PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability

George Bosilca Aurelien Bouteiller Anthony Danalis
Mathieu Faverge Thomas Herault Jack J. Dongarra

October 7, 2013

Abstract

New HPC system designs with steeply escalating processor and core counts, burgeoning heterogeneity and accelerators, and increasingly unpredictable memory access times, call for one or more 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 the resource utilization. Some candidates in this area have already begun to emerge.

Here we present an approach based on task parallelism, one which reveals the application's parallelism by expressing its algorithm as a task flow, with data dependencies in-between. This strategy allows the algorithm to be decoupled from the data distribution and the underlying hardware, since the algorithm is entirely expressed as flows of data. This kind of layering provides a clear separation of concerns among architecture, algorithm, and data distribution. Developers benefit from this separation because they can focus solely on the algorithmic level without the constraints involved with programming for current and future hardware trends.

Keywords: C.1.4.d Scheduling and task partitioning; D.1.8 Distributed programming; D.2.17.i Programming Paradigms

1 The Jungle

"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, because of improvements to the price/performance ratio of conventional microprocessors, massively parallel processor architectures, interconnected by network interface cards, replaced vector processor systems, with the Symmetric Multi Processor designs dominating 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 point 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 since each vendor usually has their 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 [10]; and the additional complexity hinders all efforts at writing high performing yet portable applications.

Looking outside the boundaries of a single processor reveals new challenges because, as the number of processors on a node increases beyond a certain point, it excludes the use of flat interconnection backbones. 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 inter-socket memory bandwidth becomes a scarce resource that must be carefully managed to avoid contention. Moreover, the heterogeneity of the computing resources further complicates the challenge of ensuring a efficient distribution of work among computing resources, which in turn generates unsolvable multi-dimensional optimization problems. In summary, the massive parallelism and multi-dimensional 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 massively parallel resources that constitute the workhorse of computational simulation. Since the middle of the 1980s, the ubiquitous programming model for parallel applications has been 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, and highly successful supporting stacks: Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI), for example, on the side of explicit inter-node communications; OpenMP or Parallel Global Address Space (CoArrays, UPC, etc.) for shared-memory machines. It can be argued 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— and 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, where sequential processes work in parallel, and then synchronize, sometimes globally, to ensure the consistency of the computation. Consequently, these models do not cope productively with the system noise, variable completion times, and performance heterogeneity of processing units that we see in the new era of massively many-core and heterogeneous systems.

In addition, due to the multi-dimensional 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 level of performance. With systems that encompass both large NUMA shared memory processors and accelerators gathered in large constellations, performance conscious developers are forced to employ multiple abstract models at the same time. We now see this illustrated by the way in which CUDA, OpenMP, and MPI are being combined in the same application to map the 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 have to make decisions as to which parts of the algorithm should be expressed using a particular abstraction of parallelism 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 developers to optimize and tune their code in this way has become unsustainable. Heavy modifications to adapt the application to the cutting edge architectures becomes a daunting task, distracting the attention of scientists from their core competencies. It is clear that establishing an appropriate separation of concerns is now more necessary than ever. To support the development of applications on a diversity of target hardware architectures, while achieving efficiency and performance with minimum programming effort, a far more flexible infrastructure is required. Such an infrastructure may incorporate auto-configurability as one possible approach to achieving this goal.

Another significant concern is that legacy programming approaches (and their combined forms) have assumed that a static load balance, often completely under the supervision of the programmer, is enough to exploit the parallel potential of the machine. However, due 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 entire duration of the computation. It is 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.

The challenging environment described above clearly calls for flexible execution models that can adapt the execution flow with respect to the algorithm to match not only the capabilities of the available hardware but their availability during the execution. In short, in a dynamic environment, a dynamic execution model is needed.

Historically, this idea has been investigated in other contexts, mainly in grid environments. But the increasing complexity of execution environments has brought this concept back to the fore for 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 the task-based graph concept supported by a runtime [1, 5, 8, 9]).

