PTG: an abstraction for unhindered parallelism

Anthony Danalis, George Bosilca,
Aurelien Bouteiller, Thomas Herault
University of Tennessee

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester

ABSTRACT

Increased parallelism and use of heterogeneous computing resources is now an established trend in High Performance Computing (HPC), a trend that, looking forward to Exascale, seems bound to intensify. Despite the evolution of hardware over the past decade, the programming paradigm of choice was invariably derived from Coarse Grain Parallelism with explicit data movements. We argue that message passing has remained the de facto standard in HPC because, until now, the ever increasing challenges that application developers had to address to create efficient portable applications remained manageable for expert programmers.

Data-flow based programming is an alternative approach with significant potential. In this paper, we discuss the Parameterized Task Graph (PTG) abstraction and present the specialized input language that we use to specify PTGs in our data-flow task-based runtime system, PaRSEC. This language and the corresponding execution model are in contrast with the execution model of explicit message passing as well as the model of alternative task based runtime systems. The Parameterized Task Graph language decouples the expression of the parallelism in the algorithm from the control-flow ordering, load balance, and data distribution. Thus, programs are more adaptable and map more efficiently on challenging hardware, as well as maintain portability across diverse architectures. To support these claims, we discuss the different challenges of HPC programming and how PaRSEC can address them, and we demonstrate that in today's large scale supercomputers, PaRSEC can significantly outperform state-of-the-art MPI applications and libraries, a trend that will increase with future architectural evolution.

1. INTRODUCTION

High performance parallel computing is the vehicle through which scientific discovery through simulation is achieved. Over the years, parallel machines worthy of the supercomputer title have become increasingly more complex, while the application development process and paradigms have generally remained the same. As highlighted in [34], the isomorphism of parallel architectures over the last couple of decades is one of the major factors for the hegemony of the flat MPI+X type of programming paradigm. Such a programming model exposes the underlying architecture as a simplistic two dimensional space; one dimension being inter node with MPI as the major programming paradigm, and one dimension being intra node where a complementary, shared memory, programming paradigm (e.g., OpenMP, OpenACC, OpenCL, CUDA) is involved. However, the drastic increase in the number of potential computational resources, supported by the growth of heterogeneity at all the levels of the parallel architecture (memory, network, computational resources), highlight a need for a revolutionary change in the way algorithms and applications are programmed, an approach completely orthogonal to the current MPI+X. The need to be able to expose a dynamic degree of parallelism, one that can be adapted to the underlying capabilities of the execution architecture must become one of the drivers of programming paradigms, language, and runtime research. Looking more carefully, we can identify multiple sources of complexity facing the application developers:

The number of processing units (sockets and cores per socket) has been increasing, providing a higher raw computational performance, but demanding from the applications to expose an increased level of parallelism. This is not always feasible, but even when it is, efficient execution requires a mapping of work onto compute resources that avoids idle resources and balances the workload. This challenge, known as scheduling, has been an active field of research for several decades and involves particularly difficult problems.

The memory hierarchy has become more complex. In modern hardware a core may access data from its L1 cache, or caches that it probably shares with other cores in the same socket, or parts of the RAM that may be controlled by its local memory controller, or a remote controller. While any data that resides in local memory – at any level of the hierarchy – can be accessed by an application that is oblivious to the hierarchy, such an application will most certainly face increased latencies in comparison to one that is aware of the memory hierarchy. On the other hand, developing applications that take advantage of the memory hierarchy is

This material is based upon work supported by the Department of Energy under Award Number DE-SC0010682 and by the National Science Foundation under Grant Number CCF-1244905

significantly more complex than treating the local memory as a flat address space with uniform access latencies.

When remote data storage is considered, the picture becomes even more complex. Some software solutions (most notably HPF [26]) attempt to remove this source of complexity by providing the application developer with a view of the globally distributed memory as a single shared address space. While this level of abstraction seems desirable, HPF and similar solutions failed to gain traction in the HPC community [24]. Other solutions, such as the PGAS languages [11, 19, 12, 28], offer a partitioned view of the global address spaces. In this abstraction, the application developer is aware of the distribution of memory between nodes and makes explicit references to remote memory, but does so using language primitives and abstractions.The least abstract programming model is that of explicit message passing using communication libraries (such as MPI) at the application layer. At the time of this writing, this model – where the whole burden of managing the communication and load balancing of an HPC application falls on the application developer – is by far the most widely used programming model for distributed memory machines.

Heterogeneity of computing resources adds yet another source of complexity in modern hardware. In supercomputers that contain processing resources other than traditional CPUs – GPUs, APUs, Xeon Phi's, etc. – applications must manage these resources in order to harness the full processing power of the computer. However, accelerators often have a separate address space, so duplicate buffers must be kept, and explicit copying must be performed by the application. Synchronization, and, most importantly, load balancing between the CPUs and the accelerators is also necessary and can prove to be challenging for achieving high performance execution. Furthermore, load balancing requires a different solution in every hardware environment with a different balance between the processing power of the CPUs and the accelerators.

Despite all these challenges, application developers still develop applications using a Coarse Grain model where parallelism, message passing, and resource management are explicitly handled at the application layer. This has been the case because, until now, doing so delivered a reasonable level of performance on large scale heterogeneous supercomputers. In this paper we present a different approach to programming hybrid architectures, reviving the concept of data-flow. We demonstrate that such a modernized approach, presents a highly efficient alternative programming paradigm able to tackle all the above mentioned challenges. The underlying infrastructure is a task-based runtime called PaRSEC. PaRSEC is an engine for scheduling tasks on distributed hybrid environments. It offers a flexible API to develop moldable domain specific languages, minimizing the disturbance on application developers by allowing them to shift their focus from repetitive architectural details toward meaningful algorithmic improvements. By calling attention to one of the domain specific languages supported by PaRSEC, the Parameterized Task Graph (PTG), we will stress the language features supporting portable efficient application development that can drastically outperform MPI-based state-of-the-art applications at scale.

