Performance Modeling of Gyrokinetic Toroidal Simulations for a many-tasking runtime system

Matthew Anderson, Maciej Brodowicz, Abhishek Kulkarni, Thomas Sterling
Center for Research in Extreme Scale Technologies
Indiana University
{andersmw, mbrodowi, adkulkar, tron}@indiana.edu

***Abstract*—Conventional programming practices on multicore processors in high performance computing architectures are not universally effective in terms of efficiency and scalability for many algorithms in scientific computing. One possible solution for improving efficiency and scalability in applications on this class of machines is the use of a many-tasking runtime system employing many lightweight, concurrent threads. Yet a priori estimation of the potential performance and scalability impact of such runtime systems on existing applications developed around the bulk synchronous parallel (BSP) model is not well understood. In this work, we present a case study of a BSP particle-in-cell benchmark code which has been ported to a many-tasking runtime system. The 3-D Gyrokinetic Toroidal code (GTC) is examined in its original MPI form and compared with a port to the High Performance ParalleX 3 (HPX-3) runtime system. Phase overlap, oversubscription behavior, and work rebalancing in the implementation are explored. Results for GTC using the SST/macro simulator complement the implementation results. Finally, an analytic performance model for GTC is presented in order to guide future implementation efforts.**

I. Introduction

The level of thread parallelism provided by the multicore processors pervasive in present-day high performance computing systems has increased the relative prominence of the concept of many-tasking: implementing an application using many lightweight concurrent threads for a wide variety of application components. Many-tasking enables several key execution concepts crucial for improving performance and scalability, including: task oversubscription, or the overdecomposition of a problem resulting in multiple tasks competing for a single computational resource; overlapping of computational phases, including overlapping communication and computation phases in order to hide network latency; and intelligent task scheduling, resulting in implicit load balancing controlled by the task scheduler. These benefits have been documented across a wide variety of software libraries and runtime systems, including more recently using MPI [1], OpenMP [2], Charm++ [3], Unified Parallel C (UPC) [4], MPI+OpenMP [5], MPI+UPC [6], Intel Threading Building Blocks (TBB) [7], [8], High Performance ParalleX (HPX) [9], [10], Cilk plus [11], Chapel [12], XKaapi [13], Coarray Fortran 2.0 [14], and Qthreads [15]. For several decades, the qualitative characteristics of many-tasking have been set forth in the Actors model [16], Multilisp [17], Fortress [18], and X10 [19].

While the many-tasking capabilities of runtime systems and libraries continue to improve, scientific applications overwhelmingly employ the bulk synchronous parallel (BSP) model [20], [21]. Porting an application designed around the BSP model to a many-tasking execution model can involve significant development time and algorithm redesign costs while the performance benefits of such a transition are hard to quantify before performing the port. Specific case studies and discrete event simulators can assist in identifying and quantifying such performance benefits. This work provides a specific case study as well as a discrete event simulator built for the ParalleX execution model [22] to assist in quantifying expected performance improvements resulting from transitioning an application from the BSP model to a many-tasking execution model.

In the case study presented here, we examine the effects of transitioning the 3D Gyrokinetic Toroidal Code (GTC) [23] from a BSP model to the ParalleX execution model as implemented by the HPX-3 runtime system. HPX-3 is an experimental, ParalleX-compliant runtime system developed in C++ at the Louisiana State University. It features lightweight (user space) multithreading, advanced synchronization primitives called Local Control Objects (LCOs), parcel based communication that extends the concept of active messages, and Active Global Address Space (AGAS). All computational objects created by an HPX program are assigned unique identifiers by AGAS and are free to migrate between compute nodes; HPX provides mechanisms that can transparently access both local and remote objects using the same interface. This approach facilitates building of parallel, asynchronous, message-driven applications that are capable of migrating work to data when beneficial to overall performance.

GTC uses the particle-in-cell method [24] for plasma simulations and forms part of the NERSC-6 suite of benchmarks [25]. There are numerous performance studies on the MPI version of GTC [26], [27] across a wide array of architectures making it an ideal candidate for this case study. The metrics explored here include performance, communication characteristics, and the overlap of phases both in cases with an ideal load balance and a moderate load imbalance.

Complementing the GTC implementation effort in HPX-3 are two additional performance modeling efforts: one using coarse grained simulation of GTC with the SST/macro simulator [28] and the second using an analytic performance model of GTC for ParalleX. Overall, this work makes the following new contributions:

- It provides both a simple legacy migration path for a BSP style code to run in a many-tasking runtime system with minimal modifications and a performance comparison between the different modalities of computation. The legacy migration path consists only of enforcing thread safety and replacing

MPI calls with task model equivalents.
- It examines the phase overlap and implicit load balancing capabilities of a many-tasking runtime system executing a code designed for BSP and quantifies the benefits derived from these capabilities.
- It provides a performance simulator for comparing performance, communication, and phase overlap characteristics for a BSP style code in a many-tasking runtime system.
- It provides an analytic performance model of GTC using the ParalleX execution model.

This work is divided into six parts. Work related to this study and where this study fits into the broader discussion about many-tasking execution models is discussed in Section II. Section III introduces the GTC code as well as the methodology behind the GTC port to HPX-3. It presents the implementation results exploring performance, communication, and phase overlap. Section IV introduces the skeleton GTC code and its use within SST/macro for gauging the impact of key runtime system overheads on performance. Section V presents an analytic performance model based on ParalleX and developed for GTC. Section VI presents our conclusions.

II. Related Work

There are several recent efforts to explore how an application changes when transitioning from MPI to a new runtime system or programming model. The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) applications was recently examined using a wide range of conventional and emerging programming models, including MPI, OpenMP, MPI+OpenMP, CUDA, Charm++, Chapel, Liszt, and Loci [29]. The numerous application implementations contain a wide range of source code line counts and implementation choices specific to each programming model in order to systematically explore productivity benefits of each. They found that several emerging programming models showed significant productivity benefits over conventional approaches as measured by the number of lines of code needed to produce the parallel implementation of LULESH. However, they also found that several models required significant additional development just to match the performance of the MPI version.

