PaRSEC: Exploiting Heterogeneity to Enhance Scalability

New high-performance computing system designs with steeply escalating processor and core counts, burgeoning heterogeneity and accelerators, and increasingly unpredictable memory access times call for dramatically new programming paradigms. These new approaches must react and adapt quickly to unexpected contentions and delays, and they must provide the execution environment with sufficient intelligence and flexibility to rearrange the execution to improve resource utilization.

"Nothing endures but change." This old adage certainly applies to the hardware available to computer scientists since the dawn of the computing age. Beginning in the 1970s, vector computing was indisputably the technology for those seeking the highest possible performance; in the 1980s, the introduction of multiprocessor vector systems added a new dimension to this approach. By the 1990s, improvements to the price/performance ratio of conventional microprocessors led to massively parallel processor architectures. Interconnected by network interface cards, they replaced vector processor systems with symmetric multiprocessor designs. This design dominated most of the market until the end of the decade, when the concept of cluster computing emerged. In the middle of the 2000s, however, traditional processor designs hit physical limits that prevented them from continuing the race for improved performance by simply running the clock of each new generation of processors at ever-higher frequencies.

Having reached a hard upper limit on clock frequencies, designers began to seek higher performance by increasing the number of computing resources on each chip; the manycore revolution began. Manycore designs have indeed been able to sustain (now familiar) exponential improvements in processor performance, but only at the cost of a sharp escalation in the amount of parallelism inside a node. Fast forward several years, and we find that issues of power consumption and performance price points have given rise to dedicated hardware accelerators, providing a large number of specialized cores not directly under the control of the traditional operating system. These accelerators come from diverse vendors, and because each vendor usually has its own programming paradigm, as well as frequently changing interfaces and design characteristics, they confront software developers with a formidable set of new programming challenges. The usual abstractions provided by the operating system and the traditional software stack (programming models, execution environments, and tools) only partially help the programmer striving to utilize heterogeneous resources (http://herbsutter.com/welcome-to-the-jungle); the additional complexity hinders all efforts at writing high-performing yet portable applications.

1521-9615/13/$31.00 © 2013 IEEE
Copublished by the IEEE CS and the AIP

George Bosilca, Aurelien Bouteiller, and Anthony Danalis
University of Tennessee
Mathieu Faverge
Bordeaux Institute of Technology
Thomas Herault and Jack J. Dongarra
University of Tennessee

The challenging environment we describe in the next section calls for flexible models that can adapt the execution flow with respect to the algorithms used to match not only the available hardware's capabilities but also its availability. In short, a dynamic environment calls for a dynamic execution model. Historically, this idea has been investigated in other contexts, typically in grid environments. But the increasing complexity of execution environments has brought this concept back to the fore for high-performance computing (HPC), with models exhibiting finer task granularity and runtimes supporting larger and more heterogeneous platforms. Several research groups are actively investigating programming paradigms based on ideas revolving around a runtime-supported task-based graph concept.[1–4] Here, we describe our particular approach.

Today's Challenging Environment

Looking outside the boundaries of a single processor reveals several challenges: as the number of processors on a node increases beyond a certain point, the use of flat interconnection backbones is excluded. Consequently, the use of deep Non-Uniform Memory Access (NUMA) has become pervasive, with communication delays varying according to the position of a given process in the communication topology; each synchronization results in an unpredictable waiting time, and intersocket memory bandwidth becomes a scarce resource that must be carefully managed to avoid contention. Moreover, the heterogeneity of the computing resources involved further complicates the challenge of ensuring an efficient distribution of work among those resources, which in turn generates unsolvable multidimensional optimization problems. In summary, the massive parallelism and multidimensional heterogeneity of current and expected high-performance platforms both differentiates them sharply from the machines of the past and, for the same reasons, causes them to clash with the legacy SPMD programming model.

In an attempt to accompany this evolution on the software side, the HPC community has brought about a complex ecosystem of middleware dedicated to facilitating the use of the massively parallel resources that constitute the workhorse of computational simulation. Since the mid-1980s, the ubiquitous programming model for parallel applications has been both explicit message passing to exchange information between computing nodes and parallel threads inside the node (with explicit, or implicit, synchronization) through a thread library, such as Pthreads, or through the use of a parallel language. These two dominant abstractions gave birth to numerous highly successful supporting stacks—for example, Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI), on the side of explicit internode communications, and OpenMP for shared-memory machines. We could argue that these models have been successful because they provided a level of abstraction that delivered portable performance—a code written in MPI, for instance, could be deployed, unchanged, on many target systems—yet still achieved reasonable performance levels. However, the issue with all these variations of the SPMD programming model is that they encourage bulk-synchronous programming (BSP), in which sequential processes work in parallel and then synchronize, sometimes globally, to ensure the computation's consistency. Consequently, these models don't cope productively with the system noise, variable completion times, and performance heterogeneity of the processing units that we see in the new era of massively manycore and heterogeneous systems.