2 A Dynamic Runtime for a Dynamic World

Task-based runtime systems have properties that make them more versatile than legacy execution models. For example, as it manages the execution, such a runtime can perform dynamic, opportunistic scheduling decisions. It can also orchestrate an adaptive response conditions of the resources it currently senses (e.g., idling accelerators, load imbalance, network congestions, etc.), 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 dataflow programming model supported by a runtime, in this case the PaRSEC [3, 7] runtime system, to alleviate some of the challenges imposed by the changes at the hardware level described above. We

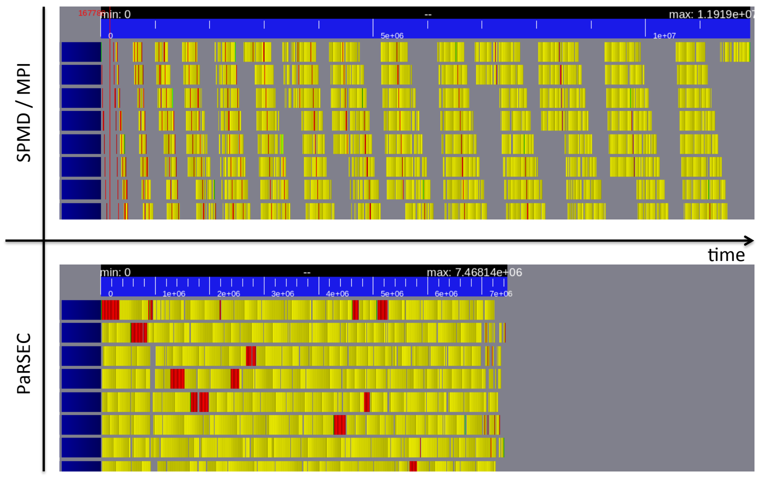


Figure 1: Comparison of traces of the executions of the same algorithm using the SPMD/MPI programming model and the dataflow model. Grey areas denote idle time. The SPMD/MPI approach makes it difficult to resolve imbalances that result in a longer execution time and more idle / wait time. In the dataflow model supported by a runtime, most of the jitter and imbalance are resolved through an adaptive rebalancing of the work.

emphasize the fact that such an approach not only has benefits on current architectures, but also provides a portable way to automatically adapt algorithms to new hardware trends. At the same time we will introduce some of the capabilities of the PaRSEC runtime, a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. PaRSEC can be used to boost performance of distributed, task-based algorithms, as was demonstrated for the case of dense linear algebra by the DPLASMA [3, 6] library, 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 which is commonly used in numerous applications 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 using the colors red and yellow, and idle time is depicted as gray. Clearly, the highly dynamic scheduling approach featured in PaRSEC utilizes the hardware much more efficiently than static approaches, and although the algorithm and data set 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, as it must be able to discover concurrency in the application to feed all computing units. 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. This calls for an expression of the parallelism that is practical to end-users, expressive, and avoids cumbersome restrictions that prevent the flexible scheduling of operations on heterogeneous hardware.

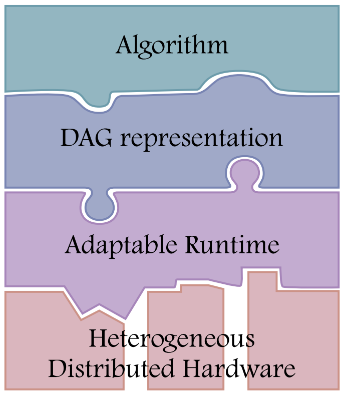


Figure 2: A dynamic runtime permits adapting the execution to the dynamic conditions of the execution. The runtime must see an expression of parallelism that gives it the flexibility to manage the execution in the most efficient fashion, but also gives appropriate control to end-users.

