Abstractions and Middleware for Petascale Computing and Beyond

Ivo F. Sbalzarini*
Institute of Theoretical Computer Science and Swiss Institute of Bioinformatics,
ETH Zurich, CH-8092 Zurich, Switzerland

ABSTRACT
As high-performance computing moves to the petascale and beyond, a number of algorithmic and software challenges need to be addressed. We review the main performance-limiting factors in today's high-performance computing software and outline a possible new programming paradigm to address them. The proposed paradigm is based on abstract parallel data structures and operations that encapsulate much of the complexity of an application, but still make communication overhead explicit. We argue that all numerical simulations can be formulated in terms of the presented abstractions, which thus define an abstract semantic specification language for parallel numerical simulations. Simulations defined in this language can automatically be translated to source code containing the appropriate calls to a middleware that implements the underlying abstractions. We outline the structure and functionality of such a middleware and demonstrate its feasibility on the example of the parallel particle-mesh library (PPM).

Keywords: Parallel Processing Systems, Middleware, Scientific Computing, High-Performance Computing, Parallel Programming, Abstractions.

INTRODUCTION
Numerical simulations are well established as the third pillar of science, alongside theory and experiments. As numerical methods become more powerful, and more data and knowledge become available about complex real-world systems, the simulations become increasingly elaborate. The availability of parallel high-performance computing (HPC) systems has enabled simulations with unprecedented numbers of degrees of freedom. The corresponding simulation codes are mostly tightly coupled, which means that the different processors of the HPC machine need to exchange data (communicate) several times while solving a problem, and not only at the beginning and the end of the simulation. Minimizing the communication overhead is thus key to parallel efficiency. In this article we propose a novel abstraction layer that provides the proper level of granularity to address some of the software challenges the field is facing as we move beyond petascale systems. We focus on tightly coupled simulations as they occur in classical HPC applications in, e.g., material science, fluid dynamics, astrophysics, or computational chemistry, and in emerging user fields such as biology, finance, or social science.

Despite the proliferation of HPC applications to new areas of science, programming and using HPC machines is becoming increasingly difficult. As the performance of single processors has stopped increasing, speedup can only be achieved through parallelism. There is no more "free

speedup" for legacy codes. In addition, memory capacity is growing faster than memory bandwidth (aggravated by the fact that several processor cores are sharing a memory bus), such that accessing memory becomes increasingly expensive compared to compute operations. Presently, it takes about one to two orders of magnitude longer to access the main memory than to perform a floating-point multiplication[1]. In GPUs and other emerging heterogeneous cores this ratio is even higher. Efficient codes should thus minimize memory access counts rather than operation counts. This presents challenges to both the traditional HPC user fields as well as the emerging fields. In traditional fields, well-tested and efficient codes have existed for several decades. These codes are usually large and of limited parallel scalability. They have to be ported to heterogeneous multi-core HPC systems or re-written altogether. For emerging users in biology or social science, there is a high entry hurdle into HPC since parallel programming is notoriously difficult, requires experience, and takes a long time. The predominantly used message-passing paradigm resembles "communication assembly language" with every single point-to-point communication explicitly coded by the programmer. Together, these developments cause several growing gaps in parallel HPC:

1. *the performance gap:* the actual sustained performance of scientific simulation codes is a decreasing fraction of the theoretical peak-performance of the hardware,
2. *the knowledge gap:* efficient use of HPC resources requires more and more specialized knowledge and is restricted to a smaller and smaller community,
3. *the reliability gap:* as machines contain more and more processor cores, the mean-time between failure drops below the typical runtime of a simulation, and
4. *the data gap:* storing, accessing, and analyzing the peta-bytes of data generated by large simulations or experiments (such as those in astronomy or particle physics) becomes increasingly difficult.

Since the advent of multi-core CPUs, some of these gaps even exist on the single-processor level. Portable, generic scientific software libraries such as GSL or "numerical recipes" are about one order of magnitude slower than vendor-provided, machine-specific libraries such as Intel's IPP/MKL. This is mainly due to the fact that the latter are explicitly optimized and often contain hand-tuned assembly language code for the performance-critical sections, which, however, limits their portability. Moreover, as memory is becoming more expensive than processor cores, the available memory per core decreases, causing bottlenecks due to memory contention when independent heavy-weight processes (such as MPI processes) are running on the different cores (Sbalzarini, Walther, Bergdorf, et al., 2006). Simulations are increasingly memory limited and often use only as many cores per processor as are needed to saturate the memory bandwidth, disabling the rest of the cores or reducing their clock frequency (Gruber & Keller, 2009). In order to maximize the number of usable cores, multi-core parallelism thus has to account for coordinated memory access and be able to profit from shared caches.

In high-end HPC systems, the situation is proportionally worse. Current petascale systems contain around 130 000 processor cores of potentially different architectures, such as the mixture of Opteron and Cell processors used in the Roadrunner system at Los Alamos. These cores are connected by a hierarchy of communication layers of different speeds and operate on data that are stored in memory systems with at least 4 levels of hierarchy (registers, L1 cache, L2 cache, main memory). The different cores on the same processor (chip) have a very fast interconnect

and additionally benefit from shared L2 cache and shared main memory. The different chips in the same compute node are connected on the next level of the hierarchy and may still share the main memory, but no cache. The various compute nodes of the machine are finally connected by a network of a certain, again potentially hierarchical, topology. This hardware complexity is expected to increase even more as we move toward exaflop systems. Exaflop systems are expected around 2020 (J. Dongarra, ISC, Dresden, 2008) and could well have tens of millions of heterogeneous cores.

Addressing the above-mentioned gaps in these HPC systems requires novel programming paradigms and novel algorithms (Asanovic et al., 2006), the availability of which will be the decisive factor determining the utility of future supercomputers. These programming paradigms should emphasize the hierarchical organization and access cost of the memory and the interconnect, but should be independent of the number of processing elements. In addition, they should be application-independent, portable, and easy to learn and use. One strategy is to introduce an additional layer of abstraction between the computer system and the programmer. This layer should provide a transparent view of the machine, independent of the number and architecture of the processors, such that it will not be necessary for the user to write "communication assembly language". One could then implement scalable, memory-aware algorithms on top of this abstraction layer. While this idea has already proven useful in areas such as numerical linear algebra (ScaLAPACK, PETSc, ...), it remains unextended to other simulation paradigms, including agent-based models, particle methods, tree codes, etc.