In addition, due to the multidimensional heterogeneity of modern architectures, it is becoming increasingly clear that using only one of these abstract models in a one-size-fits-all approach fails to deliver the desired performance level. With systems that encompass both large NUMA shared-memory processors and the accelerators gathered in large constellations, performance-conscious developers are forced to employ multiple abstract models simultaneously. This is evidenced by the way in which CUDA, OpenMP, and MPI are sometimes combined in the same application to map parallelism on different types of hardware in the same machine. Unfortunately, such hybrid programming efforts have had mixed results; the desired performance boost often fails to materialize after near-heroic investments in software engineering.

One explanation of this regrettable phenomenon is that the separation-of-concerns barrier is being violated. End user programmers must decide which parts of the algorithm should be expressed using a particular parallelism abstraction when developing their application. Hence, the mapping of an application to a particular type of computing resource becomes a static decision. The burden imposed on application

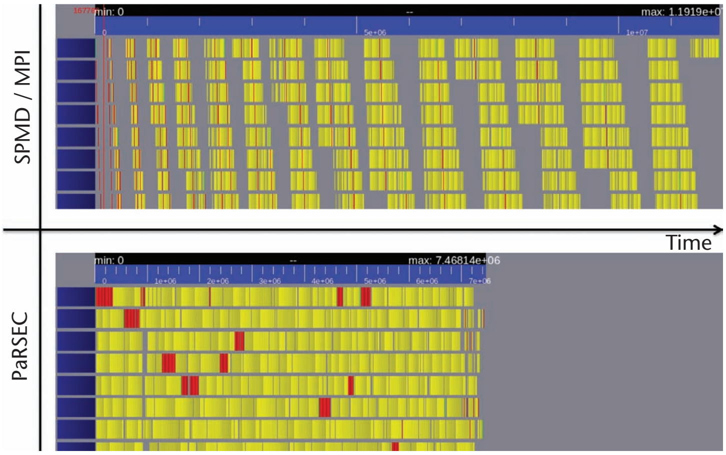


Figure 1. Comparison of execution traces for the same algorithm using the single-program, multiple data message passing interface (SPMD/MPI) programming model and the dataflow model. Gray areas denote idle time; red and yellow depict useful work happening. The SPMD/MPI approach makes it difficult to resolve imbalances that result in a longer execution time and more idle/wait time. In the runtime-supported dataflow model, most of the jitter and imbalance are resolved through an adaptive rebalancing of the work.

developers to optimize and tune their code in this way has become unsustainable: heavy modifications to adapt the application to cutting-edge architectures become a daunting task, distracting the attention of scientists from their core competencies. Supporting the development of applications on a diversity of target hardware architectures—while still achieving efficiency and performance with minimum programming effort—requires a far more flexible infrastructure. Such an infrastructure could incorporate autoconfigurability as one possible approach to achieving this goal.

Another significant concern is that legacy programming approaches assume that a static load balance, often completely under the programmer's supervision, is enough to exploit the machine's parallel potential. However, due to the variety of processing elements participating in the computation, programmers find it increasingly difficult to express a load balance that can hold for all types of hardware and for the computation's entire duration. It's therefore necessary to find solutions that dynamically rebalance the work among resources so as to tolerate the inevitable jitter that arises in heterogeneous compute nodes.

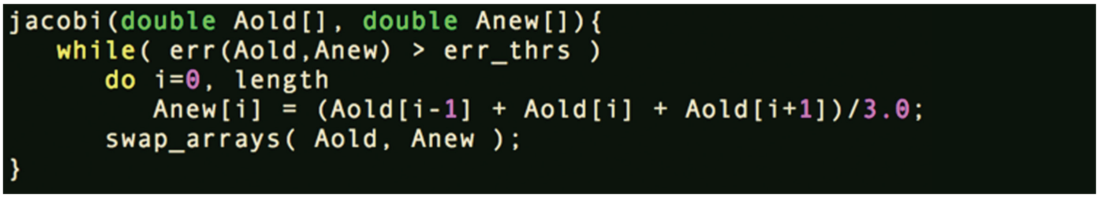
A Dynamic Runtime for a Dynamic World

Task-based runtime systems have properties that make them more versatile than legacy execution models. For example, because such a runtime system manages the execution, it can perform dynamic, opportunistic scheduling decisions. It can also orchestrate an adaptive response to conditions of the resources it currently senses (idling accelerators, load imbalance, network congestions, and so on), adapting the way in which it maps and schedules computations onto resources, while at the same time minimizing data transfers, either on the network or with memory banks.