2. CONTROL-FLOW & DATA-FLOW BASED PROGRAMMING

In the era of serial execution and simple processing units, implementing an algorithm as a computer program meant specifying the statements that the program must execute and the sequential order in which those statements should execute. In other words, application developers dictated the control-flow of their program. With the advent of more advanced hardware capabilities, such as vector instructions, super-scalar, out-of-order, and Very Long Instruction execution, the need for explicitly inferring and analyzing the data-flow of a program arose. Data-flow analysis became necessary because such hardware features enable different instructions to execute concurrently and independently, i.e., instruction level parallelism. As a consequence, the control flow of the original serial program is no longer strictly obeyed – some of the instructions execute out-of-order – but the application data-flow must be obeyed in order to preserve the semantics of the algorithm. However, instruction level parallelism was implemented by the hardware and the backend of compilers and applications remained serial programs with an explicit control-flow and an implicit data-flow.

As the need for ever-higher performance escalated, architectures with multiple processing elements appeared. Such machines were capable of executing different programs, or instances of the same program, in parallel. Even further need for performance brought us clusters with distributed memory. Several programming paradigms were proposed over the decades to program complex algorithms on parallel machines, but the one that prevailed was Coarse Grain Parallelism (CGP) with explicit message passing (what most HPC developers colloquially refer to as "MPI"). While this is the de facto standard, and most researchers and practitioners in the field are comfortably accustomed to it, it is a peculiar turn of events.

Parallel applications written using CGP are not structured in a fundamentally different way than serial applications. In essence, an "MPI application" consists of a long serial code, with the same control flow structure that a purely serial application would have, a few calls to the communication library, and potentially some branches that differentiate the behavior of the code for different processes of the parallel program. Research [10, 16, 17] has aimed at improving communication computation overlapping for MPI programs. Sancho et al. [31] studied the potential of overlapping computation with communication in large-scale codes and found a high degree of potential overlap. Parts of the application could be re-factored so the work is performed between the initiation of a data transfer and the point in the code where the transferred data is needed. While this appears beneficial for the communication latency, it is detrimental to the programming paradigm. If there are parts of the application that could execute when the application is waiting for a particular data transfer, why is this information not available in the original formulation of the code? Why does the original code specify a control-flow between parts of the application that do not depend on one another? And why does a third party researcher have to discover these independent code regions and propose that they be used to overlap communication with computation, instead of them being immediately available to the execution runtime?

In a data-flow programming model the units of work in a program are not ordered as a sequence, but rather are organized on a graph where each node is a unit of work and each edge defines precedence constraints between the two nodes that it connects, due to data-flow between them. If we define a procedure to be the unit of work, we arrive at a practical and efficient programming model. In this model, each procedure that is considered as a unit of work becomes a *task* and programs are collections of *tasks* and the data-flow between them. Clearly, a runtime engine capable of scheduling a program represented as tasks (with dependencies to other tasks), would automatically overlap communication and computation by executing the next available task at any given time. This is because in a data-flow programming model, tasks are not connected by loops and if-then-else branches, but by incoming and outgoing data. When a task is done and its output data is available, the program itself contains the information regarding which task can use this data to start its execution. The artificial and unnecessary constraints imposed by the control flow of CGP programs become irrelevant in this context and no deep code inspection is required to discover which parts of the code (i.e., tasks) are connected and which are independent.

In the following section, we present the Parameterized Task Graph (PTG), which constitutes the center piece of the data-flow based programming model that is supported by our task scheduling runtime, PaRSEC.

3. PTG SPECIFICATION IN PaRSEC

The original concept of the Parameterized Task Graph was described by Cosnard et al. [15, 14]. The main idea was that a program is written as a collection of task classes. The representation is problem size independent in that each task class can be described using a (small) finite amount of bytes, regardless of how many task instances of this task class will be executed when the program runs. We derived our specification from this original concept by extending it with additional constructs, and providing extra information allowing for a more detailed understanding of the unfolding of the application execution and the dependencies between consecutive algorithms.

Each task class contains information that enables the creation and execution of the task instances. In Figure 1 we show a sample PTG for a ping-pong operation that contains two task classes. In the following text we use this example to explain the different fields that each task class contains.

The class name and the symbolic parameters that are used in the other fields of the class. In this example the names of the two classes are "`PING`" and "`PONG`" and both use a single parameter called "`s`".

The valid ranges of values for each of the class parameters. We call this set of parameter ranges the *execution space* of the task class. In the example of Figure 1 "`PING`" has an execution space that is larger than that of "`PONG`" by one, so it will generate one more task instance, "`PING(max_steps-1)`." In contrast with the description of the PTG by Cosnard et al., PaRSEC allows for arbitrary expressions in the bounds of the parameters (including calls to external pure functions), accepts non-linear steps, and uses the *execution space* more

```
1  PING(s)
2   s = 0..max_steps-1
3   : A(s)
4   RW    A0 <- A(s)
5             -> A0 PONG(s)
6   READ A1 <- (s != 0) ? PONG(s-1)
7  BODY verify_response(A0, A1); END
8
9  PONG(s)
10  s = 0..max_steps-2
11  : A(s+1)
12  RW    A0 <- A0 PING(s)
13            -> A1 PING(s+1)
14 BODY /* do nothing on data */ END
```

Figure 1: PTG for ping-pong.

as an iterator than as a loop. The only limitation imposed by PaRSEC is that these bounds must be identically computed by all participants in a distributed execution. In the rest of this document, when we use the term PTG, we will refer to the augmented form accepted by PaRSEC.