In this paper, we propose a layer of abstract parallel data types and operations. Their intermediate granularity makes the parallel semantics of the program explicit while still hiding the individual communication operations. Each abstraction defines an encapsulated functionality that can be efficiently implemented as a module of a middleware or a run-time system, thus effectively hiding the hardware complexity from the scientific programmer and reducing the knowledge gap. As a result, the parallel efficiency of scientific simulations based on middleware often surpasses that of hand-parallelized codes (Sbalzarini, Walther, Bergdorf, et al., 2006; Voronenko, 2008). In addition, the abstractions presented here naturally define a language in which the semantics of a parallel simulation program can be expressed. This enables automatic generation of fine-tuned code and ensures portability across machine architectures. Ultimately, we envision a paradigm where the simulation program is specified in an abstract language, which is then automatically translated to source code containing the appropriate calls to the run-time system that implements the abstractions. These implementations can then afford to be platform-specific and auto-tuned. We believe that such a programming paradigm would increase the performance, efficiency, portability, and reliability of large-scale parallel simulations, reduce code development time, and enable a larger community of scientists to benefit from the power of supercomputing.

ABSTRACTIONS AND THEIR IMPLEMENTATION

Data and operation abstractions are a powerful concept to encapsulate complexity at various levels and to address the challenges outlined. Their purpose is to introduce transparent layers between the application programmer and the hardware of the machine in order to achieve code portability and ease of use. The price one pays for this is a loss of flexibility and generality. Using an abstraction, such as the message passing interface (MPI), necessarily restricts the types

of operations one is able to perform to those defined in the abstraction layer. This loss of flexibility is controlled by the granularity (level of detail) of the abstraction layer. The more coarse-grained an abstraction, the less flexible it is, but the easier to use. For scientific computing, it is critically important to develop abstractions at the right granularity. We argue that they do not necessarily diminish the performance of application codes. In practice, simulations that are implemented on top of abstractions are often even more efficient than hand-written, machine-specific codes (Sbalzarini, Walther, Bergdorf, et al., 2006; Voronenko, 2008). This can be explained by the abstraction layer reducing the knowledge gap. By depriving the user of some of the flexibility and complexity of the underlying system, it becomes much harder for the average programmer to write inefficient code.

We propose an abstraction layer that is based on simple parallel data structures and operations that encapsulate computation and communication separately and allow semantic specification of large-scale distributed HPC applications.

Abstractions

Scientific simulations involve discretizing space and time. We posit that most discretizations on parallel computers can be described in terms of topologies, particles, meshes, connections, interactions, and mappings. We define a topology as a (not necessarily disjoint) decomposition of the computational domain into sub-domains and the assignment of these sub-domains onto processors. Particles are computational elements that are characterized by their position in the computational domain and an arbitrary number of associated properties or methods. Meshes are structured (Cartesian) computational grids, and connections are linker objects between particles that allow grouping them. Interactions are multi-threaded compute operations that can be executed on a set of particles, mesh nodes, and/or connections. Lastly, mappings are communication operations that move particle data, mesh data, or connection data between processors. We define a processor as the collection of all cores that operate on the same memory address space. Processors support multi-threading and can possibly be distributed across several chips.

The abstractions defined here provide an intermediate level of granularity. They are coarse-grained enough to be easy to use and to hide individual point-to-point communications, and they are fine-grained enough such that all of Colella's original 7 dwarfs (Colella, 2004) can be composed from them. These dwarfs are self-contained numerical kernels that constitute the building blocks of today's numerical simulations. The original 7 dwarfs are: dense linear algebra (e.g. BLAS), sparse linear algebra (e.g. OSKI), spectral methods (e.g. FFT), N-body methods (e.g. particle methods or fast multipoles), structured grid methods (e.g. finite differences or lattice Boltzmann), unstructured grid methods (e.g. finite elements or finite volumes), and MapReduce (e.g. Monte Carlo). Six additional dwarfs have been added to also cover application areas such as databases, machine learning, computer graphics, or embedded computing. We now discuss the proposed abstractions in more detail.

Topologies are created by applying a domain decomposition algorithm and a load balancing (processor assignment) algorithm to the computational domain and, possibly, the initial distribution of particles, connections, and/or meshes. They establish a correspondence between the computational elements (particles, mesh nodes, connections) and individual processors of the

machine. Topologies are dynamically created and destroyed at run-time and several topologies can exist concurrently in order to, e.g., allow for optimal load balance when particles and mesh nodes are unequally distributed. Each topology comprises a set of cuboidal sub-domains that have halo layers on exactly half of their surfaces (Sbalzarini, Walther, Bergdorf, et al., 2006) in order to allow for symmetric communication with neighboring sub-domains and local evaluation of finite-cutoff interactions (see below). Symmetric communication significantly reduces the memory and communication overhead. Typically, the number of sub-domains is much larger than the number of available processors in order to achieve sufficient granularity for load balancing and re-balancing (see Fig. 1). Assigning more than one sub-domain per processor comes at no additional cost, as outlined below. Since several topologies may be defined, this allows for implementation of parallel tree codes or Barnes-Hut algorithms using a hierarchy of topologies that correspond to the different levels of the tree (Sbalzarini, Walther, Polasek, et al., 2006).