In this article, we make the case for a runtime-supported dataflow programming model—specifically, the PaRSEC (http://icl.cs.utk.edu/parsec/)[5] runtime system—to alleviate some of the challenges imposed by changes at the hardware level. We emphasize the fact that such an approach not only has benefits for current architectures but also provides a portable way to automatically adapt algorithms to new hardware trends. PaRSEC can boost the performance of distributed, task-based algorithms, as was demonstrated in the D-PLASMA[5] library (http://icl.cs.utk.edu/dplasma), which we developed using PaRSEC.

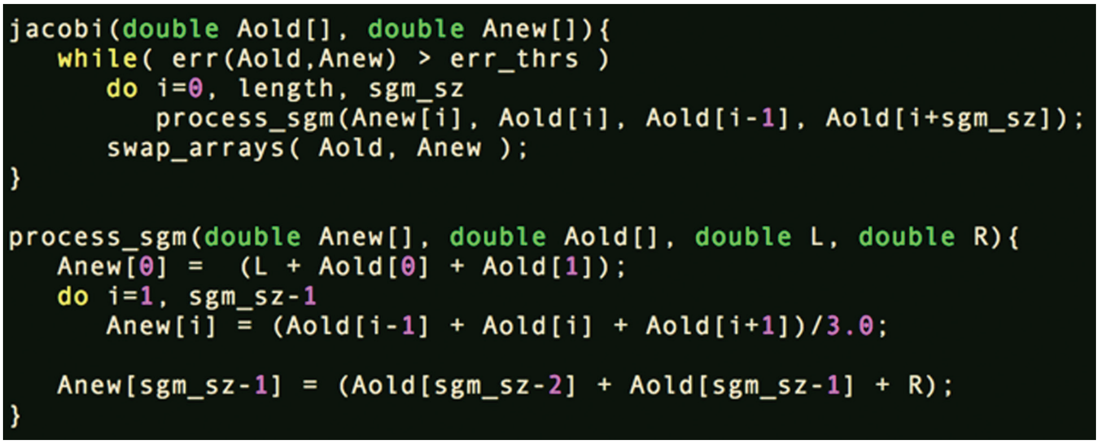
To substantiate the claim that a runtime can indeed deliver superior performance, consider the execution traces shown in Figure 1. The top trace shows the execution of an application using MPI, and the bottom trace shows the execution of the same application using the PaRSEC runtime. In both cases, the application is a QR factorization, a dense direct matrix factorization commonly used to solve a linear least square problem. In both cases, the horizontal axis depicts time, and each horizontal stripe (within each trace) represents the behavior of one thread. Useful work is depicted in red and yellow; idle time is depicted in gray. Clearly, the highly dynamic scheduling approach featured in PaRSEC utilizes the hardware much more efficiently than static approaches, and although the algorithm and dataset are the same in both figures, Figure 1 demonstrates a significant reduction in the execution time.

This dynamic runtime is only one side of the necessary abstraction. To reach the desired level of flexibility, we must be able to expose much more of the available parallelism than we have traditionally done, and the runtime must be capable of freely exploiting it to increase the opportunities for useful computation. A dynamic runtime can adapt the execution to the current resources condition, as long as it is able to discover concurrency in the application. This calls for an expression of the parallelism that is practical to



```
jacobi(double Aold[], double Anew[]){
   while( err(Aold,Anew) > err_thrs )
      do i=0, length
         Anew[i] = (Aold[i-1] + Aold[i] + Aold[i+1])/3.0;
      swap_arrays( Aold, Anew );
}
```

(a)



```
jacobi(double Aold[], double Anew[]){
   while( err(Aold,Anew) > err_thrs )
      do i=0, length, sgm_sz
         process_sgm(Anew[i], Aold[i], Aold[i-1], Aold[i+sgm_sz]);
      swap_arrays( Aold, Anew );
}

process_sgm(double Anew[], double Aold[], double L, double R){
   Anew[0] =  (L + Aold[0] + Aold[1]);
   do i=1, sgm_sz-1
      Anew[i] = (Aold[i-1] + Aold[i] + Aold[i+1])/3.0;

   Anew[sgm_sz-1] = (Aold[sgm_sz-2] + Aold[sgm_sz-1] + R);
}
```

(b)

Figure 2. 1D Jacobi method in different programming styles. (a) Plain serial code. (b) Task-based serial code. The task-based serial code expresses the problem as a combination of fine and coarse granularities. The outer coarse grain loop can be analyzed to extract dataflow parallelism.

end users, expressive, and avoids cumbersome restrictions that prevent the flexible scheduling of operations on heterogeneous hardware.

The Dataflow Model

The concept of dataflow has been center-stage for program execution since nearly the beginning of computer science. As early as 1966, A.J. Bernstein postulated a set of conditions[6] that describe what operations can be executed in parallel with any other, or how operations can be reordered while preserving the program's semantics. From these conditions, you can deduce a program's dataflow. Compiler optimizations and hardware designs aim to improve applications' execution speed, whether they're parallel or serial, while still observing the limitations set by the applications' dataflow. Dataflow research has yielded results at different levels of granularity and applicability, but in most cases, an appropriate unit of computation is considered to be a set of atomic computations that receives some input, performs some operations, and generates some output. For our purpose here, we refer to such a unit as a *task*. The interactions between these tasks—what data they use or produce—is the dataflow, which the system (compiler, hardware, or runtime) builds and uses to orchestrate the task execution and data movement.