Several other studies have compared and contrasted performance using microbenchmarks with a focus on execution overhead, such as in Appeltauer et al. [30] using context-oriented languages and in Gilmanov et al. [31] using task models. Olivier et al. [32] discusses comparisons of task models using an imbalanced task graph as the proxy application. However, no study exists which examines the performance characteristics of a full application designed for BSP but run in a many-tasking runtime system where the only change to the original code is thread safety enforcement and the replacement of MPI calls with task model equivalents. Performance simulators for task models along with their comparisons to BSP are also missing from the literature.

Madduri et al. explore the impact of multicore-specific optimizations for gyrokinetic toroidal simulations and report up to 2x speedup using hybrid MPI-OpenMP and MPI-OpenMP-CUDA GTC versions [33]. Performance improvements by overlapping computation and communication for GTC using OpenMP tasking have also been demonstrated previously [34].

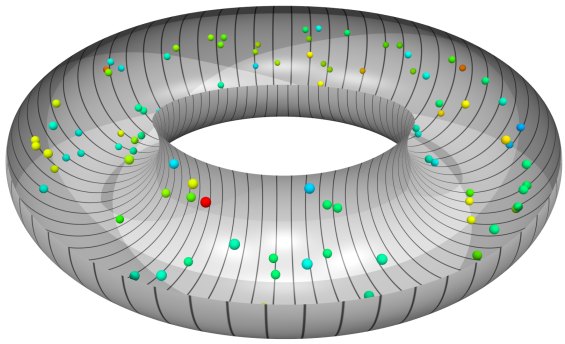


Fig. 1: A timestep in the GTC simulation showing tracking of select particles in the toroidal mesh. The color indicates the speed of the particle while the lines dividing the torus indicate the domain decomposition across processors or threads for this simulation.

There are multiple performance modeling efforts either using coarse grained simulation or analytic performance models. Hoefler et al. [35] enumerate how analytic performance models can guide systematic performance tuning. Hendry [36] analyzed and reported on the MPI based GTC skeleton code for the SST/macro coarse grained simulator with a focus on reducing power consumption while maintaining performance. Analytic performance models for MPI collectives were explored by Angskun et al. [37] while Mathis et al. [38] created a performance model for particle transport.

III. A Case Study: GTC

Using the default input parameters for GTC, we examine the code scaling, phase, and performance characteristics in this section. The default parameter case evolves 3.2 million particles using the particle-in-cell approach inside a toroidal mesh with 3.6 million gridpoints for 150 steps with four point gyro-averaging on the mesh. Figure 1 provides a visualization at a timestep in the simulation showing particles location and speed in the mesh.

The GTC algorithm utilizes six basic types of communication operations: allreduce, broadcast, split communicator, gather, reduce, and send/receive. In order to port GTC to HPX-3, nonblocking implementations of each of these operations were created using HPX-3. The port from MPI to HPX-3 consists of replacing all MPI calls with their HPX-3 equivalents, after making the original GTC code thread safe so that it can be used with a multithreaded runtime system like HPX-3. The GTC version ported to HPX-3 is identified as GTCX. Output from GTC and GTCX were verified to be identical out to 15 significant digits for 8 separate analysis fields. Simulations were conducted on a 16 node cluster of Intel Xeon E5-2670 2.60 GHz processors providing 16 cores per node with InfiniBand interconnect between nodes. Each node is equiped with 32 GB of 1600 MHz RDIMMS.

While GTC and GTCX are nearly identical in terms of the codebase, their computational phase characteristics are not. GTCX is capable of overlapping computational phases by overdecomposing the problem into more lightweight concurrent threads than execution resources (cores). This overdecomposition of the problem enables the overlap of computation and

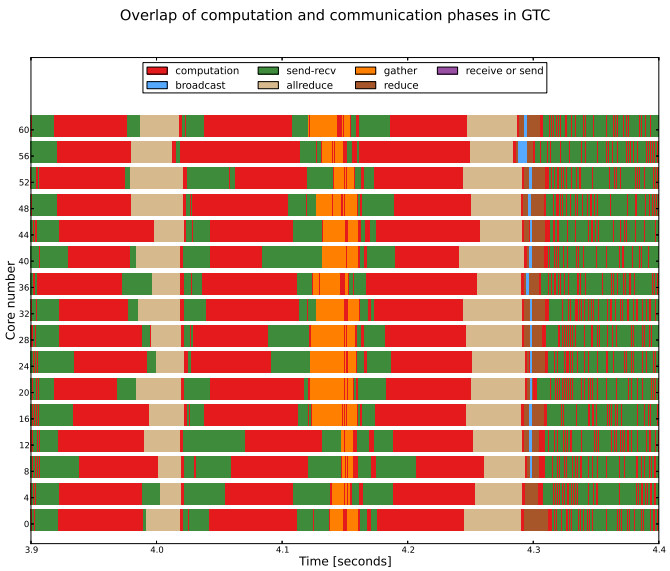


Fig. 2: The phases of computation for a portion of a second in a GTC (MPI based) simulation on 64 cores. The phases for every fourth processor are plotted in the vertical axis for GTC. There is no oversubscription in this case; blocking MPI collectives ensure the computational phases do not overlap signficantly. For comparison with GTCX (HPX-3 based), see Figure 3 where the computational phases for the operations corresponding to those here are plotted.