3 The Dataflow Model

The concept of dataflow has been at center-stage for program execution since the nearly beginning of computer science. As early as 1966, Bernstein postulated a set of conditions [2] that describe what operations can be executed in parallel with any other, or how operations can be re-ordered while preserving the semantics of the program. From these conditions, one can deduce the dataflow of a program. Compiler optimizations and hardware designs aim to improve the execution speed of applications, whether parallel or serial, while still observing the limitations set by the dataflow of the applications. Dataflow research has yielded results at different levels of granularity and different levels of applicability. 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 purposes here, we will refer to such 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 thus can proceed concurrently. At this granularity, each instruction becomes a task and the role of the dataflow analysis 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 thus can be defined as tasks. In procedural programming languages, such as C, C++, FORTRAN, etc., the natural unit of atomic computation is a function (or subroutine, in FORTRAN parlance). Functions have well defined entry and exit points and can be "pure", i.e., have only side-effects that can be described in terms of their input and output data.

```
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) Plain serial code

```
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) Task-based serial code

Figure 3: 1D Jacobi method in different programming styles. 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.

Figure 3 illustrates the idea of developing programs that lend themselves to task-based execution. In 3(a) we show a pseudocode implementation of the 1D Jacobi method. The program iterates until reaching a steady state, and in each iteration, every array element is replaced by the average of its previous value and its immediate neighborhood. In 3(b) we show a program that computes exactly the same result, only now the computation is logically segmented and every segment is processed by the function `process_sgm()`. Furthermore, this function is pure, in that it only modifies memory passed to it through its arguments. While 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. To that end, our group has developed PaRSEC, a dataflow system designed and implemented to operate at this coarse level of granularity. Operating at a coarse granularity has significant practical implications. Consider, for example, an application where each task has an execution time on the order of tens of microseconds or above. In such a case, if additional code were executed every time a task ran, such that the additional code completed in less than a few hundred nanoseconds, then the overhead incurred by the application would be less than one percent. Still, hundreds of nanoseconds is 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 bookkeeping operations enables coarse grain task-based execution models to utilize runtime engines that continuously monitor the progress of the application and make dynamic decisions. This is in stark contrast with the BSP model, where a static schedule is embedded into the expression of the algorithm, explicitly specified by the programmer in the code flow of the program. Although the BSP approach elim-

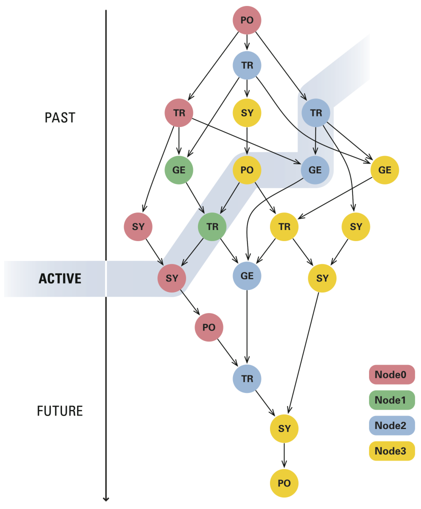


Figure 4: The PaRSEC runtime walks the DAG using a concise representation that instantiates only the relevant tasks at each computing node. Only the active, local tasks need to be stored and considered.

inates the need for management and scheduling overhead, and is therefore very efficient for instruction level handling, it lacks in flexibility necessary to adapt to runtime conditions.

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

4 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 central component of the system, the *runtime*, combines the task and dataflow information of the source program with supplementary information provided by the user — such as the distribution of data, or hints about the importance of different tasks — and orchestrates the execution of the tasks on the available hardware.

From a technical perspective, PaRSEC is an event driven system. When an event occurs, such as the completion of a task, the runtime reacts by examining the dataflow from this task 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, 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 the hardware resources to be devoted to the execution of the application code.