An affinity field. This field is shown in lines 3 and 11 in the example. It is a parameterized symbolic reference to a data element and specifies that the task instance with the given parameters should execute on the same physical location as where the specified data resides. In our example, task instance "`PONG(17)`" will execute on the node where the data element "`A(18)`" is stored. We discuss this field and its scheduling implications in greater detail in Section 5.

A set of precedence constraints. PaRSEC schedules tasks by following 1) true data-flow edges and 2) precedence constraints that do not involve data (for satisfying anti-dependency constraints and for user-defined execution throttling purposes). A data-flow edge may involve reading from initial data (as shown in line 4), or writing to final data. Alternatively, a flow could specify the transmission of data to another task class, as shown in line 5, or the reception of data from another task class as shown in line 12. In addition, all flows may be conditioned using an arbitrary logical expression, as shown in line 6. A task instance of any task class can be used to specify the peer in a data-flow. Finally, the expression that specifies which task instance is the peer can be arbitrary, including calls to external C functions.

A set of code regions contained by the keywords "`BODY`" and "`END`" as shown in Figure 1, lines 7 and 14. Each such region may include source code that will execute verbatim on the CPU, or accelerators of the node where this task instance will execute. In other words, this segment contains the work performed by this task instance, typically using the input data to produce the output data of the task instance.

As is evident by this specification, the PTG describing a program is independent of the problem size that will be solved by the program. This concise representation of the task graph is a feature unique to PaRSEC among all task-based execution engines that are currently actively developed. Other systems, such as TBB [2], CnC [1] and the OpenMP-4.0 Task API offer parameterized task dependency specifications at the source code level. However, at run time

these systems enumerate tags and store information per tag to differentiate between different task instances. These storage requirements grow with the number of task instances, and constructing the dynamic DAG of dependencies requires lookups into these structures. PaRSEC maintains the symbolic representation at run time and uses only that to evaluate the ancestors and descendants of each task instance.

4. PTG AS A PROGRAMMING PARADIGM FOR EXASCALE

In the following paragraphs, we contrast the PTG against two other programming models. First, the Coarse Grain model with explicit message passing, which is currently the de facto standard in HPC. Second, the approach taken by all other actively developed task scheduling runtimes, where the DAG of task instances is constructed, stored in memory, and used by the runtime to make scheduling decisions. In the rest of this document we will borrow the term Dynamic Task Graph (DTG) from Adve et al. [3] to refer to this approach.

In an attempt to classify the PTG of different programs based on how complex it must be in order to express a given algorithm, we realize that there is a spectrum of categories. Simpler PTGs can be scheduled efficiently by the runtime, but can express a limited set of algorithms. Conversely, more complex PTGs can express arbitrarily complex algorithms, but incur additional runtime overhead.

4.1 PTG: Performance versus Expressibility

Programs that contain only affine loop nests and array accesses result in PTGs that are fully algebraic. That is, not only does the representation of the PTG occupy $O(1)$ space, but the analysis of the data-flow of a task instance, at runtime, takes $O(1)$ time. Such PTGs can be generated automatically from compilation tools that perform polyhedral analysis, such as the front-end compiler of PaRSEC [9], but the set of algorithms that can be expressed using only affine loops and array accesses is limited.

Close to the other end of the spectrum, one can create PTGs for highly dynamic applications. Such PTGs cannot encapsulate the behavior of the program a priori, since it is dynamic, and therefore must keep dynamic meta-data structures that are updated and consulted at run-time to enable the program to change behavior based on dynamically generated program data. Unlike the fully algebraic PTG of the affine codes, fully dynamic PTG can spend an unbounded amount of time building and traversing dynamic meta-data in memory.

In between these extreme cases there are PTGs whose complexity matches that of applications that are dynamic, but obey some patterns. An example from this category is a binary tree reduction algorithm. The shape and size of the tree depends on the problem size in ways that cannot be expressed with a closed formula. Figure 2 shows the part of the PTG for a binary reduction operation that specifies the most common task. Although the PTG is not affine, it still contains a small number of task classes, and most task instances will be of the class that describes a node in the middle of the tree (i.e., the type of task that reduces its two inputs and passes the result to the next task). These tasks have well formed inputs and outputs that can be computed programmatically once the parameters are instantiated by runtime computed static expressions.

4.2 Parameterized Task Graph (PTG) versus Dynamic Task Graph (DTG)

As one can observe, in Figure 2, the execution space of the task class is dynamically determined through calls to the arbitrary functions `count_bits()`, `compute_offset()`, and `log_of_tree_size()`, and so are the data-flows. However, the task class has a very limited set of possible data-flows, which are known a priory. The total number of task classes is also known a priori (i.e., there are exactly three task classes in the PTG of the binary tree reduction). This is in contrast with the Dynamic Task Graph (DTG) approach taken by all other actively developed task execution runtimes. In the DTG approach, the DAG connecting the individual task instances for a given execution is completely unknown before it is discovered, expanded in memory, and traversed at runtime in order to make scheduling decisions. This has several drawbacks that the PTG is not susceptible to.

The memory requirements for storing a DTG grow with the problem size. In contrast, the PTG for the binary tree reduction will contain the same three task classes, and those task classes will require the same amount of memory whether the tree being reduced has two leaves, or two million leaves.

A skeleton program that "submits" the tasks (i.e., inserts them in a queue) must be used in runtimes following the DTG approach, so that the runtime can detect the tasks that should be scheduled and the dependencies between them. In the case where PaRSEC uses a PTG with lookups to meta-data a skeleton program must also run at the beginning of the execution to populate this meta-data. However, in the case of DTG, the skeleton does not merely populate some meta-data that depend on input parameters. DTG based runtimes assume nothing about the shape of the DAG. Instead, they need to record the pointers to the program data that is read and modified by each task. Then the runtime must use these pointers, for each newly discovered task, to identify the previous task that read or modified the same data, in order to build the DAG that represents the execution of the program. This is clearly a more time-consuming operation than what is needed in PaRSEC.