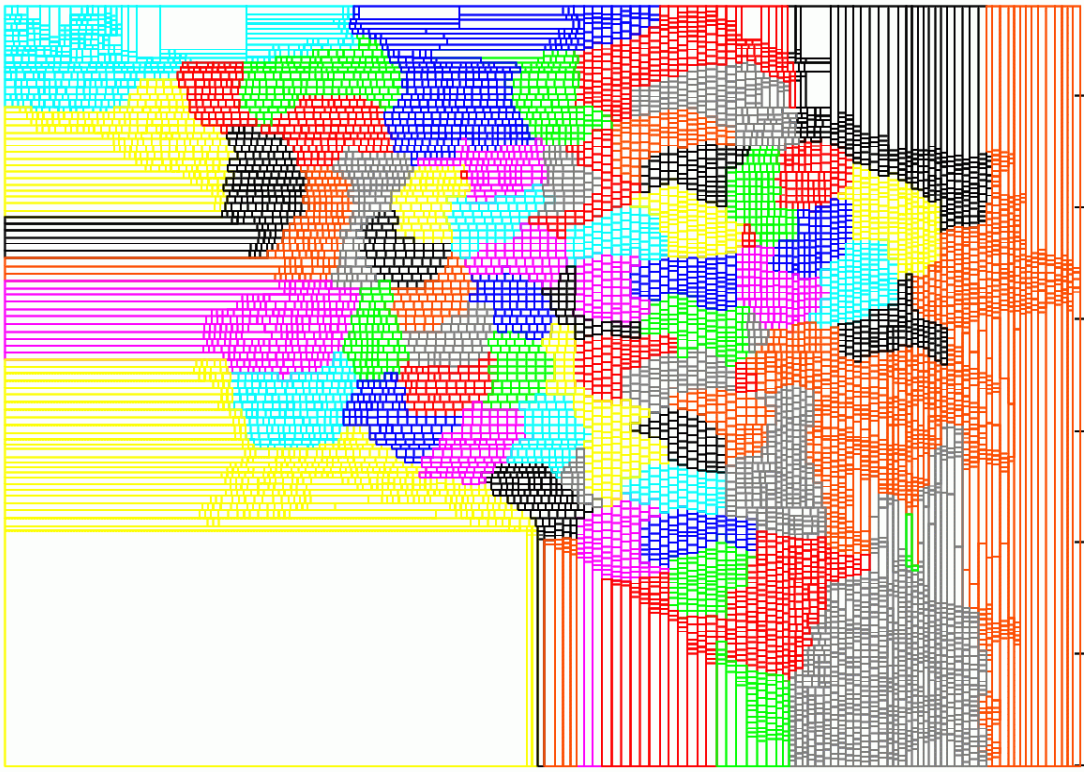

Figure 1. Example of a topology. The computational domain (bounding rectangle) is subdivided into 9311 cuboidal sub-domains (small rectangles) that are distributed onto 64 processors (patches of different gray level). Each processor is assigned a connected set of multiple sub-domains in order to provide sufficient granularity for load-balancing and shared-memory parallelism within each processor. The inter-processor boundaries are optimized to minimize the overall communication volume.

Mappings encapsulate the communication between processors. There are four basic mapping types that the user can invoke: global, local, ghost-get, and ghost-put. A global mapping involves a potentially all-to-all communication between the processors and can be used to distribute data

according to a certain topology or switch from one topology to another. In a local mapping, processors only communicate with their neighbors, defined as those other processors that are assigned sub-domains adjacent to any of the sub-domains of the current processor. Local mappings can be used when particles have moved across processor boundaries or to adjust load balance. Two ghost mappings are provided to operate on the halo layers. The ghost-get mapping populates the halo layers of all sub-domains with copies ("ghosts") of the computational elements from the neighboring processors so they are available for the local evaluation of interactions. At intra-processor sub-domain boundaries, no additional memory or communication is required since the ghost elements are, in this case, identical to the corresponding real elements, and they are in the same memory address space. The ghost-put mapping is available to send back ghost contributions from the halo layer of the current processor to the corresponding real elements on neighboring processors. This allows computing symmetric interactions that involve ghosts and performing particle–to–mesh and mesh–to–particle interpolations locally per sub-domain. The mapping abstractions internally keep track of the correspondence between ghost elements and real elements such that no bookkeeping is required from the user program. Mappings also internally determine a communication schedule such that a near-minimal number of communication operations are required and no conflicts occur. For global mappings this can be done using rings or trees. For local mappings, graph-based methods can be used, such as minimum edge coloring (or rather: an approximate solution to it). Besides their correctness, we can also require mappings to be tolerant to single-processor hardware failures. With appropriate support from the run-time system (see below), this can be implemented entirely inside the mapping routines since they encapsulate all communication.

Meshes are defined by their resolution. Each topology can have several meshes associated with it. This allows for implementation of multi-grid algorithms or adaptive mesh refinement methods.

Particles are connectivity-free computational elements that are defined by their position in the computational space and an arbitrary number of additional properties. These properties can be methods, scalar data, or higher-dimensional data of any type.

Connections can be used to link particles in order to define, e.g., molecular bonds, unstructured grids, triangulations, or graphs. They constrain the topology such that connected particles are preferably assigned to the same processor.

Interactions encapsulate local compute operations on a set of particles, mesh nodes, and/or connections within a sub-domain. Interactions therefore do not involve any communication, but can still be distributed over multiple cores of the same processor by multi-threading. In most applications, interactions amount to evaluating pair-wise kernels between elements within a certain cutoff radius. Computing particle–particle interactions allows implementing particle-based simulations or agent-based models. Mesh–mesh interactions account for purely mesh-based operations such as finite differences or FFTs. Finally, particle–mesh interactions are available to implement interpolation schemes, remeshing, or particle-in-cell methods by specifying pair-wise interaction kernels between mesh nodes and particles. Since all interactions are local to a sub-domain, they might constrain the types of topologies that can be used. FFTs, for example, require topologies with pencil domain decomposition, where all mesh nodes along

at least one spatial direction are in the same memory space. Such constraints are, however, easily accounted for since several topologies can exist concurrently. When computing interactions between particles without connectivity information, the interaction routine can transparently build and use internal cell lists (Hockney & Eastwood, 1988) or Verlet lists (Verlet, 1967) for fast neighbor search within the given cutoff radius.

In summary, we define the following data abstractions:

- topology(sub-domains, processor assignments)
- particle(position, properties)
- mesh(resolution)
- connection(particle1, particle2)

Any simulation is then expressed in terms of sets of these abstractions. It thus comprises a set (denoted by curly braces "{}") of topologies {topology}, a set of particles {particle}, a set of meshes {mesh}, and a set of connections {connection}. Several, but not all, of these sets may be empty if the corresponding data types are not used. An empty set of topologies, for example, provides for sequential single-processor simulations. On these abstract parallel data types, the following operations are defined:

- mapping(type, {particle} or mesh or {connection}, topology)
- interaction({particle} or {connection} or {mesh} or {particle, mesh}, cutoff)
- create topology(computational domain, {particle, mesh, connection}, domain decomposition algorithm, processor assignment algorithm)

By choosing different domain decomposition and processor assignment algorithms, one can include problem-specific prior knowledge.

When using these abstractions, models from all four realms can be simulated: continuous-deterministic, continuous-stochastic, discrete-deterministic, and discrete-stochastic. The classical numerical simulation methods for these four model types include mesh-based and mesh-free discretization schemes for PDEs, sampling-based schemes such as Monte Carlo, interacting particle systems, agent-based simulations, molecular dynamics, discrete element methods, and discrete-event simulations. All of these methods can be phrased in terms of particles, meshes, connections, interactions, mappings, and topologies.