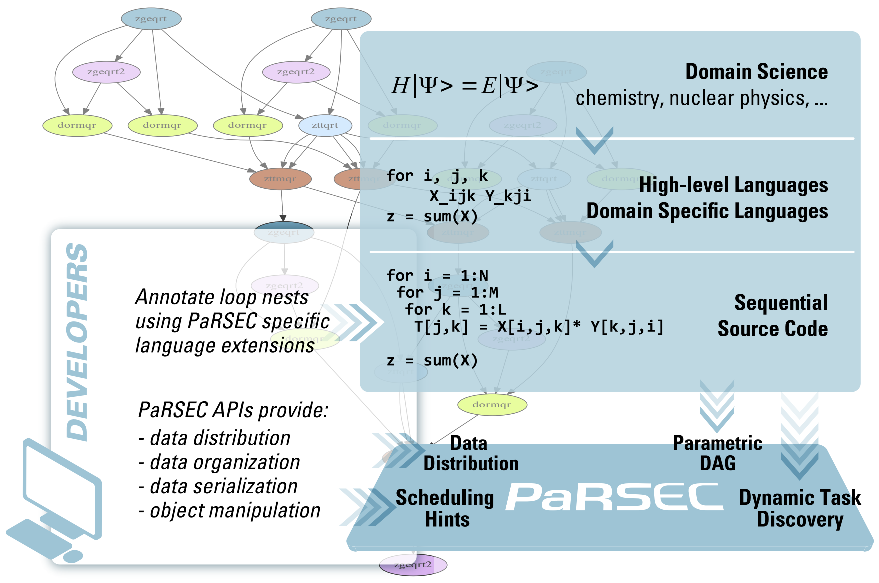


Figure 5: The PaRSEC environment: at a high level, productivity tools convert domain specific codes into a dataflow representation. The dataflow representation is combined with the PaRSEC runtime library to form the versatile application representation.

Due to the dataflow representation, the communications become implicit and thus are handled automatically and as efficiently as possible by the runtime. Specifically, in the PaRSEC model, data exchange is not explicitly coded by the developers into their application, as in MPI, but implied in the dataflow representation of the application. Given that PaRSEC is aware of the dataflow representation of the application, 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 non-blocking communication and advanced collective communication algorithms to achieve communication-computation overlapping and hide significant parts of the communication overhead.

Scheduling of tasks within each node is also one of the responsibilities of the runtime. Specifically, as tasks complete, they generate data that enables the execution of other tasks. The runtime keeps track of the tasks that have completed, discovers the tasks that can execute next, and decides which hardware resources (i.e., CPU cores, accelerators, co-processors) should be devoted to each new task. Consequently, applications that make use of PaRSEC (i.e., PaRSEC-enabled applications) 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.

5 High Productivity Ecosystem

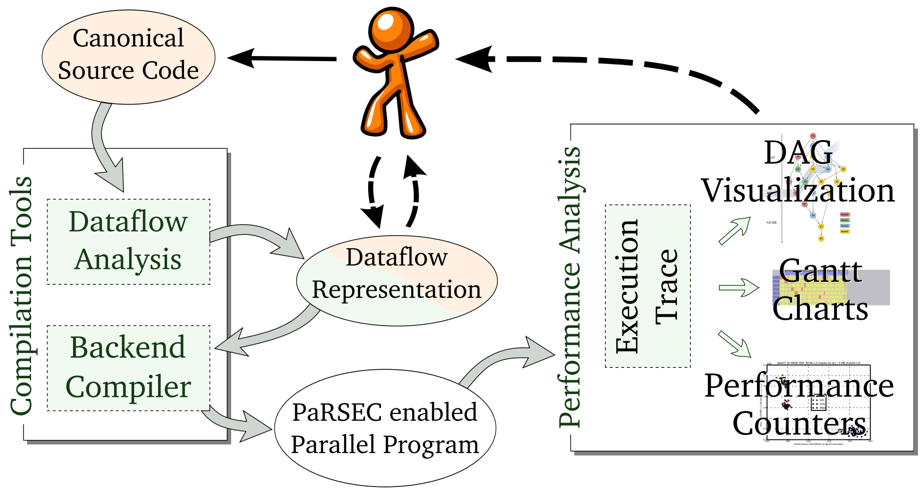


Figure 6: PaRSEC productivity tools: a rich ecosystem of tools support the developer in converting legacy applications and high level programming into a dataflow representation. The programmer has access and can directly alter the dataflow representation. A variety of debugging and performance analysis tools are available to investigate the correctness and performance of the program dataflow.