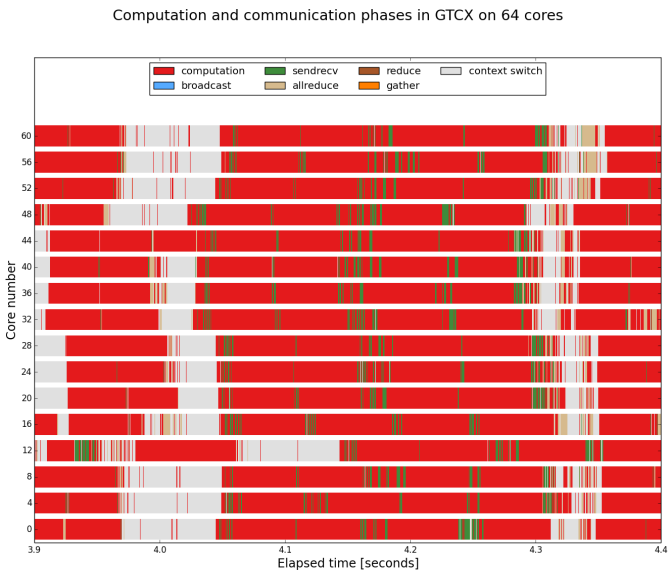


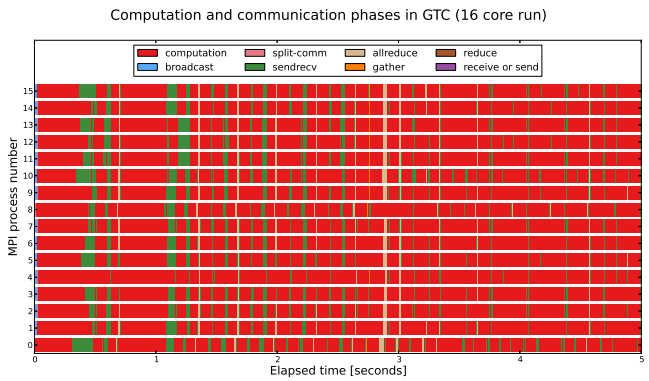
Fig. 3: The phases of computation a portion of a second in a GTCX (HPX-3 based) simulation on 64 cores. The phases for every fourth core are plotted in the vertical axis for GTCX. The simulation oversubscribes the computational resources by a factor of two in order to hide network latency and overlap more computational phases than otherwise possible when using blocking collectives. For comparison, the GTC case is shown in Figure 2.

communication phases in an effort to hide network latency when used in conjunction with the nonblocking HPX-3 equivalents of MPI collectives. The computational phases for GTC on 64 cores is illustrated in Figures 2-3. In Figure 2, the number of MPI processes running GTC was equivalent to the number of computational resources. In Figure 3, however, the number of HPX-3 threads running GTCX was a factor of two greater than the number of resources. The phases of computation are color coded; however, in the HPX-3 case in Figure 3, context switching is usually how waiting for communication is manifested since the communication calls are nonblocking. A noticeable increase in the overlap of computational phases is evident in the GTCX simulation compared with the GTC simulation.

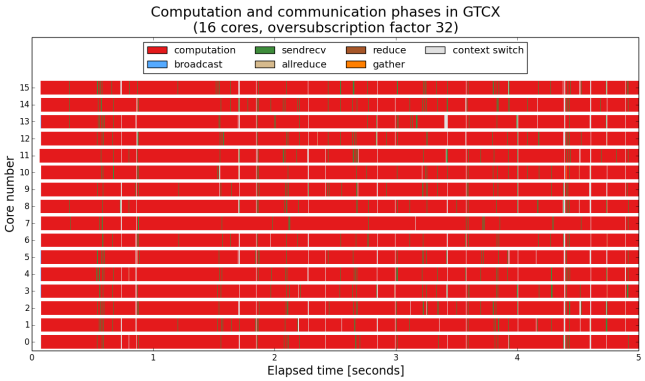
The increase in overlap of computational phases becomes more evident by introducing a synthetic load imbalance to one of the threads or processes. For GTC, a load imbalance results in the idling of resources until the slowest process catches up with the rest of the processes in its computation. For GTCX, the load is implicitly balanced among the resources on the node by the thread scheduler. Figure 4 compares the phases of computation for GTC and GTCX with and without a synthetic load imbalance. In Figure 4, a synthetic load imbalance involving computing the ϕ potential during the GTC/GTCX Poisson equation solve is added to the process or thread identified as zero. This load imbalance results in an immediate increase in time spent waiting for communication in GTC for all processes except zero. In GTCX there is also an increase in time spent in context switches; however, that increase is amortized by the thread manager maximizing resource usage resulting in less overall waiting.

Direct strong scaling performance measurements between GTC and GTCX are presented in Figure 5. In this figure, simulations using GTC and GTCX were performed five times prior to averaging and reporting the results. The results also include performance results from a version of the GTC code manually implemented to use non blocking collectives from MPICH2 but without oversubscription. The GTC and GTCX performance is nearly identical on very few codes while GTCX suffers a considerable decrease in performance at 16 cores and higher, matching the GTC performance only at 128 cores where the GTC code has already stopped scaling. The GTCX implementation continues to scale beyond 128 cores and produces the fastest result at 256 cores. As will be explored in detail later in Section III, the use of blocking collectives contributes to some of the performance degradation observed in GTC. The MPI version of GTC which uses non blocking collectives provides an intermediate comparison point between standard GTC and GTCX where blocking collectives are removed but no oversubscription is present.