Mappings are pure communication operations and interactions are pure compute operations. This makes it possible to assess the communication overhead of a simulation already in its abstract specification since computation and communication are not interleaved. The abstractions naturally define a language in which parallel simulations can be specified. As an example, a simple parallel molecular dynamics simulation could be specified as follows:

```
read or create {particle} (atoms), {connection} (bonds)
t1 = create topology(computational domain, {particle, connection}, ROB, minEdgeCut)
mapping(global, {particle}, t1)
mapping(global, {connection}, t1)
for time-step = 1,..., T do
```

```
    mapping(ghost-get, {particle}, t1)
    mapping(ghost-get, {connection}, t1)
    interaction({particle}, cutoff) # non-bonded interactions
    interaction({connection}, cutoff) # bond interactions
    update the positions and properties in {particle}
    mapping(partial, {particle})
    mapping(partial, {connection})
end
```

In this example, the topology t1 is created based on the possibly inhomogeneous distribution of atoms and bonds using a recursive orthogonal bisection (ROB) domain decomposition and a processor assignment that minimizes the edge cut in the communication graph. If the molecular dynamics simulation also includes long-range interactions such as electrostatics, the cutoff becomes too large for direct particle–particle interactions. In this case, the simulation is more efficient when solving the corresponding Poisson equation on a mesh using FFTs or multi-grid methods. In order to use FFTs in three dimensions, we define a mesh of resolution h and three additional topologies before the time loop:

```
m1 = mesh(h)
t2 = create topology(computational domain, {m1}, x-pencil, minEdgeCut)
t3 = create topology(computational domain, {m1}, y-pencil, minEdgeCut)
t4 = create topology(computational domain, {m1}, z-pencil, minEdgeCut)
```

Each of these topologies uses a load-balanced pencil decomposition of the mesh m1 where the sub-domains extend throughout the entire computational domain in one direction. Inside the time loop (before updating the particle positions and properties), an FFT can then be computed in parallel by:

```
    mapping(global, m1, t2)
    interaction({m1}, Lx) # compute the FFT in x direction
    mapping(global, m1, t3)
    interaction({m1}, Ly) # compute the FFT in y direction
    mapping(global, m1, t4)
    interaction({m1}, Lz) # compute the FFT in z direction
    mapping(global, m1, t1)
```

Using three additional interactions to interpolate {particle} to m1, solve the Poisson equation in frequency space (cutoff = 0), and interpolate the result back onto the particles completes the solver. The global mappings required by the FFT can be avoided by solving the Poisson equation using a multi-grid method. Then, only the topology t1 is needed with several meshes of different resolution defined on it: m1 = mesh(h), m2 = mesh(h/2), ...

Implementation

The abstractions defined above can directly be implemented in a run-time system or middleware as platform-specifically tuned software modules. Depending on the granularity of the underlying abstractions, different levels of encapsulation can be realized. On the lowest level, libraries that

implement the MPI standard (Message Passing Interface Forum, 2008) provide abstractions for point-to-point and collective communication operations by encapsulating the communication stacks of the operating system. Adaptive MPI (AMPI) supports dynamic load balancing and multi-threading at this level of the run-time system (Huang, Lawlor, & Kale, 2004). On an intermediate level, more coarse-grained abstractions can be implemented. Examples include the ASTRID programming environment that transparently parallelizes structured-grid finite-element and finite-volume simulations (Bonomi et al., 1989), or the parallel scalable I/O libraries PASSION (Thakur, Bordawekar, Choudhary, Ponnusamy, & Singh, 1994) and ROMIO (Thakur, Gropp, & Lusk, 1999) (now part of the MPI 2.1 standard). On the highest level of abstraction, entire numerical methods can be encapsulated using the abstraction of dwarfs. This includes libraries such as FFTW, ScaLAPACK, PETSc, or the PARTI run-time library for Monte Carlo simulations (Moon & Saltz, 1994).

The abstractions outlined in this paper are implemented on the intermediate level. Support for topology creation entails dynamic partitioning of the data and operations onto the available processors, and support for mappings can include communication scheduling and mechanisms of fault-tolerance. This is inspired by libraries such as the parallel utilities library PUL (Chapple & Clarke, 1994), which provides domain decomposition methods, data communication, and parallel file I/O for purely mesh-based simulations.

These implementations can afford to be machine-specific since portability is ensured on the level of the abstract specification language. Optimized implementations of the abstractions can thus fully benefit from language and compiler support, as well as from auto-tuning code generators.

Language support includes programming languages that are aware of (and provide some control over) the hardware memory hierarchy, and that can make data dependencies explicit in order to allow streaming or SIMD vectorization by the compiler. An example of the former is the Sequoia language (Fatahalian et al., 2006), which exposes the memory hierarchy and allows controlled memory-aware programming. Examples for the latter include array programming languages such as APL, HPF, and Co-Array Fortran (included in Fortran 2008). An interesting early example is the Vectoral programming language that was developed by Alan Wray in 1978 to provide language-level SIMD parallelism on the Illiac IV computer (Wray, 1988). In addition, some languages have a notion of MIMD/SPMD parallelism. Meta-languages such as Linda (Carriero & Gelernter, 1989), however, suffer from a loss of computational efficiency and portability (Turkiyyah, Reed, & Yang, 1996). This is avoided in compiled parallel languages such as Unified Parallel C (UPC Consortium, 2005) or the object-oriented parallel language Charm++ (Kale, Bohm, Mendes, Wilmarth, & Zheng, 2007).

Compiler support exploits the data distribution and dependencies made explicit by the programming language. Currently, however, it is almost impossible for compilers to automatically extract the parallel semantics of a program and use them for high-level optimizations. A promising approach for message-passing codes, where every communication operation is explicitly specified, is the generalization of data-flow graphs to parallel data-flow graphs (Bronevetsky, 2009).

Automatic code generators such as SPIRAL (Püschel et al., 2005; Voronenko, 2008) can be used to tune the middleware implementations of the abstractions to specific hardware platforms. Such auto-tuned code often outperforms hand-written code (Voronenko, 2008). This is mainly because elegant, human-readable code is not always the fastest code possible. Code generators that can handle vectorization (SIMD) or shared-memory parallelism (MIMD) could be used to implement the individual abstractions on multi-core processors. This, however, relies on the availability of accurate performance prediction tools or models that allow the code generator to evaluate different options and choose the one that is best suited for a specific machine architecture. Accurate parameterization of the effective hardware performance and of the resource needs of the various code optimization options can help choose the right one, abstracting from the very details of the hardware (Gruber & Keller, 2009). Given the time and space complexity of an algorithm and several measured run-time parameters, such models predict the expected execution time of the algorithm on a given machine. One example of a performance prediction model is the extended Γ–κ model. In this model, the parameter Γ quantifies how communication-limited the distributed-memory part of an applications is and κ distinguishes memory-limited multi-threaded application parts from CPU-time-limited ones (Gruber & Keller, 2009). This allows accurate prediction of the parallel scalability of an application, and an informed choice of the algorithms and hardware resources to be used. Application of such models is, however, currently hindered by unpredictable run-time influences from the hardware: caches, look-ahead logics, network routing, and dynamic over-clocking (e.g. in Intel's Nehalem architecture).