A fixed size window of task instances that is stored in memory at any given time can be used by DTG runtimes to throttle the task submission, so that the memory consumption is remedied. However, this does not come without a price. Since the skeleton program is a serial program with a fixed control flow, an arbitrarily set window size W will prevent the discovery of available tasks when these tasks happen to be more than W tasks away, based on the control flow ordering.

Consider the pseudo-code shown in Figure 3. This code defines W chains of tasks. Each chain is completely independent from the others, but within each chain, each task has to wait for the one before it. This behavior is depicted, graphically, by the DAG of this program, shown in Figure 4.

```
BT_REDUC(tree, step, i)
  tree_count = count_bits(NT)
  tree  = 1 .. tree_count
  max_step = log_of_tree_size(NT, tree)
  step  = 1 .. max_step
  i  = 0 ..(1<<(max_step-step))-1
  offset = compute_offset(NT, tree)

  : dataA(offset+i*2,0)

  READ A  <- (1==step) ? A REDUCTION(offset+i*2)
          <- (1!=step) ? B BT_REDUC(tree,step-1,i*2)
  RW   B  <- (1==step)                      ? A REDUCTION(offset+i*2+1)
          <- (1!=step)                      ? B BT_REDUC(tree, step-1, i*2+1)
          -> ((max_step!=step) && (0==i%2)) ? A BT_REDUC(tree, step+1, i/2)
          -> ((max_step!=step) && (0!=i%2)) ? B BT_REDUC(tree, step+1, i/2)
          ->  (max_step==step)              ? C LINEAR_REDUC(tree)
BODY int j; for(j=0; j<NB; j++){ REDUCE( A, B, j ); } END
```

Figure 2: PTG for main task class of binary reduction.

```
for(i=0; i<W; i++)
  Task1( RW:Data[i][0] );
  for(j=1; j<c*W; j++)
    Task2( R:Data[i][j-1], W:A[i][j]);
```

Figure 3: Code that creates independent chains.

A scheduling engine that follows the control flow of the serial program of Figure 3 is unable to discover any of the available parallelism, as it has to wait until the completion of the sequential execution of each chain of $c*W$ tasks before it can discover the tasks of the next chain. In contrast, PaRSEC will immediately recognize that each chain is independent from the rest, and schedule them in parallel.

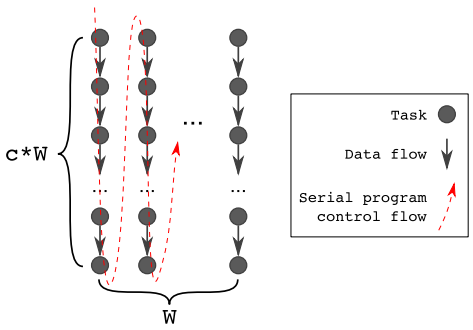

Figure 4: DAG for the independent chains.

The PTG representation of this program is shown in Figure 5. The lack of dependency between chains can be seen in the PTG, since there is no data flow from any task $Task1(i)$ or $Task2(i,j)$ to a task $Task1(i')$ or $Task2(i',j')$ for $i \neq i'$. That is, no precedence edge crosses chains. As a consequence of this difference in scheduling decisions, a system that uses a Dynamic Task Graph and a window W will require a number of steps to complete the execution of the example program equal to:

$$S_{DTG} = c * W + (W-1) * (c-1) * W$$

```
Task1(i)
 i = 0..W-1
 : Data(i,0)
 A <- Data(i,0)
   -> A Task2(i,1)
BODY ... END

Task2(i,j)
 i = 0..W-1
 j = 1..c*W-1
 : Data(i,j)
 A <- (j == 1)     ? A Task1(i)
   <- (j > 1)      ? A Task2(i,j-1)
   -> (j < c*W-1)  ? A Task2(i,j+1)
   -> Data(i,j)
BODY ... END
```

Figure 5: PTG for the independent chains.

This is the case because the first chain will take $c*W$ steps to execute and then all remaining $W-1$ chains will have W tasks that were discovered and executed in parallel with the previous chain, and $(c-1)*W$ tasks that will execute serially. Note that this will be the case even if there are P available processors ($P \leq W$). In contrast, PaRSEC can keep all P processors busy at all times and finish the execution in the following number of steps:

$$S_{PTG} = \frac{c * W^2}{P}$$

Thus, on a machine with P processors, PaRSEC will execute this code with a speedup of $O(P)$:

$$Speedup = \frac{S_{DTG}}{S_{PTG}} = P * (1 - \frac{1}{c} + \frac{1}{c * W})$$

It is unlikely that a production application would exhibit a pattern as adversarial as this, and reach such a dramatic asymptotic difference between a PTG based runtime and a DTG with fixed window system. However, the demonstration above serves as a theoretical proof that some potential parallelism can remain hidden to the DTG exploration of the DAG when considering reasonable memory limitations. In

contrast, the PTG model always discovers the same amount of parallelism as a DTG approach with an infinite window, but without the heavy memory requirement, or the high overheads of the sequential program skeleton.