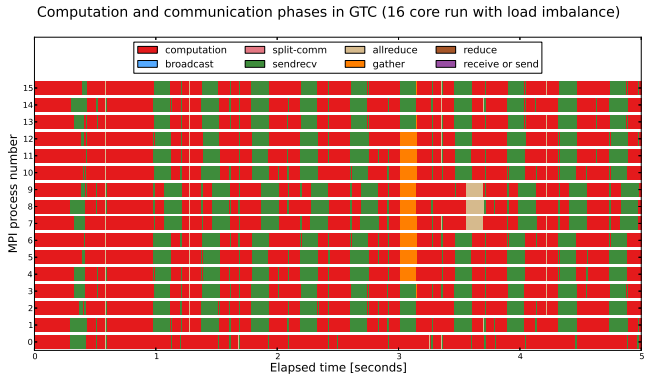
With the exception of a few cases, GTCX generally lags GTC performance in spite of the increase computational phase overlap and network latency hiding capability of GTCX. However, it also continues to scale even when the GTC code has stopped scaling. The overheads associated with thread creation (2 μs) and context switching (1.2 μs) as well as a large overhead in the network layer contributes to mitigating many of the performance benefits in GTCX resulting from an increase in overlapping computational phases. The legacy migration path used to create the GTCX code from GTC involves minimal code modification and no code



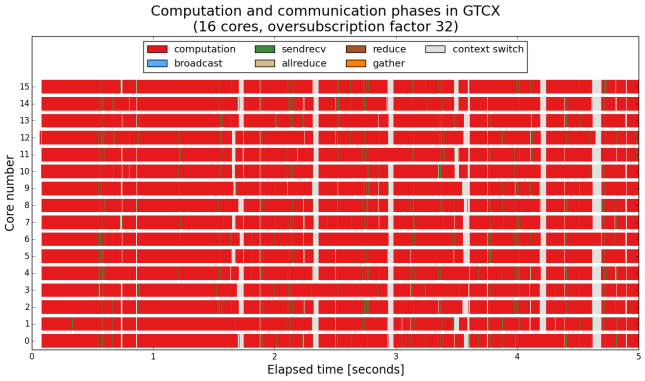
(a) GTC



(b) GTCX



(c) GTC with load imbalance



(d) GTCX with load imbalance

Fig. 4: Computational phase diagrams for GTC and GTCX with and without a synthetic load imbalance on 16 cores are presented here. For GTC, a load imbalance results in the idling of resources until the slowest process catches up with the rest of the processes in its computation as seen by comparing (a) and (c). There is also an increase in idled resources for GTCX in the presence of a load imbalance; however, the difference between (b) and (d) for GTCX is not as substantial as that for GTC due to the ability to overlap computational phases and implicitly load balance.

restructuring in order to achieve more efficient performance for a many-tasking execution model. Restructuring GTC for a specific programming model has resulted in significant performance gains for GTC before (e.g. see [33]). However, for many legacy applications, restructuring an application code base for use in a new programming model is not a viable option while the legacy migration path explored here could easily be achieved at the compiler level.

Overlapping computational phases, hiding network latency, and removing global barriers in computation give key performance benefits which can improve scalability in applications provided the runtime system overheads can be kept in check. Understanding how these overheads can affect application performance is crucial for making design decisions and extracting more parallelism in a simulation. While runtime system overheads cannot be easily changed in an implementation in order to empirically observe their impact on overall application performance, overheads can be changed and experimented with using a discrete event simulator. The following section explores this behavior in the context of the SST/macro simulator.

IV. GTCX in SST/macro

To explore the scalability characteristics beyond the bounds of available physical machines we used SST/macro simulator [28] developed at Sandia National Laboratory. SST/macro is a coarse-grain simulator, which offers a good balance of simulation speed, accuracy of results, scaling of the model to arbitrary number of nodes, and support of emulation on alternative architectures. As such, it is ideal for realistic modeling of applications at scale, studying the effects of network parameters and topology on performance, and software prototyping of new algorithms and library designs. SST/macro accomplishes this through skeletonization of the modeled applications which is a process of creating simplified code that approximately reproduces the behavior of the original application, but without having to produce the same computational results. From the network perspective, skeletons preserve the control flow of communication code, resulting in as much as order of magnitude potential speedup. Currently, the conversion process of the original application code to an equivalent skeleton is manual, although compiler assisted utilities are being developed at Indiana University. The simulation runs as a single process (shared address space), with component application processes emulated by user level threads.

SST/macro is capable of modeling with significantly more diverse range of parameters than most of the commonly available tools used to predict the performance of MPI applications based

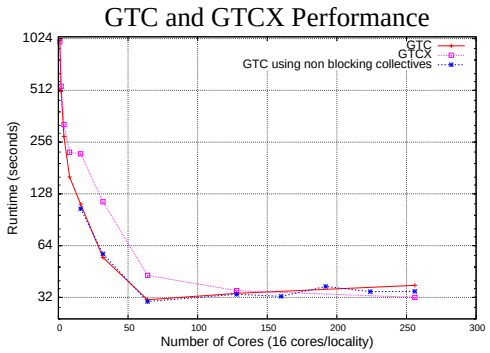


Fig. 5: Performance measurements for GTC and GTCX reflecting a strong scaling test. Also shown is a version of GTC implemented using non blocking collectives but without oversubscription. GTCX performance varies significantly based on the number of lightweight threads used to decompose the problem. The GTCX results presented here reflect the use of the empirically discovered optimal number of lightweight threads for decomposition. While GTCX is able to overlap more computational phases than GTC, it also suffers from higher overheads in the form of thread creation and context switches resulting in slightly worse performance than GTC from 16 to 128 cores.

on execution traces, such as LogGOPSim [39]. For example, compute node parameters include core affinities, memory contention along with NUMA effects, and NIC contention. Network switch models support packet arbitration, adaptive routing, and buffering parameters. The network topologies are represented accurately and may support message traffic as flows, packets, or packetized flows. Moreover, trace gathering may substantially stress the I/O subsystem due to volume of stored data; this frequently interferes with network operation if the storage devices are attached to the same interconnect. SST/macro is free of these issues.

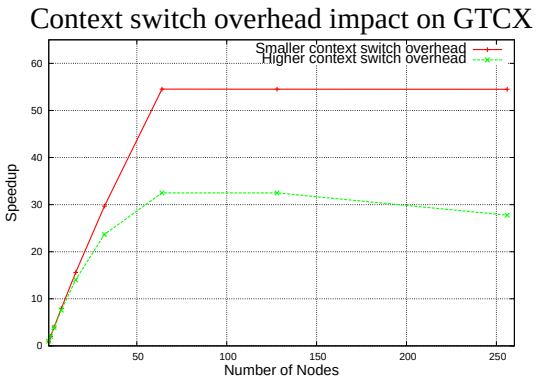


Fig. 6: Comparison of strong scaling for GTCX with two different context switch overheads. The smaller context switch overhead is .1μs while the larger is 10 ms. Results using empirically determined near-optimal oversubscription factors are plotted.

