J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, The Google Books Team, J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak, and E. L. Aiden. Quantitative analysis of culture using millions of digitized books. *Science*, 331(6014): 176–182, 2011.

F. Milde, M. Bergdorf, and P. Koumoutsakos. A hybrid model for three-dimensional simulations of sprouting angiogenesis. *Biophys. J.*, 95(7): 3146–3160, Oct 2008.

J. Misra and D. Gries. A constructive proof of Vizing's theorem. *Inform. Process. Lett.*, 41(3):131 – 133, 1992.

J. J. Monaghan. Extrapolating B splines for interpolation. *J. Comput. Phys.*, 60(2):253–262, 1985.

B. Moon and J. Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proc. IEEE Scalable High-Performance Computing Conference*, pages 176–183. IEEE, 1994.

C. L. Mueller, B. Baumgartner, G. Ofenbeck, B. Schrader, and I. F. Sbalzarini. pCMALib: a parallel fortran 90 library for the evolution

strategy with covariance matrix adaptation. In *Proc. 11th Conf. Genetic and evolutionary computation*, GECCO '09, pages 1411–1418. ACM, 2009.

D Nagle. Fortran interface to pthreads, 2005. URL `http://users.erols.com/dnagle/f2000.html`.

NVIDIA. *OpenCL Best Practices Guide*. NVIDIA, May 2010.

NVIDIA. *NVIDIA CUDA C Programming Guide*. NVIDIA, April 2012.

M. Olano and A. Lastra. A shading language on graphics hardware: the pixelflow shading system. In *Proc. 25th annual Conf. on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 159–168. ACM, 1998.

J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. 2008 IEEE Intl. Conf. on Cluster Computing*, pages 142 –151, October 2008.

POSIX. IEEE standard for information technology - portable operating system interface (POSIX). base definitions. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Base*, 2004.

M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. Sunos multi-thread architecture. In *Proc. Winter 1991 USENIX Conf.*, pages 65–80, 1991.

R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proc. 2009 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing*, pages 427 –436, feb. 2009.

J. T. Rasmussen, G.-H. Cottet, and J. H. Walther. A multiresolution remeshed Vortex-In-Cell algorithm using patches. *J. Comput. Phys.*, 230:6742–6755, 2011.

Sylvain Reboux, Birte Schrader, and Ivo F. Sbalzarini. A self-organizing Lagrangian particle method for adaptive-resolution advection–diffusion simulations. *J. Comput. Phys.*, 231:3623–3646, 2012.

J.V.W. Reynders, J.C. Cummings, M. Tholburn, P.J. Hinker, S.R. Atlas, S. Banerjee, M. Srikant, W.F. Humphrey, S.R. Karmesin, and K. Keahey. POOMA: a framework for scientific simulation on parallel architectures. In *Proc. First Intl. Workshop on High-Level Programming Models and Supportive Environments*, pages 41–49. IEEE Comput. Soc. Press, 1996.

D. Rossinelli and P. Koumoutsakos. Vortex methods for incompressible flow simulations on the GPU. *Visual Comput.*, 24(7):699–708, 2008.

D. Rossinelli, M. Bergdorf, G. H. Cottet, and P. Koumoutsakos. GPU accelerated simulations of bluff body flows using vortex particle methods. *J. Comput. Phys.*, 229(9):3316–3333, 2010.

D. Rossinelli, C. Conti, and P. Koumoutsakos. Mesh–particle interpolations on graphics processing units and multicore central processing units. *Phil. Trans. Roy. Soc. A*, 369(1944):2164, 2011.

D. W. I. Rouson, H. Adalsteinsson, and J. Xia. Design patterns for multiphysics modeling in fortran 2003 and C++. *Trans. Math. Softw*, 37(1): 3:1–3:30, January 2010a.

D. W. I. Rouson, J. Xia, and X. Xiaofeng. Object construction and destruction design patterns in fortran 2003. *Procedia Computer Science*, 1 (1):1495 – 1504, 2010b.

I. F. Sbalzarini. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Systems & Technol.*, 1(2):40–56, 2010.

I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems. *J. Comput. Phys.*, 215 (2):566–588, 2006a.

Ivo F. Sbalzarini, Arnold Hayer, Ari Helenius, and Petros Koumoutsakos. Simulations of (an)isotropic diffusion on curved biological surfaces. *Biophys. J.*, 90(3):878–885, 2006b.

H.Y. Schive, Y.C. Tsai, and T. Chiueh. GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *Astrophys. J. Suppl. Ser.*, 186:457, 2010.

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

J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, Cambridge, UK, 1999.

P. R. Shapiro, H. Martel, J. V. Villumsen, and J. M. Owen. Adaptive smoothed particle hydrodynamics, with application to cosmology: Methodology. *Astrophys. J. Suppl. Ser.*, 103:269, 1996.