Adherence to control flow, or freedom from it thereof, is another major difference between PaRSEC's meta-data driven PTG and alternative approaches that build the Dynamic Task Graph. In 1993, Wolfe [33] proposed a new loop type, `DOANY`, to capture the semantics of a loop whose iterations can execute in *any* order, because they do not depend on one another in a specific way, but they must run in *some* order because their output must be progressively processed by each new iteration. A good example of this behavior is a loop that performs multiple matrix-matrix multiply operations ($C = C + A * B$) such that all iterations use the same output matrix C (and no iteration uses C as an input matrix A or B). The `DOANY` loop has not been adopted by any popular programming language, because, in a control-flow dominated world, there is not much benefit from having a loop like this. However, a program expressed as a PTG can declare the loop iterations as independent tasks that send their output to a reduction tree that performs the $C = C + C_i$ operation for all task outputs C_i. This case is not just of theoretical importance. This structure of chained matrix-multiply operations appears multiple times in the CCSD code of the NWChem [32] package, a state-of-the-art software package for computation chemistry. As a consequence, a modified version of the NWChem software that runs over PaRSEC takes advantage of this flexibility and achieves better performance than the vanilla NWChem.

4.3 PTG versus Coarse Grain Parallelism with explicit message passing

Most HPC applications are structured as serial codes that include communication and synchronization calls to perform explicit message passing. Most such programs execute identical instances of the code across all nodes of a parallel machine and the more sophisticated applications have `if-then-else` branches that depend on process id (rank) to express divergent behavior between different nodes. Such applications are colloquially referred to as "MPI applications" and many computational scientists use the term "MPI programming paradigm." The elevation of the communication layer to a programming paradigm status in the mind of practitioners is telling. The actual programming paradigm is Coarse Grain Parallelism with explicit message passing (CGP), since the whole application is parallelized and the message passing is explicitly inserted in the application layer.

Control flow based limitations plague the CGP model, as we discussed in 2. Since the application is structured as a serial code, it has to order its work in units of computation that follow one another even if they are not dependent on each other. When such independent units of work exist, they will often be used to overlap communication and computation. However, they cannot be used to dynamically adjust the level of parallelism. In the PTG programming model, if at some point in the execution of an application the available parallelism increases, more tasks become available and the runtime can efficiently employ more computing resources. In the CGP model, the number of processes is fixed and even in the case of hybrid MPI +threads applications, a dynamically adjusting number of threads have yet to show significant penetration in the community.

Idle time is almost inevitable when a parallel application is designed at a coarse grain level, and there are logical stages that need to complete for the application to move to the next stage. Figure 6 shows, on the top, the execution trace of a QR factorization implemented in ScaLAPACK, and on the bottom, the same factorization in PaRSEC (using the same problem size, and number of processors). In this figure, gray areas represent idle time and colored areas represent different types of work. Clearly, the coarse grain parallelism used for ScaLAPACK led to barriers between logical stages of the code, and these barriers led to idle time due to load imbalance. Expressing such a dynamic and opportunistic load balance and ordering of computations in the CGP programming model would be extremely tedious. As a consequence, although the ScaLAPACK library has gone through many optimization cycles in its decade long existence, it is not until embracing the data-flow model that programmers were able to refactor the algorithms with a finer grain parallelism, at a reasonable engineering cost. The refactored code is simple enough to be expressed clearly, yet it permits decoupling the load balance from the programmatic expression of the algorithm. In the opportunistic execution over PaRSEC, the idle time is greatly reduced, since the runtime can execute any available task, and as a result the overall execution time is improved significantly.

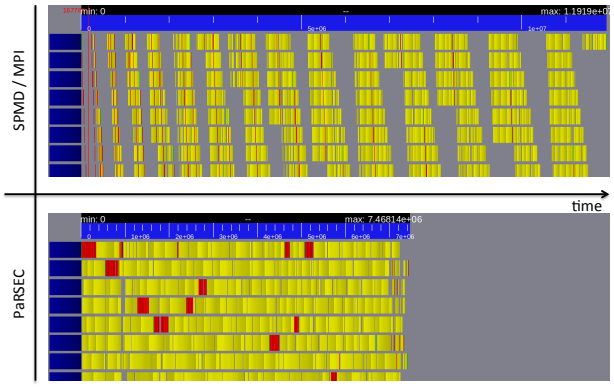

Figure 6: Execution trace of QR factorization.

Noise in the execution environment – even at very low overheads – has been shown to have a disproportionately large detrimental effect to the execution of parallel applications [20, 21]. Just as in the case of idle time, PaRSEC has no predetermined schedule, or stages, so it can absorb noise by dynamically adapting the execution of tasks to the availability of compute resources.

Communication-computation overlapping has been a topic of research for coarse grain applications, and developers have to take special care to achieve this behavior in their codes (see "Issues" in [27]). In the case of the PTG, communication is implicit. As soon as the execution of a task completes, its output data can start being transferred, while other available tasks are being scheduled by the runtime. Thus, overlapping takes place automatically.

The memory hierarchy in modern machines is complex and has multiple layers, from shared and private caches to main memory that exhibits Non-uniform memory access (NUMA). CGP applications are often oblivious to this structure; a typical deployment runs one process per core of the machine, with little regard to the hardware topology. Even when a CGP application embeds some support for a multi-tiered memory hierarchy, the resultant program loses in portability: the number of levels and the neighboring relationship between processes is hardcoded. More advanced designs include threads through solutions such as OpenMP, but are thereby dependent on the capacity of the compiler to generate code that employs the memory hierarchy efficiently; a challenge that has proven difficult in practice. In contrast, the PTG representation does not hardcode the division of the parallel workload onto the resources. Instead, the scheduling of the expressed parallelism is delegated to the runtime which manages all compute resources outside the view of the programmer. Since PaRSEC makes all the dynamic scheduling decisions, it is fully aware of the resource each task executed on. Given this information and the data-flow information from the PTG, PaRSEC is able to schedule tasks so that reuse of local memory resources is achieved, and cross traffic between memory banks is only allowed when some computing unit is idle.

Heterogeneity of computing resources creates management challenges for coarse grain applications and makes the challenge of load balancing even more difficult. In the case of PaRSEC, both challenges are mitigated by the runtime, since the management of data is performed implicitly by the runtime and load balancing is achieved through dynamically adapting the task scheduling to the available workload and available computing resources.