The GTCX implementation explored in Section III illustrated key characteristics distinguishing the many-tasking behavior of ParalleX from the MPI behavior of the Communicating Sequential Processes (CSP) exection model. These include

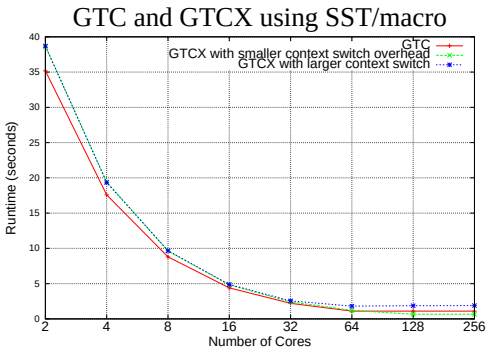


Fig. 7: Predicted performance comparison for GTC and GTCX using SST/macro. For GTCX, two different context switch overheads are examined. No oversubscription was applied to GTCX; consequently GTCX and GTC closely mirror each other in performance.

computational phase overlap, overdecomposition, network latency hiding, implicit load balancing, and intelligent task scheduling. However, the GTCX implementation performance in Section III was generally at par or worse than the MPI implementation due to the large overheads introduced by the runtime system implementation. The SST/macro toolkit, in contrast to a full runtime system implementation, is able to represent the parallel machine using models to estimate processing and network components and thereby modify the size of the overheads. Recently, HPX-3 semantics were added to SST/macro in order to model application performance using that runtime system at different overhead levels [40]. This section explores GTCX, the HPX-3 implementation of GTC, using the SST/macro simulator with different runtime system overheads on the Hopper supercomputer (Cray XE6, Opteron 6172 12 cores at 2.10 GHz).

As a tool for co-design, SST/macro is frequently used for coarse grained rather than cycle accurate simulation. SST/macro is also often used in conjunction with a skeleton code, where all computation has been removed from an application except for communication and control. This enables the skeleton to be used for fast prototyping and making other design decisions. Actual non-control computations in the skeleton code are replaced with counters in order to record the computational cost without actually performing the computation entailed. Consequently, skeleton applications can run orders of magnitude faster than their application counterparts while still faithfully modeling the communication and control characteristics of the original application. A skeleton application of the MPI version of GTC is provided as an example in SST/macro; this skeleton served as the basis for the GTCX skeleton used for the data provided in this section. The MPI GTC skeleton for SST/macro, while sharing qualitative performance behavior with the actual MPI GTC implementation, does differ in runtime performance prediction. These differences between the implementation performance and the skeleton's predicted performance are due, in part, to the difficulty of replacing computational loops in the implementation with computation counters in the skeleton. Consequently, in this sec-

tion we directly compare the GTC skeleton performance with the GTCX skeleton performance rather than the full implementations.

Using the GTC skeleton provided in SST/macro, the same legacy migration path described in Section III was applied producing the GTCX skeleton. Unlike the implementation in III, however, the thread overheads and context switching overheads can be changed as a parameter in SST/macro in order to model the performance impact of these overheads. Figure 6 shows GTCX strong scaling behavior for context switch overheads differing by 5 orders of magnitude from .1μs to 10 ms. Figure 7 compares the predicted performance between GTC and GTCX in SST/macro without the benefit of oversubscription. Not surprisingly, without oversubscription in GTCX, the results from GTC and GTCX are very similar. Context switching overheads become more pronounced at 64 cores or higher.

V. GTCX Analytic Performance Model

Performance prediction of applications in alternate execution models is an emerging research problem. The overall performance gain largely depends on the characteristics of the algorithm itself. Limited improvement in performance is observed already by a straightforward translation of the BSP-style communication primitives to their equivalent in HPX-3, as shown earlier. This is due entirely to the finer-grained concurrency and dataflow parallelism allowed by the execution model. Better performance can be achieved by leveraging more features admitted by HPX-3, and at times, through a complete rewrite of the application's algorithm. Our port of GTC to the HPX-3 many-tasking runtime system evaluated the effectiveness of its core parallelism constructs and provided a feedback on the performance of the implementation at relatively modest scales. To understand and appropriately quantify the effect of overlapping computation with communication through oversubscription and dataflow parallelism, in this section, we introduce an analytic performance model of the gyrokinetic toroidal simulation (GTC) code.

Analytic models that calculate the long-time running cost of parallel applications have to strike a trade-off between accuracy and overall cost. The model described in this section captures the essential computation and communication characteristics of the six key phases in the application. Through parameter sweeps of the overheads involved, and varying the degree of overlap, bounds for performance benefits of an alternate execution model are obtained.

GTC employs the Particle-in-Cell formalism to model the plasma interactions. The PIC algorithm involves six main phases executed at each time step. The static call-graph of the core GTC algorithm is shown in Figure 8.

The total runtime of GTC for a given sample input depends several parameters such as the number of particle domains (*npartdom*), the 1D toroidal decomposition (*ntoroidal*), the size of the grid, the number of particles per cell etc. For a given input and simulation parameters, the total runtime cost is the sum of the initialization and the execution time of the six phases for n timesteps as given in Eqn. 1.