N. Shavit and D. Touitou. Software transactional memory. *Distrib. Comput.*, 10:99–116, 1997.

T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale phase-field simulation for dendritic solidification on the tsubame 2.0 supercomputer. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 3:1–3:11, New York, NY, USA, 2011. ACM.

S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *Proc. 2006 Intl. Symp. on Software testing and analysis*, ISSTA '06, pages 157–168, New York, NY, USA, 2006. ACM.

F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proc. Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

R. Speck, L. Arnold, and P. Gibbon. Towards a petascale tree code: Scaling and efficiency of the PEPC library. *J. Comput. Sci.*, 2(2):138 – 143, 2011.

G. Stantchev, W. Dorland, and N. Gumerov. Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. *J. Par. Distrib. Comput.*, 68(10):1339–1349, 2008.

D. Stein and D. Shah Sunsoft. Implementing lightweight threads. In *Proc. 1992 USENIX Summer Conf.*, pages 1–9, 1992.

W. Stein and D. Joyner. Sage: system for algebra and geometry experimentation. *SIGSAM Bull.*, 39(2):61–64, June 2005.

G. Sutmann and V. Stegailov. Optimization of neighbor list techniques in liquid matter simulations. *J. Mol. Liq.*, 125:197–203, 2006.

W. C. Swope, H. C. Andersen, P. H. Berens, and K. B. Wilson. 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.*, 76(1):637–649, 1982.

K. H. Tsoi and W. Luk. Axel: a heterogeneous cluster with fpgas and gpus. In *Proc. 18th annual ACM/SIGDA Intl. Symp. on Field programmable gate arrays*, FPGA '10, pages 115–124. ACM, 2010.

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

J. S. Turner. Almost all k-colorable graphs are easy to color. *J. Algorithms*, 9(1):63 – 82, 1988.

UPC. UPC language specifications, v1.2. Technical report lbnl-59208, Lawrence Berkeley National Lab, 2005.

L. Verlet. Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys. Rev.*, 159(1):98–103, 1967.

V.G. Vizing. On an estimate of the chromatic class of a p-graph. *Diskret. Analiz*, 3(7):25–30, 1964.

J. H. Walther and I. F. Sbalzarini. Large-scale parallel discrete element simulations of granular flow. *Eng. Computations*, 26(6):688–697, 2009.

U. Welling and G. Germano. Efficiency of linked cell algorithms. *Comput. Phys. Commun.*, 182:611–615, 2011.

M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and Gibbon P. A massively parallel, multi-disciplinary barnes–hut tree code for extreme-scale n-body simulations. *Comp. Phys. Commun.*, 183(4):880 – 889, 2012.

Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Comput. Phys. Commun.*, 161(1-2):27 – 35, 2004.

Index

Publications

Refereed publications during PhD studies:

- **Omar Awile**, Ivo F. Sbalzarini, A pthreads wrapper for Fortran 2003, ACM Transactions on Mathematical Software, (submitted)
- Ferit Büyükkeçeci, **Omar Awile**, Ivo F. Sbalzarini, A portable OpenCL implementation of generic particle-mesh and mesh-particle interpolation in 2D and 3D, Parallel Computing, Available online 13 December 2012
- **Omar Awile**, Ferit Büyükkeçeci, Sylvain Reboux, Ivo F. Sbalzarini, Fast neighbor lists for adaptive-resolution particle simulations, Computer Physics Communications, 183(5), 2012, 1073-1081
- **Omar Awile**, Ömer Demirel, Ivo F. Sbalzarini, Toward an object-oriented core of the PPM library, Proc. ICNAAM, International Conference, 1313-1316, 2010
- **Omar Awile**, Anita Krisko, Ivo F. Sbalzarini, Bojan Zagrovic, Intrinsically Disordered Regions May Lower the Hydration Free Energy in Proteins: A Case Study of Nudix Hydrolase in the Bacterium Deinococcus radiodurans, PLoS Computational Biology 6(7), 2010

Earlier refereed publications:

- **Omar Awile**, Laurent Balmelli, Fumihiko Kitayama, Masayuki Numao, Method and apparatus for using design specifications and measurements on manufactured products in conceptual design models, JP820060617 (patent)

Curriculum Vitae

Name:	Omar Awile
Born:	December 4th, 1981
Citizen of:	Switzerland
May 2001	Matura Typus C – natural sciences and math
2001 - 2007	Diploma and Master studies in Computer Science at ETH Zurich, Switzerland, with major in computational science and minor in astrophysics
January 2008	M.Sc. ETH in Computer Science
2008 - 2012	PhD studies under the supervision of Prof. Dr. Ivo F. Sbalzarini at the MOSAIC Group, Institute of Theoretical Computer Science, ETH Zurich