At the finest level of granularity, compiler optimizations, such as instruction scheduling or vectorization, and hardware features, such as pipelining and superscalar execution, rely on speeding up execution by analyzing the dataflow of small blocks of a program to discover instructions that are independent and can thus proceed concurrently. At this granularity, each instruction becomes a task, and the dataflow analysis's role is to examine the operands of different instructions to discover how they depend on one another.

At the other extreme of granularity, entire parallel programs can be written such that computation takes place in large groups of operations that have well-defined dependencies with one another and can thus be defined as tasks. In procedural programming languages, such as C, C++, Fortran, and so on, the natural unit of atomic computation is a function (or subroutine, in Fortran parlance). Functions that can behave as tasks have well-defined entry and exit points and can be "pure"—that is, have only side effects that can be described in terms of their input and output data.

Figure 2 illustrates the idea of developing programs that lend themselves to task-based execution. Figure 2a shows a pseudocode implementation of the 1D Jacobi method. The program iterates until reaching a steady state; in each iteration, every array element is replaced by the average of its previous value and its immediate neighborhood. Figure 2b shows a program that computes exactly the same result, only now the computation is logically segmented, with every segment processed by the function `process_sgm()`. This function is pure in that it only modifies memory passed to it through its arguments. Although both programs are serial, the latter can be readily processed by a dataflow system and executed using a task-based runtime.

We believe future high-performance applications and runtimes should be targeted at this coarse level of granularity, which is achieved when whole functions are defined as tasks. Operating at a coarse granularity has significant practical implications. Consider, for example, an application in which each task has an execution time on the order of tens of microseconds or above. In such a case, if additional code execute every time a task ran, such that the additional code complete in less than a few hundred nanoseconds, then the overhead incurred by the application would be less than 1 percent. Still, hundreds of nanoseconds are sufficient time for a modern computer to perform a large number of operations, including traversals of several memory structures; this is especially so if these structures are traversed frequently and thus reside in some level of the cache hierarchy. This tolerance for external book-keeping operations enables coarse-grain, task-based execution models to utilize runtime engines that continuously monitor the application's progress and make dynamic decisions. This is in stark contrast with the BSP model, in which a static schedule is embedded into the algorithm's expression as explicitly specified by the programmer in the program's code flow. Although the BSP approach eliminates the need for management and scheduling overhead and is therefore very efficient for instruction-level handling, it lacks the flexibility necessary to adapt to runtime conditions.

A runtime engine that's aware of both the tasks to be executed and the dataflow that connects them provides other significant benefits. A runtime engine that's continuously aware of the current state of execution and the next tasks that will become available, as well as the data that they'll require, can automatically handle the communication necessary to transfer this data between nodes of a distributed memory system. This capability makes the transition from shared to distributed memory execution seamless. Furthermore, the runtime can schedule tasks based on specialized rules or constraints deduced from algorithmic priorities, generated communication volume, cache locality, or several other (combinations of) heuristics. These different scheduling heuristics can optimize a variety of goals, such as task duration, energy consumption, or the amount of communication-computation overlap. Of course, the further into the future of an application's execution that a runtime can see, the higher quality the scheduling decisions it can make. In the best case, an application's component tasks and the dataflow between them can be given an algebraic expression that can be evaluated in constant time, so that the application's future execution can be explored to arbitrary depths. This programming and execution model is one of the models that PaRSEC supports.

The PaRSEC Runtime

PaRSEC employs the dataflow programming and execution model to provide a dynamic platform that can address the challenges posed by distributed heterogeneous hardware resources. The system's central component, the *runtime*, combines the source program's task and dataflow information with supplementary information provided by the user—such as data distribution or hints about the importance of different tasks—and orchestrates task execution on the available hardware.

From a technical perspective, PaRSEC is an event-driven system. When an event occurs, such as task completion, the runtime reacts by examining the dataflow to discover what future tasks can be executed based on the data generated by the completed task. The runtime handles the data exchange between distributed nodes, and thus it reacts to the events triggered by the completion of data transfers as well. When no events are triggered because the hardware is busy executing application code, the runtime gets out of the way, allowing all hardware resources to be devoted to the application code's execution.