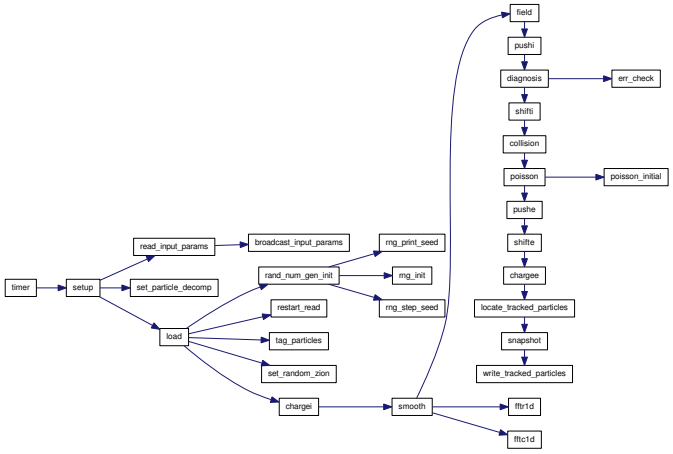


Fig. 8: Static Call Graph of GTC.

$$T_{total} = T_{setup} + \sum_{i=1}^{n} \Big(T_{charge} + T_{poisson} + T_{field} + T_{smooth} + T_{push} + T_{shift} \Big) \quad (1)$$

A. Setup

In this phase, the GTC simulation is set up by reading the input parameters. Both integer and real parameters are read separately from an input file by the main process. These parameters are packed and broadcasted to all other processes where they are subsequently unpacked and assigned locally. Relative to other phases, the time taken by this phase is rather negligible and bounded by constant factor. We expand on this phase only to elucidate our analytic modeling methodology and highlight the key time gains in GTCX over GTC.

Type	Action	Weight	Description
Comp	s_1	$\Theta(nints)$	Read integer parameters
Comp	s_2	$\Theta(nreals)$	Read real parameters
Comm	bcast	144	Broadcast integer parameters
Comm	bcast	224	Broadcast real parameters
Comp	k_1	$\Theta(nints)$	Unpack integer parameters
Comp	k_2	$\Theta(nreals)$	Unpack real parameters

TABLE I: Setup

Table I shows the key communication and computation steps involved in this phase. Since GTC is implemented in a BSP style using MPI, each of the communication steps are inherently parallel. For instance, T_{bcast} represents the time taken by all processes to perform the broadcast communication operation. We use the same terms for GTCX to keep the presentation of our model simple. However, the collective communication operations can themselves be decomposed into their constituent point-to-point operations as shown below.

The time taken by the main process for this phase is given by

$$T_{setup} = s_1 + s_2 + T_{send}(144) + T_{send}(224)$$

and the time at all of the other processes is

$$T_{setup} = T_{recv}(144) + T_{recv}(224) + k_1 + k_2$$

Here, $T_{send}(x)$ and $T_{recv}(x)$ is the time taken to send and receive x bytes respectively. The computation time between the communication operations is measured empirically by profiling the application. Since we are primarily interested in comparative analysis (between GTC and GTCX), using one of the many analytic communication models to represent the communication costs is also a viable approach.

For instance, in the Hockney communication model [41], the time to send or receive a message of size m is given by $T_m = \alpha + \beta m$. Using this, a linear broadcast of n messages of size m to P processes can be computed as follows:

$$T_{bcast} = n(P-1) \cdot \Big(m \cdot (\alpha + \beta m)\Big)$$

Assuming no network congestion and full bisection-bandwidth, this approximates the time to perform broadcast at all processes.

Since there are no data dependences between the broadcast of the integer and real parameters, GTCX can execute them concurrently in separate HPX-3 tasks, such that the computation and communication is overlapped. Even when using a single task in HPX-3, we have Eqn. 2.

$$\begin{aligned} T_{setup} &= s_1 + T_{bcast}(144) + \max\Big(s_2 + T_{bcast}(224), T_{bcast}(144) + k_1\Big) \\ &\quad + T_{bcast}(224) + k_2 \end{aligned} \tag{2}$$

Here, the packing (and sending) of the real parameters (s_2 and $T_{bcast}(224)$) is overlapped with the unpacking (and receiving) of the integer parameters (k_1 and $T_{bcast}(144)$).

By having two lightweight threads (one to send the integer parameters, and the other to send the real parameters) run concurrently, both the tasks can be effectively overlapped, as in Eqn. 3.

$$\begin{aligned} T_{setup} &= \max\Big(s_1 + T_{bcast}(144), s_2 + T_{bcast}(224)\Big) \\ &\quad + \max\Big(k_1, k_2\Big) \end{aligned} \tag{3}$$

The degree of overlap is bounded by δ subject to factors such as the number of system threads in HPX-3 and the progress of communication with respect to computation.

B. Charge

In this phase, the charge from the particle data is deposited onto the grid using a particle-grid interpolation step. The computation operations in this phase, as shown in Table II are asymptotically bounded above the number of particles in each process. The load-balance of the computation step, thus, largely depends on the particle distributions. The steps c_1, c_2 iterate through the particle array and update grid locations in memory corresponding to the four-point ring representing the charged particles. The effective time taken by these steps depends on the arithmetic intensity of each step and non-deterministic architectural factors such as cache behavior etc. These can either be ignored completely, or measured empirically for a given run.

The time taken by this step, Eqn. 4, is the global sum of the computation steps, an allreduce communication step to deposit charge density on the grid and the computation of the global sum of ϕ.

$$\begin{aligned} T_{charge} &= c_1 + c_2 + T_{allreduce}(mgrid \cdot (mzeta+1)) + c_3 \\ &\quad + T_{sendrecv}(mgrid) + c_4 + T'_{allreduce}(mpsi+1) \end{aligned} \tag{4}$$

Type	Action	Weight	Description
Comp	c_1	$\Omega(mi)$	Particle-grid interpolation
Comp	c_2	$\Omega(mgrid \cdot mzeta)$	Set density
Comm	allreduce	$mgrid \cdot (mzeta+1)$	Deposit charge density on the grid
Comp	c_3	$\Omega(mpsi)$	Poloidal end cell
Comm	sendrecv	$mgrid$	Send density array to left and receive from right
Comp	c_4	$\Omega(mpsi \cdot mzeta \cdot mtheta_i)$	Flux surface average and normalization
Comm	allreduce	$mpsi+1$	Global sum of phi00