Given the ongoing increase in system complexity, it is clear that any viable programming model will need to help developers achieve the best performance possible, even while helping them to keep their efforts below a reasonable threshold. Arguably then, for emerging system and programming paradigms, ease of use and development are not only relevant metrics, but are often as important as achievable performance. A system that does not 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. 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 direct acyclic graph of tasks and an associated dataflow, and this implies 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 have created, and are actively extending, tools that aim to help developers make their codes "PaRSEC-enabled". Since the users of our system are parallel application developers, the primary tools we have created for interfacing with the users are analysis and compilation tools. Specifically, our system includes two compilation tools, the front end compiler and the back end compiler.

The front end compiler aims to make it as easy as possible to use PaRSEC and to enable a seamless upgrades of legacy software, such as the linear algebra package PLASMA, which were created for multi-core processors, but not for distributed heterogeneous systems. The input to the front-end compiler has to 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 tasks of the input program and the dependencies between them in a symbolic and problem size independent way [4].

However, the canonical form of the source code limits the input to affine codes. That is, loops with bounds that may 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. While several interesting problems meet this limitation (e.g., dense linear algebra, tensor contraction, etc.), not all parallel applications can be expressed as such. For this reason, PaRSEC allows the human developer to alter the dataflow representation of the program, or even write it completely by hand. This way, the developer can go beyond the limitations of the front-end compiler and trade simplicity for expressivity.

There is a significant difference between this approach and what other task-based runtimes typically do. In other task-based runtimes the execution flow is directly derived from the 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 a scalable task discovery and scheduling in distributed environments.

Data Affinity and Movement: As communications are implicit in the dataflow model, the expression of the algorithm 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 portability of codes. The application developer first focuses 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 specifics of the algorithm. Finally, if required, the developer can fine tune communication volume and pattern 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 dataflow representation of the input program 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, i.e. this generated code can be compiled and linked with the run-time library of PaRSEC using a traditional C compiler like 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 is similar to the familiar fact that, 40 years after the creation of C, expert programmers still write critical code snippets in assembler. We believe that offering application developers a choice in the level of complexity vs. expressivity, and 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 move from legacy programming models to dataflow representations are essential, it is also important to provide tools for debugging and analysis that are adapted to the new model. An application programmer may want to verify that a parameterized task graph, whether automatically or manually written, correctly describes the data dependencies of a given program. For this reason we provide a tool for generating and displaying the DAG of the application at the level of detail specified by the developer. Figure 4 shows the dependency graph of an algorithm with four different kernels. This representation displays useful information for the developer. For example, the shape of the graph informs the user about the length of the critical path of the algorithm, 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 obtain this result, several solutions are possible: rethink or change the algorithm to minimize or remove the need for synchronizations, or generate smaller grain computational tasks. The first solution might be impossible, and the second solution 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 does not provide helpful information regarding the efficiency of the algorithm in exploiting the resources of the system. One common way to study performance of parallel applications is measuring 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, the dataflow representation of an algorithm removes most, if not all, of the synchronization points. Therefore, in the case of data-driven, task-based execution, two things need to be studied. The first is the performance of the tasks themselves, which is observed by collecting statistics such as time spent, cache misses, etc. The second is the efficiency of the scheduling for ensuring that the correct choices have been made for the given DAG. PaRSEC enables the developer to collect this kind of detailed information about the tasks and the scheduling so that the behavior of the system can be analyzed, understood, and tuned.

One has to make sure that the choices are an efficient compromise between keeping data locality for local performance and maximizing parallelism within the DAG. In order to validate those choices, PaRSEC provides the ability to visualize execution traces as Gantt diagrams, such as those shown in the Figure 1. Execution traces, coupled with the DAG representation of the dataflow, 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 is also a clear need for new tools to explore the complementary aspects of data and/or task distribution. Such tools will help programmers determine the efficiency of the tasks and the way in which they 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.

6 Conclusion

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 last two decades, the complexity and heterogeneity of new hardware has relentlessly eroded its effectiveness. In spite of 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, as well as falling short on both energy efficiency and resilience. The dataflow-driven programming paradigm described in this paper, together with a corresponding runtime, provides an exciting opportunity to close this gap, and to increase code portability at the same time. The era of dynamic and heterogeneous hardware that is 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 the runtime behavior, should also be developed. But similar twists and turns on the narrow climb to Exascale is likely to provide many more such exciting perspectives and challenges.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous mu lticore architectures. *Concurrency Computat. Pract. Exper.*, 23(2):187–198, 2011. DOI: 10.1002/cpe.1631.