Once the implementations are available in a middleware or run-time system, it is conceivable that the semantic description of a parallel numerical simulation, formulated in terms of abstractions, is directly translated by a "simulation compiler" to source code containing the appropriate calls to the run-time system or middleware that implements these abstractions.

Run-Time System Structure

We outline the structure and functionality of a run-time system that supports implementations of the abstractions defined above. In order to implement interaction operations with multiple levels of parallelism (mixed multi-processing/multi-threading), as well as self-optimizing and fault-tolerant mappings, the run-time system has to provide:

1. a multi-level parallelization layer that is aware of the communication/memory hierarchy,
2. a hardware-aware communication scheduler and dynamic load balancer, and
3. a fault-tolerant communication layer.

Multi-level parallelism is frequently implemented using nested OpenMP or MPI-2 threads. In the context of the presented abstractions, shared-memory threads are used within individual interaction and mapping modules, whereas each processor runs a separate distributed-memory message-passing process. The parallelization layer can also internally probe, at run-time, the network topology and the memory hierarchy of the machine and make this information available to the communication scheduler and the load balancer.

The *hardware-aware communication scheduler and load balancer* uses this information to optimize the communication schedule through graph algorithms. Approximately solving the

minimum edge-coloring problem (Vizing, 1964) on the graph of required (from the application software) communications provides near-optimal schedules with a +1 error bound (Djordjevic & Tosic, 1996; Durand, Jain, & Tseytlin, 2003). Bandwidth, latency, and distance between the processors (according to the network's hardware topology) can be accounted for by weights on the edges of the graph (Bhatele & Kale, 2008). The load balancer can use the measured run-time parameters in a performance prediction model in order to decide which part of an application should run on cores of which architecture, or to dynamically migrate processes to better-suited parts of the machine (Gruber & Keller, 2009). Among cores of the same architecture, processor assignment can be based on graph partitioning algorithms (Karypis & Kumar, 1998a, 1998b). There, each node of the graph corresponds to a work package (such as a sub-domain) and each edge corresponds to a required communication between work packages. Every node is attributed a weight reflecting the computational cost of that work package, and every edge has a weight that represents the communication volume. The nodes are then partitioned onto the processors of the machine such that the variance of computational costs across processors and the total edge-cut are minimized. In this step, the relative speeds of the processors, their architecture (in a heterogeneous machine), and their proximity relations in the machine interconnect can be accounted for by using, for example, parameterized descriptions (Gruber & Keller, 2009). This ensures that every work package runs in its "optimal" hardware environment. In the framework of the proposed abstractions, this can be done at the level of topologies for the multi-processing parts of an application, and at the level of interactions for the multi-threaded parts.

A *fault-tolerant communication layer* becomes important when machines grow large enough such that the mean time between hardware failures is less than the average run-time of a simulation. In a machine with 1 million processors, at least one node is expected to fail every 30 minutes. While the mean-time between failures can be increased by about a factor of 2 if all unused cores and machine parts are dynamically powered down and the clock frequency is adjusted to the application needs (Gruber & Keller, 2009), fault tolerance remains an issue. Mostly, it is addressed by checkpointing (Elnozahy, Alvisi, Wang, & Johnson, 2002), which means that the data of the simulation are periodically copied from the main memory of the computer to hard disks. If one of the nodes fails, the whole simulation program halts and restarts from the most recent checkpoint by recovering the data from disk. While global checkpointing does not scale to large systems, local recovery protocols have an overhead that is independent of the number of processors. In the simplest local scheme, checkpoint files are written to the scratch disks of the individual compute nodes with a copy onto the neighboring node. The constant overhead of local checkpointing can be reduced if the communication layer of the run-time system includes mechanisms of fault tolerance. This has been implemented in the Converse run-time system (Kale, Bhandarkar, Jagathesan, Krishnan, & Yelon, 1996) for Charm++ and AMPI, and the fault-tolerant HARNESS run-time system (Beck et al., 1999; Angskun, Fagg, Bosilca, Pjesivac-Grbovic, & Dongarra, 2006). Based on such run-time systems, fault-tolerant message-passing libraries have been implemented. Examples include the fault-tolerant FT-MPI (Fagg et al., 2004), which is implemented based on HARNESS, and AMPI (Huang, 2004) as implemented based on Converse. Other fault-tolerance techniques include direct (possibly asynchronous) use of communication sockets, message logging (Elnozahy et al., 2002), and in-memory data redundancy (Zheng, Shi, & Kale, 2004).

A Feasibility Study: The Parallel Particle-Mesh Library (PPM)

The PPM library (Sbalzarini, Walther, Bergdorf, et al., 2006; Sbalzarini, Walther, Polasek, et al., 2006) was a first attempt to implement the abstractions outlined in this paper in a transparent, portable middleware. The overall goal is to provide a processor-independent, data-transparent programming model for distributed-memory computers and to completely hide MPI from the application programmer. In PPM, this has been done for tightly coupled parallel numerical simulations that are formulated in the framework of hybrid particle-mesh methods. PPM is a middleware in the sense described here, introducing an additional, transparent layer between infrastructure libraries such as MPI, Metis (Karypis & Kumar, 1998a, 1998b), and FFTW and the user's simulation programs. This is schematically depicted in Fig. 2 and has made it possible to implement scalable parallel simulations without having to know MPI, which immediately renders supercomputing accessible to a larger user community. Since PPM limits the flexibility of the application programmer to hybrid particle-mesh methods described by the abstractions defined above, even inexperienced programmers can implement simulations that outperform state-of-the-art hand-parallelized codes (see (Sbalzarini, Walther, Bergdorf, et al., 2006) for examples).