TABLE II: Charge

In GTCX, the loop to iterate through the array in c_2 is fused with the point-to-point communication operations in $T_{allreduce}$ so that they are interleaved and executed concurrently. Thus, the time to compute the global sum at all processes ($T_{allreduce}$) is offset by the time to iterate through the array to be reduced (c_2) by a constant factor δ. The time taken for the complete charge step for GTCX is given by

$$\begin{aligned} T_{charge} &= c_1 + \max\Big(c_2, T_{allreduce}\Big) + c_3 + T_{sendrecv}(mgrid) \\ &\quad + \max\Big(c_4, T'_{allreduce}\Big) \\ &= c_1 + c_2 + \delta + c_3 + T_{sendrecv}(mgrid) + c_4 + \delta' \end{aligned} \tag{5}$$

where $T_{allreduce} = c_2 + \delta$, $T'_{allreduce} = c_4 + \delta'$

Note that, here, we assume that there is a strong synchronization step (barrier) between phases. The degree of overlap between computation and communication (δ and δ') depend on the subscription factor in GTCX. They cannot be zero due to the data dependence between the operations, and the finite time to execute them.

Type	Action	Weight	Description
Comp	p_1	$\Omega(mzeta \cdot mgrid \cdot mring)$	Initialization
Comm	allgather	$\frac{mzeta}{npartdom}$	Gather full array ϕ on PEs
Comp	p_2	$\Omega(mzeta \cdot mgrid)$	Assign full array ϕ
Comp	p_3	$\Omega(mzeta \cdot mpsi)$	In equilibrium unit

TABLE III: Poisson

C. Poisson

The gyrokinetic Poisson equation is solved on the grid in this phase. The compute phases are asymptotically bounded above by the grid size ($\Omega(mgrid)$). The only communication step in the reference codes (GTC and GTCX) is an allgather collective operation. The other steps are shown in Table III.

In case of GTC, the time taken to execute this phase depends on the available processor parallelism. It is given by Eqn. 6.

$$T_{poisson} = p_1 + T_{allgather}(\cdot) + p_2 + p_3 \tag{6}$$

For GTCX, the performance gain due to oversubscription and dataflow parallelism is bounded by the factor δ. With true dataflow parallelism obtained by loop fusion and the conversion of sub-arrays into futures, the allgather communication step is effectively executed concurrently with both p_1 and p_2. The marked end times of the allgather and p_2 steps are determined by:

$$\begin{aligned} T_{poisson} &= \max\Big(p_1, T_{allgather}, p_2\Big) + p_3 \\ &= p_1 + \delta + p_3 \end{aligned} \tag{7}$$

where $T_{allgather} = p_1 + \delta_1, \quad p_2 = T_{allgather} + \delta_2$
$\delta = \delta_1 + \delta_2$

D. Field

This phase computes the electric field on the grid. As the previous phase (Poisson), it scales with the number of the poloidal grid points. The data redistribution involves array shifts and thus, the communication pattern is captured by MPI's `sendrecv` function calls. In GTCX, the send and receive operations are decoupled and hence can be scheduled closer to the actual data sinks and sources in the dataflow graph.

Type	Action	Weight	Description
Comp	f_1	$\Omega(mzeta \cdot mgrid \cdot mpsi)$	Finite difference for e-field
Comm	sendrecv	$mgrid$	Send ϕ to right and receive from left
Comm	sendrecv	$mgrid$	Send ϕ to left and receive from right
Comp	f_2	$\Omega(mzeta \cdot mgrid)$	Unpack ϕ boundary and calculate E_zeta
Comm	sendrecv	$3 \cdot mgrid$	Send E to right and receive from left
Comp	f_3	$\Omega(mgrid + mzeta \cdot (mpsi + 1))$	Unpack end points

TABLE IV: Field

The operations to shift the ϕ array and compute the electric field E are referred in Table IV. The time taken in GTC to perform the field phase for a given simulation step is approximated by

$$T_{field} = f_1 + 2 \cdot T_{sendrecv}(mgrid) + f_2 + T'_{sendrecv}(3 \cdot mgrid) + f_3 \quad (8)$$

The array shift of ϕ is overlapped with the computation of finite difference for the electric field. After the ϕ boundaries are unpacked, the calculuation of the electric field is fused with its communication operations. These optimizations result in a time reduction given by Eqn 9

$$T_{field} = \max\left(f_1, T_{sendrecv}(mgrid), T_{sendrecv}(mgrid)\right) + f_2 + \delta \quad (9)$$

where

$$T'_{sendrecv} = f_2 + \delta_1, \quad f_3 = T'_{sendrecv} + \delta_2$$
$$\delta = \delta_1 + \delta_2$$

E. Smooth

In this phase, the potential and charge density undergo radial smoothing. The computation kernels s_1, s_2ands_3 perform grid-accesses relative to the grid size (mgrid). As shown in Table V, a 2D-matrix is transposed using scatter and gather collective operations.

Eqns 10 and 11 can be used to approximate the times required to execute this phase in GTC and GTCX respectively.

$$T_{smooth} = s_1 + 2 \cdot T_{sendrecv}(mgrid) + s_2 + T_{sendrecv}(mgrid) + \sum_{i=1}^{ntoroidal}\left(s_3 + T_{gather}\right) + \sum_{i=1}^{ntoroidal}\left(s_4 + T_{scatter}\right) + s_5 + T'_{sendrecv}(mgrid) + T_{gather}(\cdot) \quad (10)$$