Programmability is a subjective issue. Implementing a parallel application using an imperative language such as C or FORTRAN in conjunction with MPI has a low startup overhead since these are well known technologies. In contrast, PaRSEC is new in the community and so is the concept of writing large applications using a PTG. However, implementing a parallel algorithm with particularly complex data-flow patterns can be easier if the PTG approach is used rather than coarse grain parallelism. A case in point: the Hierarchical QR Factorization Algorithms presented in [18] have only been implemented in PTG, because the authors deemed the PTG programming model to be easier for this problem than CGP. The importance of HQR becomes evident when one compares the superior performance of this algorithm when compared to the traditional QR factorization – as demonstrated in Figure 7

It may initially seem easier to write a parallel application using the CPG programming model than using a PTG. However, writing a CPG application that addresses all the aforementioned challenges as effectively as they are addressed by PaRSEC is becoming increasingly difficult, as is becoming evident by the performance success stories of PaRSEC.

5. TASK AFFINITY AND GRAPH SCHEDULING

Task scheduling on multiprocessor systems is a well-studied problem in parallel processing. Kwok et al. [25] offer a thorough survey of the literature for the interested reader. Finding an optimal schedule is an NP-complete problem in general, therefore most research has focused on efficient heuristics and approximation algorithms.

In PaRSEC we took a different approach. As the DAG of tasks is never available as a whole, other mechanisms for efficient scheduling must be involved, mechanisms that require less global knowledge. The runtime reacts to the availability of new ready tasks and tries to schedule them on the most appropriate computing resources. The user-provided task affinity clues have little impact on the execution within a node, they are used to map tasks onto different compute nodes. As a result, from the perspective of each node, the effort of scheduling tasks locally is independent of the effort on other nodes. Fortunately, the state of the computing resources, together with the knowledge of ready tasks, provides enough information to allow for simple, yet efficient, scheduling algorithms. By default, PaRSEC includes several scheduling algorithms (that a user can choose from) that try to optimize the following heuristics: (a) memory locality; (b) starvation minimization; and (c) user defined task priorities. However, even smarter scheduling strategies can be envisioned by extending the scheduler knowledge of future tasks using PaRSEC's ability to examine ancestor and descendant tasks in the symbolic execution DAG. To understand the importance of examining ancestor and descendant tasks symbolically, consider the task "`Task2(37,190)`" from the PTG in Figure 5. By evaluating the symbolic expressions in the PTG, PaRSEC knows that this task has "`Task2(37,191)`" as its descendant, or no descendants if the condition "`j < c*W-1`" is false. This knowledge is obtained without traversing any in-memory data structures, since PaRSEC does not store the dynamic DAG of tasks in memory. It is obtained by setting the parameters "`i`" and "`j`" to 37 and 190 respectively, and evaluating the symbolic expressions; and this is done regardless of the state of the execution and without the need for the runtime to first "discover" those tasks. In PaRSEC, all possible tasks can be examined at any time.

Memory locality is maximized by maintaining a hierarchy of local queues of ready tasks that maps the memory hierarchy of the computing node (i.e., typically a queue per core, one shared per socket, one shared for the entire node). This heuristic guarantees some level of locality for the following reason. Consider that task T_a enabled task T_b by producing, as output, the data element D that T_b needs as input. Since ready tasks are stored in local queues by this scheduling algorithm, and since T_b was enabled by T_a, the two tasks will go into the same queue and therefore run on the same hardware resource.

Starvation minimization is achieved through the use of shared task queues across the node. Clearly, this criterion is antithetical to the memory locality criterion. A good balance is achieved in PaRSEC by using a hybrid approach. Short local queues, that improve locality, are used to store ready tasks and when these queues are full, additional ready tasks are stored in shared queues, that reduce starvation.

Finally, PaRSEC includes a variety of algorithms for scheduling tasks based on user provided priorities. Priorities can be

arbitrary expressions, providing the developer with a versatile tool for affecting scheduling. This approach is useful for algorithms that are regular and well understood by human developers, and where there is a clear bijection between priorities and the execution critical path.

6. RELATED WORK

Multiple task based execution solutions are being currently actively developed [1, 2, 4, 6, 7, 8, 13, 30, 36]. While their level of similarity with PaRSEC varies, none of them supports the PTG programming model. They all rely on the Dynamic Task Graph of the task instances. These systems allow the generation of asynchronous tasks that are scheduled by the runtime, but use in-memory structures that grow with the number of task instances to detect the dependencies between individual task instances. The PTG approach, discussed in this paper, enables the declaration of parameterized dependencies between task classes without enumerating individual task instances.

PGAS languages [11, 19, 12, 28] eliminate explicit message passing and offer high programmability. However, these approaches put too much of the burden on the compiler and have yet to consistently deliver high performance at scale. Also, PGAS languages either do not support tasks, or support a limited form of fork-joint task parallelism.

Charm++ [22, 23] proposes a rich programming language for parallel computing that tries to cope with the issues related to the traditional Coarse Grain Model, by oversubscribing execution resources. An extension of this Charm++ language, Structured Dagger, provides support for *chares* (tasks) linked through structured control flow dependencies. One major difference with PaRSEC is that the flow of data between these chares is defined through explicit message passing instead of providing the flexibility of the parametric definition of data dependencies.

PYRROS is a parallel programming tool developed in the 90's by Yang and Gerasoulis [35]. While the language used to specify tasks and their dependencies is semantically equivalent to the PTG, PYRROS differs from PaRSEC fundamentally in that it is a static graph scheduling tool. Specifically, it examines the DAG of execution and tries to find an approximate solution to the problems of clustering tasks and mapping the clusters on processors.