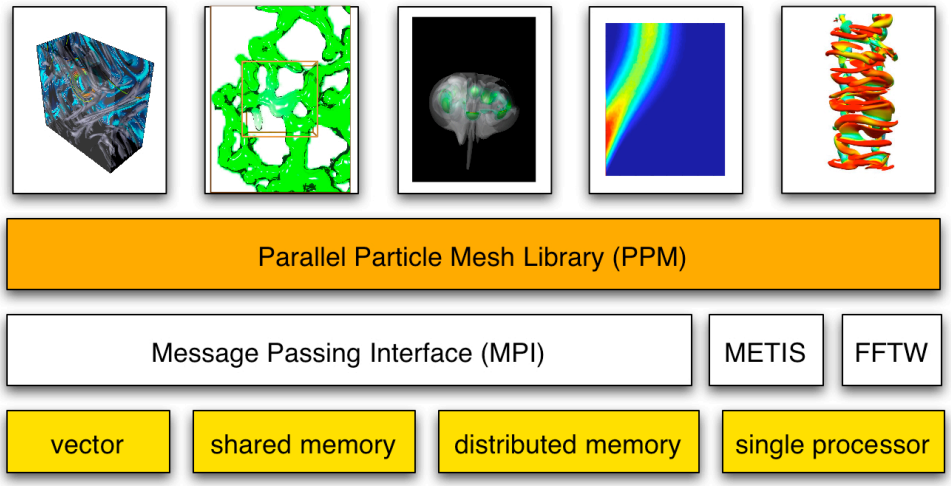

Figure 2. The parallel particle mesh (PPM) library is a transparent middleware layer between system-specific low-level libraries (MPI, METIS, FFTW) and the user's parallel simulation programs. It provides a run-time environment that implements abstract parallel data structures and operations in encapsulated software modules. The simulation codes are then specified in terms of these abstractions (see main text for details).

The PPM library is independent of specific applications. The library design goals include ease of use, efficient parallel scalability in both CPU time and memory requirements, and SIMD vectorization of all major loops. PPM implements encapsulated modules for the abstractions defined above and provides the adaptive domain decomposition, dynamic load balancing, and communication scheduling methods required to create topologies and mappings based on a specific data (particle or mesh) distribution at run time. Communication scheduling is done using an approximate solution of the minimum edge-coloring problem (Vizing, 1964). The SAR heuristic (Moon & Saltz, 1994) is used to decide when to re-decompose a problem in order to achieve a good trade-off between the cost of domain decomposition and arising load imbalance

from particle motion. All of this is done transparently in the background, without participation of the user program. For further details about the PPM library architecture, the reader is referred to the original publication (Sbalzarini, Walther, Bergdorf, et al., 2006).

Supplementing the library core functionalities, PPM also includes frequently used numerical solvers, such as multi-grid and FFT Poisson solvers, multi-stage ODE integrators, fast multipole methods (Greengard & Gropp, 1990), and fast marching methods (Sethian, 1999) as well as group marching methods (Kim, 2001) for level sets. These numerical modules can be interpreted as dwarfs that are implemented using the data and operation abstractions provided by the PPM library core. In addition, PPM also provides transparent parallel file I/O.

The efficiency, scalability, and ease of use of PPM have been demonstrated in a number of past applications as summarized in Table 1. In all of these past applications, the PPM-based simulation codes outperformed the corresponding state-of-the-art hand-written codes in wall-clock time, parallel efficiency, or both (Sbalzarini, Walther, Bergdorf, et al., 2006). Also, it is interesting to note that the PPM codes were implemented in short time and without using a simulation compiler. The molecular dynamics code was, for example, implemented in less than 3 months by a first-year PhD student with no prior experience in parallel programming or in using PPM. This implementation outperformed (by more than a factor of 2) an existing handcrafted molecular dynamics code that was developed over 6 years by the same group. It completed one time step of an 8 million-atom simulation in 0.25 seconds at 63% parallel efficiency on 256 processors (see Table 1). The discrete element simulation (Walther & Sbalzarini, 2009) was implemented in less than 2 days by two of the PPM developers who are acquainted with the middleware. Discrete element methods (DEM) are particularly hard to parallelize due to the dynamically changing contact lists and the need to integrate contact deformations over time. The PPM-based simulation code sustained 40% parallel efficiency on 192 processors of a standard Ethernet-Linux cluster (Table 1). The simulation used 122 million fully resolved visco-elastic spheres, constituting the largest DEM simulation done thus far.

Table 1. Weak scaling results from past applications using the PPM library.

Application	***# particles***	***# proc.***	***Machine***	***Parallel efficiency***	***Reference***
smoothed particle hydrodynamics of compressible flow	$268 \cdot 10^6$	128	Cray XT4	91%	(Sbalzarini, Walther, Bergdorf, et al., 2006)
vortex method for incompressible flow (using the PPM-based multi-grid Poisson solver)	$268 \cdot 10^6$	512	IBM p690	76%	(Sbalzarini, Walther, Bergdorf, et al., 2006)
particle strength exchange for diffusion in a complex biological geometry (Sbalzarini, Mezzacasa, Helenius, & Koumoutsakos, 2005; Sbalzarini, Hayer, Helenius, & Koumoutsakos, 2006)	10^9	242	IBM p690	84%	(Sbalzarini, Walther, Bergdorf, et al., 2006)
molecular dynamics of Lennard-Jones atoms	$8 \cdot 10^6$	256	IBM p690	63% (0.25 s/time step)	unpublished

discrete element method for granular flow	$122 \cdot 10^6$	192	AMD Opteron Linux cluster, Gigabit Ethernet	40%	(Walther & Sbalzarini, 2009)
vortex method for incompressible turbulent flow	$6.5 \cdot 10^9$	16 384	IBM BG/L	62%	(Chatelain et al., 2008a; Chatelain et al., 2008b)

The remeshed smoothed particle hydrodynamics (SPH) simulation in Table 1 used 268 million particles transparently distributed onto 128 processors of the Cray XT4 computer at the Swiss National Supercomputing Centre (CSCS) to simulate the fluid dynamics of a compressible vortex ring (Sbalzarini, Walther, Bergdorf, et al., 2006). The simulation sustained 91% parallel efficiency (weak scaling) on 128 processors. This compares well to the 85% efficiency achieved by the GADGET SPH code of the Max-Planck Institute for Astrophysics on 32 processors. The same PPM-based SPH code was later also used to simulate a self-propelled swimmer with an immersed-boundary SPH. Using 13 million particles, the code took 70 seconds per time step on a Linux cluster with 32 AMD Opteron 2.2 GHz processors (Hieber & Koumoutsakos, 2008).