[2] A. J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757 –763, Oct. 1966.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J.J. Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach. *Scalable Computing and Communications: Theory and Practice*, pages 699–733, January 2013.

[4] Michel Cosnard and Emmanuel Jeannot. Automatic Parallelization Techniques Based on Compact DAG Extraction and Symbolic Scheduling. *Parallel Processing Letters*, 11:151–168, 2001.

[5] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 394–401, 2009.

[6] Innovative Computing Laboratory. D-PLASMA. `http://icl.cs.utk.edu/dplasma/`.

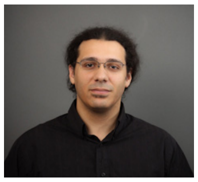
[7] Innovative Computing Laboratory. PaRSEC. `http://icl.cs.utk.edu/parsec/`.

[8] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[9] J. Planas, R. M. Badia, E. Ayguadé, and J . Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094 342009106195 .

[10] Herb Sutter. Welcome to the jungle. `http://herbsutter.com/welcome-to-the-jungle/`, August 2012.

George Bosilca is an Assistant Research Professor at the Innovative Computing Laboratory, University of Tennessee. He received his Ph.D. from University of Paris in 2004 working on scalable and resilient runtimes for parallel architectures. His research interests range from low-level communication protocols to high-level constructs to support novel parallel programming paradigms, everything with potential to diminish the gap between peak and practical performance of parallel machines or to increase the parallel developers productivity. He is a main contributor in several widely used projects Open MPI, PaRSEC and ULFM. He has co-authored over a hundred papers and journals.

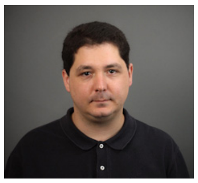


Aurelien Bouteiller is a researcher at the Innovative Computing Laboratory, University of Tennessee. He received his Ph.D. from University of Paris in 2006 on the subject of rollback recovery fault tolerance. His research is focused on improving performance and reliability of distributed memory systems. He investigated checkpointing approaches, Algorithm Based fault tolerance, mechanisms to improve communication speed and balance of many-core clusters, and employing emerging data flow programming models to negate the raise of jitter on large scale systems. These works resulted in over thirty international publications and three distinguished paper awards from IPDPS and EuroPar. He is also a contributor to Open MPI, PaRSEC, and a leading designer of the fault tolerance MPI standardization effort.

Anthony Danalis is currently a Research Scientist II with the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests come from the area of High Performance Computing. Recently, his work has been focused on the subjects of Compiler Analysis and Optimization, System Benchmarking, MPI, and Accelerators. He received his Ph.D. in Computer Science from the University of Delaware on Compiler Optimizations for HPC. Previously, he received an M.Sc. from the University of Delaware and an M.Sc. from the University of Crete, both on Computer Networks, and a B.Sc. in Physics from the University of Crete.

Jack 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, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; and in 2011 he was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

Mathieu Faverge is an Assistant Professor at the Bordeaux Institute of Technology, France. He received a PhD degree in Computer Science from the University of Bordeaux 1, France. His main research interests are numerical linear algebra algorithms for sparse and dense problems on massively parallel architectures, and especially DAG algorithms relying on dynamic schedulers. He has experience on hierarchical shared memory, heterogeneous and distributed systems and his contributions to the scientific community includes efficient linear algebra algorithms for those systems.



Thomas Herault is a Research Scientist at the Innovative Computing Laboratory, University of Tennessee. He received his Ph.D. from the University of Paris-Sud in 2003, working on resilience in self-stabilizing systems. His research interests include fault tolerance, high-performance computing and distributed algorithms. He contributes to several widely distributed open source software (in particular PaRSEC and ULFM), applying his research to the service of a more efficient, and reliable, usage of high performance computers. He is co-author of more than sixty papers in international conferences, and journals.