In summary, the PTG programming model, as realized by PaRSEC, differs from existing alternatives in that it allows applications to be described – in a concise symbolic representation – as a set of task classes with parameterized data-flow edges between them and no unnecessary control-flow. Such a description reveals the most parallelism available on the particular instantiated problem in a format that allows the runtime to freely explore the entire execution space.

7. PERFORMANCE EXPERIENCES WITH PaRSEC

In this section we briefly outline a set of experimental results from various experiments that demonstrate that PaRSEC can (a) scale to large numbers of processing units and take advantage of multiple accelerators in a highly efficient way; and (b) outperform state-of-the-art applications and libraries implemented using Coarse Grain Parallelism with MPI. Some of these results are from previous publications and some are from projects that are still a work in progress. In some experiments we used codes from the DPLASMA and LibSCI libraries. DPLASMA is a dense linear algebra library that we have implemented on top of PaRSEC and aims to provide functionality similar to that of ScaLAPACK. LibSCI is a version of ScaLAPACK that is provided by Cray and is tuned for Cray machines.

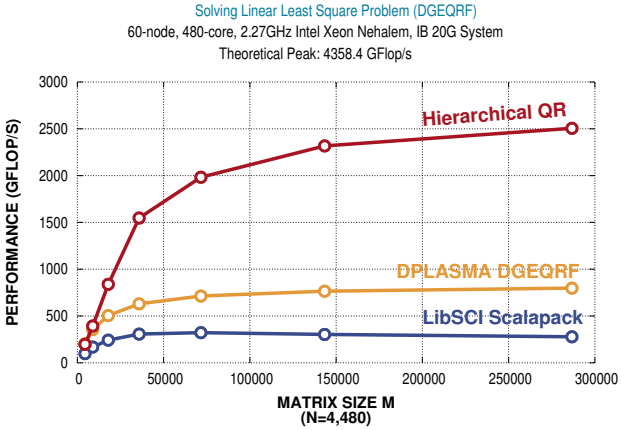

Figure 7: Performance of QR and HQR.

Figure 7 shows the performance of the Hierarchical QR factorization in PaRSEC [18], along with the more traditional QR factorization (DGEQRF) found in the DPLASMA library and the QR factorization found in libSCI. As we discussed in Section 4.3, HQR is a superior algorithm that only exists in PaRSEC. However, PaRSEC outperforms the MPI code by a significant margin, even if we do not take the more advanced algorithm into consideration. The main reason for this performance improvement comes from the dynamic rebalancing of the computation among threads, which reduces the idle time spent in (unnecessary) synchronization from communications.

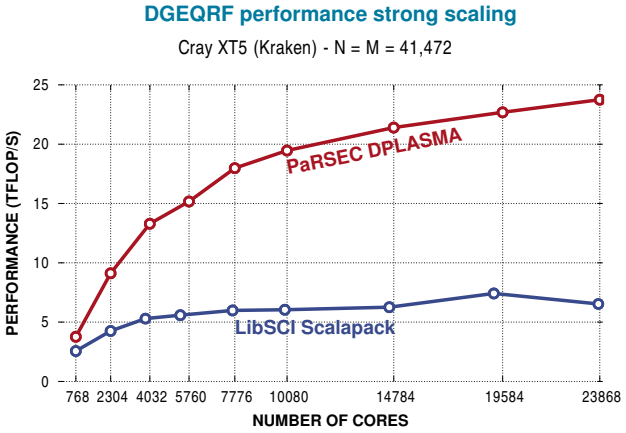

Figure 8: Performance of Systolic QR.

In Figure 8 we show a strong scaling experiment on a large scale run. The algorithm tested here is the Systolic QR [5] that is implemented in the DPLASMA library. The QR factorization from LibSCI is included in the graph for reference. We see that the MPI implementation stops scaling at about 4,000 cores, while the PaRSEC based solution keeps scaling all the way to 23,868 cores and beyond. One can note that expressing a systolic algorithm in a PTG form is natural, as there is a close match in the concepts between the PTG and systolic paradigm.

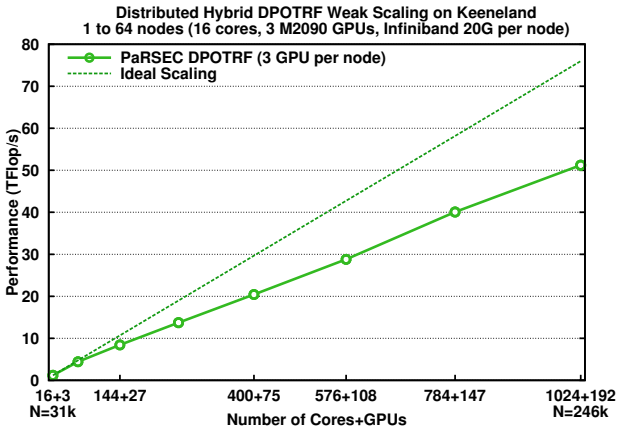

Figure 9: Performance of DPOTRF using distributed GPUs.

In Figure 9 we show the weak scaling of the Cholesky factorization when using up to 1024 cores and 192 GPUs (three per node) simultaneously. Clearly, the performance keeps increasing steadily and does not taper off even at this scale of heterogeneous execution. The GPU data transfer and the inter-node communications are handled automatically by the PaRSEC runtime, freeing the developer from the tedious expression of these operations, and allowing the runtime to schedule these communications with more freedom in order to maximize overlap.