Vortex methods are hybrid particle-mesh methods to simulate incompressible fluids. In contrast to SPH, the incompressibility constraint requires the solution of a Poisson equation at every time step. This introduces long-range interactions that need global mappings, hence increasing the communication overhead of the parallel simulation. Together with the larger number of processors needed, this leads to lower parallel efficiencies compared to SPH simulations. The PPM-based vortex method code used in Table 1 nevertheless sustained 76% parallel efficiency on 512 processors for 268 million particles (Sbalzarini, Walther, Bergdorf, et al., 2006). A further optimized and adapted version of this code was later used on 16384 processors of an IBM BG/L for a vortex method simulation using 6.5 billion particles at 62% parallel efficiency (Chatelain et al., 2008a; Chatelain et al., 2008b).

In complex-shaped geometries, the adaptive domain decomposition and load-balancing methods of the PPM library become important. This has been tested in particle simulations of diffusion in the endoplasmic reticulum, a complex tubular network in biological cells (Sbalzarini, Mezzacasa, Helenius, & Koumoutsakos, 2005; Sbalzarini, Hayer, Helenius, & Koumoutsakos, 2006). The simulations used up to 1 billion particles distributed onto up to 242 processors of the IBM p690 computer at CSCS. The weak-scaling efficiency was better than 84% in all cases, demonstrating the scalability of automatic, transparent data distribution using the abstractions presented above (Sbalzarini, Walther, Bergdorf, et al., 2006).

In summary, the PPM library has demonstrated the feasibility and viability of a processor-independent, data-transparent parallel programming model for hybrid particle-mesh simulations. The model was based on a preliminary version of the abstractions presented above.

CONCLUSION

As computing is becoming an integral part of many sciences and the models become more complex, HPC will be indispensable catalyst of progress as its use proliferates. However, the complexity of such large hardware systems and the architecture of modern (heterogeneous) multi-core processors has led to several gaps, of which the knowledge gap is arguably the most important one. Efficient use of HPC machines and their availability to emerging user fields depends on new programming paradigms and algorithms. Some of the resources must thus be invested in computer science research and education. It is widely accepted that an additional layer of abstraction, which hides the hardware complexity and exposes a programming model that is independent of the number and architecture of processors, is one possible solution.

We have introduced data and operation abstractions that provide intermediate granularity and disentangle computation from communication, enabling automatic analysis and optimization. These abstractions also define a language in which the parallel semantics of simulations can be expressed, and they define encapsulated functionalities that can be implemented as architecture-specifically optimized modules of a middleware or run-time system. In the future, these implementations could be constructed and optimized using auto-tuning code generators, provided the necessary performance prediction tools are available. It is also possible to construct simulation compilers that directly translate an abstract definition of a simulation to the proper sequence of middleware calls. This would have several benefits: (a) the abstract definition is independent of the number of processors, ensuring portability and ease of use; (b) the implementations of the individual abstractions (i.e., the modules of the middleware) can be tuned to the machine architecture, thus transparently optimizing hardware use; (c) implementing a parallel simulation is reduced to writing its abstract definition, thus greatly reducing code development time and the knowledge gap; (d) if better algorithms become available they only have to be implemented in the middleware, immediately benefitting all simulations without changing their abstract descriptions.

The presented abstractions are deliberately kept simple, encapsulating simple and regular data structures such as particles and meshes. This is motivated by our expectation that simple data structures tend to scale better to larger numbers of processors, which seems to be confirmed by the experiences made with the PPM library. In particular, operation counts lose importance compared to memory access and communication (latency) counts. Several efficient single-processor algorithms use trees or graphs, which might not parallelize well and tend to be memory limited. Instead of using a complex surface triangulation on a large distributed system, for example, it might be favorable to use a simple Cartesian mesh with an embedded boundary method.

In summary, the abstractions and middleware presented in this paper provide a starting point that has already led to highly scalable and easy-to-implement parallel simulations (Sbalzarini, Walther, Bergdorf, et al., 2006). At this stage, however, there are more open questions (and research opportunities!) than solutions. Even though many bits and pieces exist in all areas (abstractions, languages, compilers, middleware, tools), they are yet to be combined in a programming model that is independent of the number of processors and their architecture. In this venture, care must be taken that the abstractions are general enough not to “over-fit” the present-day multi-core architectures. Abstractions should be sufficiently independent of the

hardware and based on theoretical compute models. This will help prevent today's simulations from becoming tomorrow's legacy codes.

REFERENCES

Angskun, T., Fagg, Bosilca, G., Pjesivac-Grbovic, J., & Dongarra, J. J. (2006). Scalable fault tolerant protocol for parallel runtime environments. In *Euro PVM/MPI* (p. ICL-UT-06-12).

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., et al. (2006). *The landscape of parallel computing research: A view from Berkeley* (Technical Report No. UCB/EECS-2006-183). University of California at Berkeley.

Beck, M., Dongarra, J. J., Fagg, G. E., Geist, G. A., Gray, P., Kohl, J., et al. (1999). HARNESS: A next generation distributed virtual machine. *Future Generation Computer Systems*, 15, 571–582.

Bhatele, A., & Kale, L. V. (2008). Application-specific topology-aware mapping for three dimensional topologies. In *Proceedings of the IEEE international symposium on parallel and distributed processing* (pp. 1–8).

Bonomi, E., Flück, M., George, P., Gruber, R., Herbin, R., Perronnet, A., et al. (1989). Astrid: Structured finite element and finite volume programs adapted to parallel vectorcomputers. *Computer Physics Reports*, 11, 81–116.

Bronevetsky, G. (2009). Communication-sensitive static dataflow for parallel message passing applications. In *International symposium on code generation and optimization (CGO)* (pp. 1–12).

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

Chapple, S. R., & Clarke, L. J. (1994). The parallel utilities library. In *Proceedings of the IEEE scalable parallel libraries conference* (pp. 21–30).

Chatelain, P., Curioni, A., Bergdorf, M., Rossinelli, D., Andreoni, W., & Koumoutsakos, P. (2008a). Billion vortex particle direct numerical simulations of aircraft wakes. *Computer Methods in Applied Mechanics and Engineering*, 197, 1296–1304.

Chatelain, P., Curioni, A., Bergdorf, M., Rossinelli, D., Andreoni, W., & Koumoutsakos, P. (2008b). Vortex methods for massively parallel computer architectures. *Lecture Notes in Computer Science*, 5336, 479–489.

Colella, P. (2004). *Defining software requirements for scientific computing*. Presentation slides.