Due to the dataflow representation (see Figure 3), communications become implicit and thus are handled automatically as efficiently as possible by the runtime. Specifically, in the PaRSEC model, data exchange isn't explicitly coded by the developers into their application, as in MPI, but implied in the application's dataflow representation. Given that

PaRSEC is aware of this representation and has knowledge of the mapping of tasks onto compute nodes, its runtime can perform all necessary data exchanges without user intervention. This has the benefit of simplifying the development of distributed memory parallel applications; most importantly, it allows the runtime to automatically make use of efficient nonblocking communication and advanced collective communication algorithms to achieve communication-computation overlapping and hide significant parts of the communication overhead.

Task scheduling within each node is also one of the runtime's responsibilities. Specifically, as tasks complete, they generate data that enables the execution of other tasks. The runtime keeps track of the tasks that have completed (the active tasks in Figure 3), discovers the tasks that can execute next, and decides which hardware resources (CPU cores, accelerators, and coprocessors) should be devoted to each new task. Consequently, applications that use PaRSEC can enjoy high efficiency because of its advanced scheduling algorithms for managing data locality, load balancing, and algorithmic priorities. At the same time, it liberates application developers from the difficult and tedious intricacies of micromanaging processes, threads, and other exotic low-level library primitives and interfaces. By exposing a flat view of the system, the PaRSEC runtime manages all this complexity internally.

High-Productivity Ecosystem

Given the ongoing increase in system complexity, it's clear that any viable programming model will need to help developers achieve the best performance possible, even while helping them keep their efforts below a reasonable threshold. Arguably, then, for emerging system and programming paradigms, ease of use and development aren't only relevant metrics but are often as important as achievable performance. A system that doesn't promote usability can hardly expect to be widely adopted by end users, even if it delivers better performance than the status quo. In this section, using PaRSEC to illustrate the case, we describe how a shift toward the dataflow programming model from conventional practices can preserve and enhance programmer productivity while achieving superior efficiency and scalability. It can accomplish this, in part, by providing a unified, expressive, and powerful representation of application parallelism.

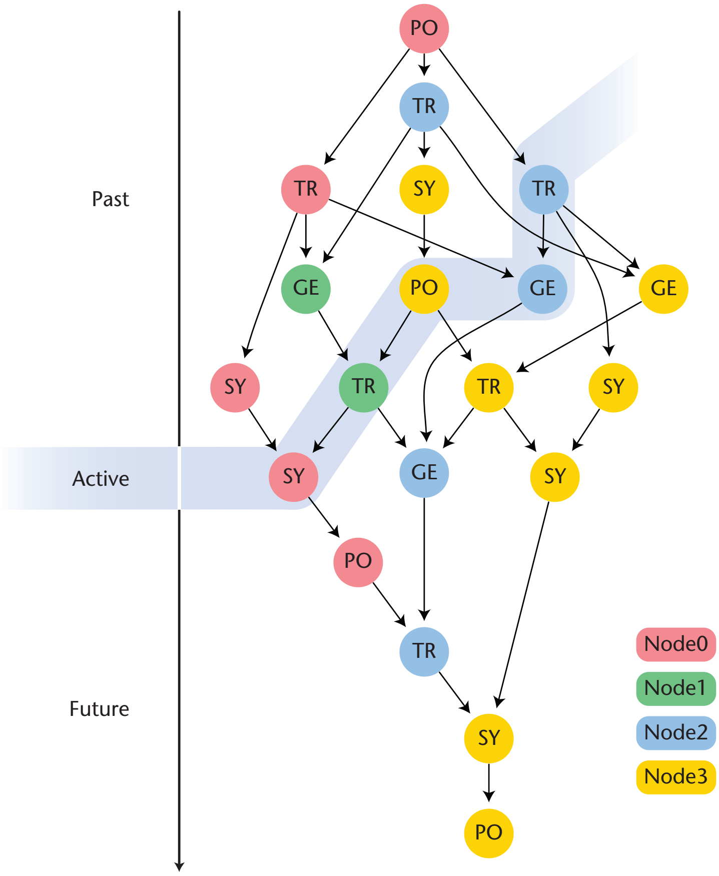


Figure 3. The PaRSEC runtime walks the DAG using a concise representation that instantiates only the relevant tasks at each computing node. Only the active, local tasks need to be stored and considered.

The resulting parallel workload of concurrent tasks is managed by the runtime scheduler and decouples the communication patterns from the algorithm specification.