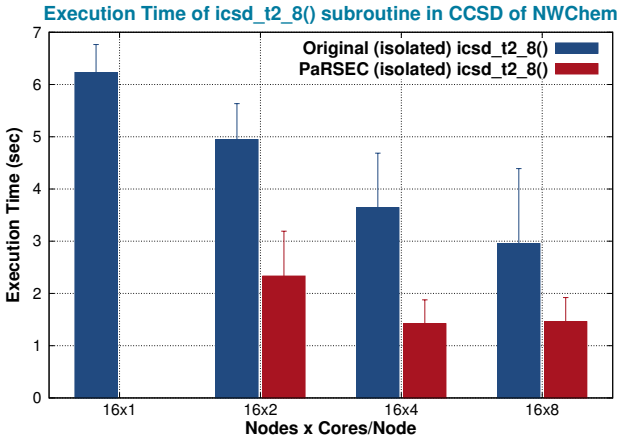

Figure 10: Execution time of NWChem subroutine.

Adapting the Coupled Cluster (CCSD) code of the computational chemistry package NWChem – that is FORTRAN 77 code automatic generated by the Tensor Contraction Engine (TCE) – to PaRSEC has been challenging and it is still a work in progress. However, we have successfully generated a PTG from a subset of this code and ran it over PaRSEC. Figure 10 shows the execution time of one of the most computationally intensive subroutines, in isolation, in the case of the original code and the PaRSEC version of the code (using uracil-dimer as input.) In addition to the computation threads, PaRSEC spawns a thread to handle the communication. Consequently, there is no entry for the single core execution for PaRSEC and the entries for 2,4, and 8 cores use one of the cores for the communication thread and the remaining for the actual work. The boxes show the fastest measured execution of the `icsd_t2_8()` subroutine across iterations and across nodes and the error bars extend to the slowest execution across iterations and nodes. As can be seen in the graph, PaRSEC achieves significantly faster execution for all core counts although it actually uses one less core every time.

8. CONCLUSION

There is little doubt that the tools and the programming methodologies for upcoming HPC systems will eventually evolve with the hardware architecture. But for our own sake, and by means of productivity, it is now time for a revolution rather than an evolution, a drastic shift toward more flexible and powerful programming constructs that will allow a more portable experience to applications development and a smoother transition with every novel architectural trend. Instead of settling for small steps forward in the name of safety, it is now time to take a leap forward, and develop languages capable of exposing a dynamic degree of parallelism together with supporting runtime or execution environments able to use this information in the most constructive and thus efficient way. What we have described in this paper, the PTG, is a stepping stone toward a more comprehensive description of parallelism constructs.

9. REFERENCES

[1] Intel Concurrent Collections for C/C++. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc.

[2] Intel Corporation. Threading Building Blocks. `https://www.threadingbuildingblocks.org/`.

[3] V. S. Adve and R. Sakellariou. Compiler synthesis of task graphs for parallel program performance prediction. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 208–226, London, UK, UK, 2001. Springer-Verlag.

[4] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23(2):187–198, 2011.

[5] G. Aupy, M. Faverge, Y. Robert, J. Kurzak, P. Luszczek, and J. Dongarra. Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC. In *Euro-Par Workshops*, pages 657–667, 2013.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International*

Conference on High Performance Computing, Networking, Storage and Analysis, SC '12. IEEE, 2012.

[7] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. Symposium on Parallel Algorithms and Architectures*, July 1995.

[8] O. A. R. Board. OpenMP Application Program Interface, Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[9] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, and J. Dongarra. From serial loops to parallel execution on distributed systems. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2012.

[10] R. Brightwell, R. Riesen, and K. D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. *Int. J. High Perform. Comput. Appl.*, 19(2):103–117, May 2005.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented approach to non-uniform Clustered Computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[13] S. Chatterjee, S. TasÄśrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating Asynchronous Task Parallelism with MPI. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725, May 2013.

[14] M. Cosnard and E. Jeannot. Compact DAG representation and its dynamic scheduling. *J. Parallel Distrib. Comput.*, 58(3):487–514, 1999.

[15] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, page 113, Washington, DC, USA, 1995. IEEE Computer Society.

[16] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. MPI-aware Compiler Optimizations for Improving Communication-computation Overlap. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM.

[17] S. Didelot, P. Carribault, M. Pérache, and W. Jalby. Improving MPI Communication Overlap with Collaborative Polling. *Computing*, 96(4):263–278, Apr. 2014.

[18] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert. Hierarchical QR Factorization Algorithms for Multi-core Clusters. *Parallel Comput.*, 39(4-5):212–232, Apr. 2013.

[19] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Specification v. 1.3. `http://upc.gwu.edu/documentation.html`, 2003.

[20] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.

[21] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[22] L. Kale. Parallel Programming with CHARM: An Overview. Technical Report 93-8, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[23] L. V. Kale and G. Zheng. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects. In M. Parashar, editor, *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282. Wiley-Interscience, 2009.

[24] K. Kennedy, C. Koelbel, and H. Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7–1–7–22, 2007.

[25] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, Dec. 1999.

[26] D. Loveman. High performance Fortran. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):25–42, Feb 1993.

[27] A. Nomura and Y. Ishikawa. Design of kernel-level asynchronous collective communication. volume 6305 of *Lecture Notes in Computer Science*, pages 92–101. Springer Berlin Heidelberg, 2010.

[28] R. W. Numrich and J. K. Reid. Co-Array Fortran for parallel programming. ACM Fortran Forum 17, 2, 1-31, 1998.

[29] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 142–151, Tsukuba, Japan, 29 Sept - 1 Oct 2008.

[30] 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.

[31] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[32] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Aprà, T. L. Windus, and W. de Jong. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, SEP 2010.

[33] M. Wolfe. Doany: Not just another parallel loop. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 421–433, London, UK, UK, 1993. Springer-Verlag.

[34] M. Wolfe. Compilers and More: MPI+X. `http://www.hpcwire.com/2014/07/16/compilers-mpix/`, 2014.

[35] T. Yang and A. Gerasoulis. Pyrros: Static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 428–437, New York, NY, USA, 1992. ACM.

[36] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.