Djordjevic, G. L., & Tosic, M. B. (1996). A heuristic for scheduling task graphs with communication delays onto multiprocessors. *Parallel Computing*, 22, 1197–1214.

Durand, D., Jain, R., & Tseytlin, D. (2003). Parallel I/O scheduling using randomized, distributed edge coloring algorithms. *Journal of Parallel and Distributed Computing*, 63, 611–618.

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., & Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), 375–408.

Fagg, G. E., Gabriel, E., Bosilca, G., Angskun, T., Chen, Z., Pjesivac-Grbovic, J., et al. (2004). Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the international supercomputing conference (ISC2004)*.

Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., et al. (2006). Sequoia: Programming the memory hierarchy. In *Proceedings of the SC06 conference on high performance networking and computing* (p. 83). ACM/IEEE.

Greengard, L., & Gropp, W. D. (1990). A parallel version of the fast multipole method. *Computers & Mathematics with Applications*, 20(7), 63–71.

Gruber, R., & Keller, V. (2009). *HPC @ GreenIT*. Berlin, Germany: Springer.

Hieber, S. E., & Koumoutsakos, P. (2008). An immersed boundary method for smoothed particle hydrodynamics of self-propelled swimmers. *Journal of Computational Physics*, 227, 8636–8654.

Hockney, R. W., & Eastwood, J. W. (1988). *Computer simulation using particles*. London, UK: Institute of Physics Publishing.

Huang, C. (2004). *System support for checkpoint/restart of Charm++ and AMPI applications*. Unpublished Master thesis, University of Illinois, Dept. of Computer Science.

Huang, C., Lawlor, O., & Kale, L. V. (2004). Adaptive MPI. *Lecture Notes in Computer Science*, 2958, 306–322.

Kale, L. V., Bhandarkar, M., Jagathesan, N., Krishnan, S., & Yelon, J. (1996). Converse: an interoperable framework for parallel programming. In *Proceedings of the IEEE international parallel processing symposium (IPPS)* (pp. 212–217).

Kale, L. V., Bohm, E., Mendes, C. L., Wilmarth, T., & Zheng, G. (2007). Programming Petascale Applications with Charm++ and AMPI. In *Petascale computing: Algorithms and applications*. CRC Press.

Karypis, G., & Kumar, V. (1998a). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 359–392.

Karypis, G., & Kumar, V. (1998b). Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48, 96–129.

Kim, S. (2001). An O(N) level set method for Eikonal equations. *SIAM Journal on Scientific Computing*, 22(6), 2178–2193.

Message Passing Interface Forum. (2008). *MPI: A message-passing interface standard, version 2.1*. Stuttgart, Germany: High-Performance Computing Center Stuttgart.

Moon, B., & Saltz, J. (1994). Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the IEEE scalable high-performance computing conference* (pp. 176–183). IEEE.

Püschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., et al. (2005). SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 232–275.

Sbalzarini, I. F., Hayer, A., Helenius, A., & Koumoutsakos, P. (2006). Simulations of (an)isotropic diffusion on curved biological surfaces. *Biophysical Journal*, 90(3), 878–885.

Sbalzarini, I. F., Mezzacasa, A., Helenius, A., & Koumoutsakos, P. (2005). Effects of organelle shape on fluorescence recovery after photobleaching. *Biophysical Journal*, 89(3), 1482–1492.

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

Sbalzarini, I. F., Walther, J. H., Polasek, B., Chatelain, P., Bergdorf, M., Hieber, S. E., et al. (2006). A software framework for the portable parallelization of particle-mesh simulations. *Lecture Notes in Computer Science*, 4128, 730–739.

Sethian, J. A. (1999). *Level set methods and fast marching methods*. Cambridge, UK: Cambridge University Press.

Thakur, R., Bordawekar, R., Choudhary, A., Ponnusamy, R., & Singh, T. (1994). PASSION runtime library for parallel I/O. In *Proceedings of the IEEE scalable parallel libraries conference* (pp. 119–128). IEEE.

Thakur, R., Gropp, W., & Lusk, E. (1999). Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th symposium on the frontiers of massively parallel computation* (pp. 182–189).

Turkiyyah, G., Reed, D., & Yang, J. (1996). Fast vortex methods for predicting wind-induced pressures on buildings. *Journal of Wind Engineering and Industrial Aerodynamics*, 58, 51–79.

UPC Consortium. (2005). *UPC language specifications, v1.2* (Technical Report LBNL-59208). Lawrence Berkeley National Laboratory.

Verlet, L. (1967). Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1), 98–103.

Vizing, V. G. (1964). On an estimate of the chromatic class of a p-graph. *Diskret. Analiz*, 3, 25–30. (in Russian)

Voronenko, Y. (2008). *Library generation for linear transforms*. Unpublished doctoral dissertation, Carnegie Mellon University.

Walther, J. H., & Sbalzarini, I. F. (2009). Large-scale parallel discrete element simulations of granular flow. *Engineering Computations*, 26(6), 688–697.

Wray, A. A. (1988). *A manual of the vectoral language* (Internal report). Moffett Field, California: NASA Ames Research Center.

Zheng, G., Shi, L., & Kale, L. V. (2004). FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *Proceedings of the IEEE conference on cluster computing* (pp. 93–103).

ACKNOWLEDGMENTS

I am deeply grateful to Prof. Dr. Jens H. Walther (DTU, Copenhagen, Denmark) for his initiative and vision to start the PPM project as well as for countless discussions on parallel programming and joint programming sessions. I also thank Prof. Dr. Petros Koumoutsakos (ETH Zurich, Switzerland), in whose group the PPM project started and is still on going, as well as all contributors to the PPM library, in particular Dr. Michael Bergdorf (ETH Zurich, Switzerland) and Dr. Philippe Chatelain (ETH Zurich, Switzerland). Special thanks also to Prof. Dr. Ralf Gruber (EPFL, Switzerland), Dr. Greg Bronevetsky (LLNL, USA), Prof. Dr. Jens H. Walther (DTU, Copenhagen, Denmark), Rajesh Ramaswamy (ETH Zurich, Switzerland), and Justin Park (ETH Zurich, Switzerland) for proofreading the manuscript and providing valuable feedback and suggestions. We thank the Swiss National Supercomputing Centre (CSCS) for dedicated access to their HPC systems.

ENDNOTES

[1]On the Intel Nehalem architecture, the ratio is one order of magnitude if only 1 or 2 cores of the processor are used in a pipelined way.