Parallelism Expression

PaRSEC expresses an algorithm as a Directed Acyclic Graph (DAG) of tasks and an associated dataflow, implying a significant shift in software engineering practice. It can enable the use of much larger and much more complex supercomputers, but only if users embrace it as an effective way of developing production code. To that end, we created, and are actively extending, tools that aim to help developers make their codes "PaRSEC-enabled." Because our system's users are parallel application developers, the primary tools we created for interfacing with them are analysis and compilation tools. Figure 4 illustrates how these tools integrate in the larger perspective of existing productivity ecosystem (here represented by the Tensor

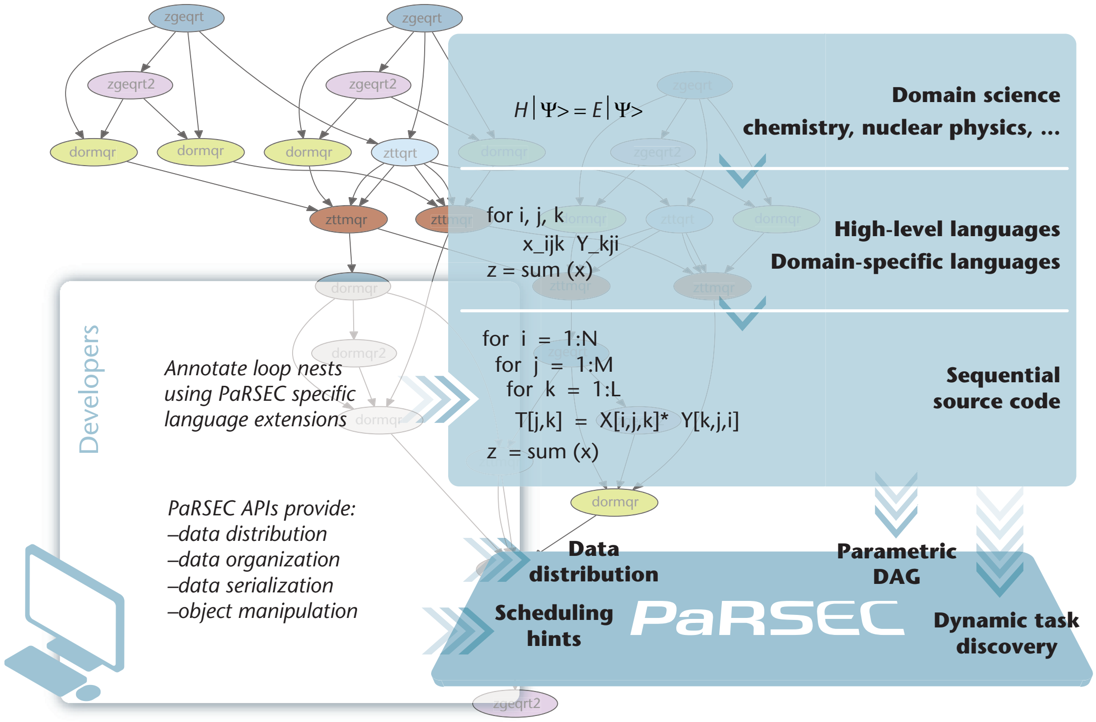


Figure 4. The PaRSEC environment. At a high level, productivity tools convert domain-specific codes into a dataflow representation. The dataflow representation is combined with the PaRSEC runtime library to form the versatile application representation.

Contraction Engine). Specifically, our system includes two compilation tools: the front-end and back-end compilers.

The front-end compiler aims to make it as easy as possible to use PaRSEC and to enable a seamless upgrade of legacy software that was created for multicore processors but not for distributed heterogeneous systems. The input to the front-end compiler must conform to a canonical form, which enables the compiler to extract and analyze all dataflow information. After the analysis, the compiler produces a file containing a parameterized task graph—represented in a PaRSEC-specific notation—that describes the input program's tasks and the dependencies between them in a symbolic- and problem-size-independent way.[7]

However, the source code's canonical form limits the input to affine codes. That is, loops with bounds that might be parameterized but fixed—neither the loop bounds nor the induction variable can be altered within the loop, or depend on function calls or user data—and memory accesses are linear functions of the induction variables and constant parameters. This is akin to programs written using the DO loop construct of Fortran 77. Although several interesting problems meet this limitation (dense linear algebra, tensor contraction, and so on), not all parallel applications can be expressed as such. For this reason, PaRSEC allows the human developer to alter the program's dataflow representation or even write it completely by hand. This way, the developer can go beyond the front-end compiler's limitations and trade simplicity for expressivity.

There's a significant difference between this approach and what other task-based runtimes typically do. In the latter, the execution flow is directly derived from sequential execution of the target application; discovering the task graph in a scalable way in such cases is a challenge in distributed environments. In contrast, PaRSEC's parameterized task graph provides a concise symbolic task representation that allows scalable task discovery and scheduling in distributed environments.

Data Affinity and Movement

As communications are implicit in the dataflow model, the algorithm's expression is independent of task placement and data affinity. Yet, maximum performance on distributed memory machines demands that the developer control (even loosely) the communication volume and pattern. By default, PaRSEC expresses task placement as affinity functions to data following the "owner computes" rule; common data distributions, such as 2D-cyclic matrices, are provided in the PaRSEC toolkit. The programmer can write functions to describe arbitrary distributions for both input data and task placement. This three-stage development process—algorithm, data distribution, and optional task placement—helps improve code portability. Application developers first focus on expressing the algorithm in the most efficient manner with respect to parallelism. Then, they need to define an appropriate data distribution to fit the algorithm's specifics. Finally, if required, developers can fine tune communication volumes and patterns by replacing the owner computes rule with a different strategy for mapping tasks to data. When transporting the code to new hardware featuring a different (and possibly exotic) network topology, only the data and/or task distribution functions need to be tuned. The general algorithm can remain unchanged, thereby improving productivity when porting codes.

Fine Tuning and Expert Interface

When the input program's dataflow representation has been generated, PaRSEC provides a second compilation tool, the back-end compiler that translates this representation into C code stubs that are PaRSEC-enabled. This generated code can be compiled and linked with PaRSEC's runtime library using a traditional C compiler such as the GNU C Compiler (gcc) or Intel C Compiler (icc). This generated C code consists of the actual steps that will be taken when the program runs and has no limitation regarding its behavior. Therefore, an expert developer could alter or directly write code at this level.

Offering the option for such low-level programming might sound counterproductive, but it's similar to the familiar fact that, 40 years after C's creation, expert programmers still write critical code snippets in assembler. We believe that offering application developers a choice in the level of complexity versus expressivity, as well as the ability to combine different levels, offers the best promise for delivering excellent performance while keeping the amount of programmer effort within reasonable limits.

Performance and Correctness Analysis

While tools to facilitate the conversion from legacy programming models to dataflow representations are essential, it's also important to provide tools for debugging and analysis that are adapted to the new model. Programmers are then placed at the center of a feedback loop (as illustrated in Figure 5), taking input from a variety of correctness and performance analysis tools to fine tune their code at all levels of the PaRSEC compilers stack. An application programmer might want to verify that a parameterized task graph, whether automatically or manually written, correctly describes a given program's data dependencies. For this reason, we provide a tool for generating and displaying the application's graph of tasks at a developer-specified level of detail. Figure 3 shows the algorithm's dependency graph with four different kernels. This representation displays useful information for the developer—the graph's shape indicates the length of the algorithm's critical path, as well as the potential parallelism that can be automatically extracted from the application. The developer might want to generate a wider DAG to increase the available parallelism. To do so, at least two solutions are possible: rethink or change the algorithm to minimize or remove the need for synchronization or generate smaller grain computational tasks. The first solution might be impossible, and the second might lead to larger scheduling overhead. For this reason, we also provide an instrumentation framework for gathering low-level information at the task level.

The DAG representation is helpful for debugging purposes as well as for providing hints on parallelism available in the algorithm. However, it doesn't provide helpful information regarding the algorithm's efficiency in exploiting system resources. One common way to study the performance of parallel applications is to measure the elapsed time on each section of the code contained between synchronization points. The most expensive section is then analyzed to reduce the time spent on it. However, an algorithm's dataflow representation removes most, if not all, of the synchronization points. Therefore, in the case of data-driven, task-based execution, two things must be studied. The first is the performance of the tasks themselves, which is observed by collecting statistics such as time spent, cache misses, and so on. The second is scheduling efficiency to ensure that the correct choices have been made for a given DAG. PaRSEC lets the developer collect this kind of detailed information about tasks and scheduling so that system behavior can be analyzed, understood, and tuned.

For performance reasons, the developer must make sure that the choices are an efficient

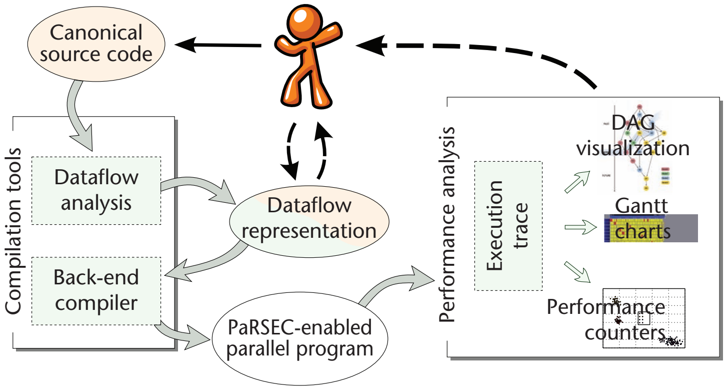


Figure 5. PaRSEC productivity tools. A rich ecosystem of tools supports the developer in converting legacy applications and high-level programming into a dataflow representation. The programmer has access and can directly alter the dataflow representation. A variety of debugging and performance-analysis tools are available to investigate the correctness and performance of the program's dataflow.

compromise between keeping data locality for local performance and maximizing parallelism within the DAG. To validate those choices, PaRSEC provides the ability to visualize execution traces as Gantt diagrams, such as those in Figure 1. Execution traces, coupled with the dataflow's DAG representation, show the set of active tasks at a given time with respect to the number of available resources. This functionality is an asset in understanding and adapting the scheduling to each class of problems.

In addition to "legacy"-type analysis tools, there's also a clear need for new tools to explore the complementary aspects of data and/or task distribution. Such tools will help programmers determine task efficiency and the way in which tasks affect the memory and computational load balance, both between and within the heterogeneous resources available on computational nodes. The community has only begun to explore the needs for debugging and analysis tools that these new algorithm representations are introducing.

Compute-intensive simulation has become a pillar of scientific discovery in the modern age. Ensuring that such simulations can run efficiently with high performance and accuracy on current and future parallel machines is critical to high scientific throughput and, consequently, is likely to have a significant impact on the pace of scientific progress.

Although the classical programming paradigm of hybrid message passing and shared memory served this purpose well over the past two decades, the complexity and heterogeneity of new hardware has relentlessly eroded its effectiveness. Despite possible improvements in the performance of some narrow benchmarks, without a changing of the guard in accepted programming models, we risk seeing declining benefits for real-world applications. As the gap between peak and sustained performance continues to increase, algorithms will be unable to reach their maximum performance potential, and fall short on both energy efficiency and resilience. The dataflow-driven programming paradigm described in this article, together with a corresponding runtime, provides an exciting opportunity to close this gap and increase code portability at the same time.

The era of dynamic and heterogeneous hardware that's now dawning clearly requires radical changes in the standard execution environment, and we believe that a model based on task graphs meets that requirement well. To further improve productivity and portability, new domain-specific extensions, as well as tools to better analyze, understand, and improve runtime behavior, should also be developed. Similar twists and turns on the narrow climb to exascale are likely to provide many more such exciting perspectives and challenges.

References

1. C. Augonnet et al., "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *J. Concurrency and Computation: Practice & Experience*, vol. 23, no. 2, 2011, pp. 187–198.
2. H. Kaiser, M. Brodowicz, and T. Sterling, "ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications," *Int'l Conf. Parallel Processing*, IEEE, 2009, pp. 394–401.
3. C. Lauderdale and R. Khan, "Towards a Codelet-Based Runtime for Exascale Computing: Position Paper," *Proc. 2nd Int'l Workshop Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ACM, 2012, pp. 21–26.
4. J. Planas, R. M. Badia, E. Ayguade, and J . Labarta. "Hierarchical Task-Based Programming with StarSs," *Int'l J. High-Performance Computing Applications*, vol. 23, no. 3, 2009, pp. 284–299.
5. G. Bosilca et al., "Dense Linear Algebra on Distributed Heterogeneous Hardware with a Symbolic DAG Approach," *Scalable Computing and Communications: Theory and Practice*, Jan. 2013, pp. 699–733.

6. A.J. Bernstein, “Analysis of Programs for Parallel Processing,” *IEEE Trans. Electronic Computers*, vol. 15, no. 5, 1966, pp. 757–763.
7. M. Cosnard and E. Jeannot, “Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling,” *Parallel Processing Letters*, vol. 11, 2001, pp. 151–168.

George Bosilca *is an assistant research professor at the University of Tennessee’s Innovative Computing Laboratory. His research interests range from low-level communication protocols to high-level constructs to support novel parallel programming paradigms. Bosilca received a PhD from the University of Paris. Contact him at bosilca@icl.utk.edu.*

Aurelien Bouteiller *is a researcher at the University of Tennessee’s Innovative Computing Laboratory. His research is focused on improving performance and reliability of distributed memory systems, algorithm-based fault tolerance, mechanisms to improve communication speed and balance of many-core clusters, and emerging dataflow programming models. Bouteiller received a PhD from the University of Paris. Contact him at bouteill@icl.utk.edu.*

Anthony Danalis *is a research scientist at the University of Tennessee’s Innovative Computing Laboratory. His research interests are in high-performancing computing (HPC), compiler analysis and optimization, system benchmarking, MPI, and accelerators. Danalis received a PhD in computer science from the University of Delaware. Contact him at danalis@icl.utk.edu.*

Mathieu Faverge *is an assistant professor at the Bordeaux Institute of Technology, France. His main research interests are numerical linear algebra algorithms for sparse and dense problems on massively parallel architectures, especially DAG algorithms relying on dynamic schedulers. Faverge received a PhD in computer science from the University of Bordeaux 1, France. Contact him at mathieu.faverge@inria.fr.*

Thomas Herault *is a research scientist at the University of Tennessee’s Innovative Computing Laboratory. His research interests include fault tolerance, HPC, and distributed algorithms. Herault received a PhD from the University of Paris-Sud. Contact him at herault@icl.utk.edu.*

Jack J. Dongarra *holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. Dongarra is a Fellow of the American Association for the Advancement of Science (AAAS), ACM, IEEE, and Society for Industrial and Applied Mathematics (SIAM) and a member of the National Academy of Engineering. Contact him at dongarra@icl.utk.edu.*