Type	Action	Weight	Description
Comp	s_1	$\Omega(mzeta \cdot (mgrid + mpsi))$	-
Comm	sendrecv	$mgrid$	Parallel Smoothing: send ϕ to right and receive from left
Comm	sendrecv	$mgrid$	Parallel Smoothing: send ϕ to left and receive from right
Comp	s_2	$\Omega(mgrid \cdot mzeta)$	-
Comm	sendrecv	$mgrid$	Toroidal BC: send ϕ to left and receive from right
Comp	s_3	$\Omega(mzeta \cdot (mgrid + mpsi))$	-
Comm	gather	$mtdiag \cdot mz \cdot (idiag2 - idiag1 + 1)$	Transpose a 2D-matrix from $(ntoroidal, mzeta \cdot mzbig)$ to $(1, mzetamax \cdot mzbig)$
Comp	s_4	$\Omega(ntoroidal \cdot mz)$	-
Comm	scatter	$mtdiag \cdot mz \cdot (idiag2 - idiag1 + 1)$	Transpose a 2D-matrix from $(ntoroidal, mzeta \cdot mzbig)$ to $(1, mzetamax \cdot mzbig)$
Comp	s_5	$\Omega(mgrid)$	Interpolate field
Comm	sendrecv	$mgrid$	Toroidal BC: send ϕ to right and receive from left
Comm	gather	$\frac{2 \cdot mtdiag}{ntoroidal} \cdot nummode$	Dominant (n, m) mode history data

TABLE V: Smooth

The optimizations due to overlapping of computation with the send and receive operations are similar to those described in the "field" phase. The transposition of the 2D-matrix involves gather and scatter operations equal to the number of toroidal domains.

$$T_{smooth} = \max\left(s_1, T_{sendrecv}(mgrid), T_{sendrecv}(mgrid)\right) + \max\left(s_2, T_{sendrecv}(mgrid)\right) + \sum_{i=1}^{ntoroidal}\left(s_3 + \delta_1 + s4 + \delta_2\right) + s_5 + \delta_3 + T_{gather}(\cdot) \quad (11)$$

where

$$T_{gather} = s_3 + \delta_1, \quad T_{scatter} = s_4 + \delta_2, \quad T'_{sendrecv} = s_5 + \delta_3$$

F. Push

In this phase, the particles are advanced using the field array computed in the previous phases. Since, this phase is dependent on the size of the particle arrays at each processor, the particle distribution among the processors determines the execution time of the computation steps that are bounded below by $\Omega(mi)$.

Type	Action	Weight	Description
Comp	p_1	$\Omega(mi)$	Runge-Kutta method
Comm	allreduce	8	Calculate total sum of weights
Comm	allreduce	4	Calculate total number of particles
Comp	p_2	$\Omega(mi)$	Out of boundary particle
Comm	allreduce	$2 \cdot mflux$	Restore temperature profile
Comp	p_3	$\Omega(mflux + mi)$	-
Comm	allreduce	$mpsi + 1$	Heat flux psi
Comm	allreduce	$2 \cdot mflux$	Compute marker,energy,particle
Comp	p_4	$\Omega(mpsi(mzeta + 1))$	Field energy
Comm	reduce	3	Total field energy from all toroidal domains

TABLE VI: Push

Table VI lists the computation and communication steps involved in this phase. Despite having a higher arithmetic intensity, the communication operations in this phase are

relatively expensive, making it a critical phase in the algorithm. The grid data accesses made by tasks $\{p_1, \ldots, p_4\}$ are irregular and results in bad cache behavior without any data reorganization.

$$T_{push} = p_1 + T_{allreduce}(8) + T_{allreduce}(4) + p_2 + 2 \cdot T_{allreduce}(2 \cdot mflux) + p_3 + T_{allreduce}(\cdot) + p_4 + T_{reduce}(\cdot) \quad (12)$$

The time taken to execute this phase in GTC (Eqn 12) and that to execute in GTCX (Eqn 13) are shown.

$$T_{push} = \max\left(p_1, T_{allreduce}(8), T_{allreduce}(4)\right) + \max\left(p_2 + \delta_1, p_3 + \delta_2\right) + T_{allreduce}(2 \cdot mflux) + p_4 + T_{reduce}(\cdot) \quad (13)$$

where

$$T_{allreduce}(2 \cdot mflux) = p_2 + \delta_1, \quad T_{allreduce}(mpsi + 1) = p_3 + \delta_2$$

G. Shift

The shift phase actually moves particles between toroidal domains. This involves communication of the number of particles to be shifted, the packing and shifting of particles to neighboring toroidal domains (using MPI's `sendrecv` operation), and unpacking of particles. The particles are moved in the $\pm zeta$ direction only one domain at a time.

Type	Action	Weight	Description
Comp	s_1	$\Omega(mi)$	-
Comm	allreduce	4	Total number of particles to be shifted
Comp	s_2	$40 \cdot \Theta(1)$	Pack particles and fill holes
Comm	sendrecv	8	Send number of particles to move right
Comm	sendrecv	$msendright$, $mrecvleft$	Send particle to right and receive from left
Comm	sendrecv	8	Send number of particles to move left
Comm	sendrecv	$msendleft$, $mrecvright$	Send particle to left and receive from right
Comp	s_3	$20 \cdot \Theta(1)$	Unpack particles

TABLE VII: Shift

The time required for GTC (T_{shift}) is simply the sum of the times required for the communication and computation steps shown in Table VII.

$$T_{shift} = s_1 + T_{allreduce}(4) + s_2 + 2 \cdot T_{sendrecv}(8) + 2 \cdot T_{sendrecv}(msend, mrecv) + s_3 \quad (14)$$

As shown in Eqn 15, the computation step s_1 cannot be overlapped with the first allreduce operation. Since the time to send the number of particles to shift is negligible, it is not accounted by the overlapping of the time to move the particle data.

$$T_{shift} = s_1 + T_{allreduce}(4) + 2 \cdot T_{sendrecv}(msend, mrecv) + s_3 \quad (15)$$

where

$$T_{sendrecv}(msend, mrecv) = T_{sendrecv}(8) + \delta$$
$$T_{sendrecv}(8) \ll T_{sendrecv}(msend, mrecv)$$

Parameter	Value	Parameter	Value
nints	36	micell	2
nreals	28	ntoroidal	1 to 256
mzetamax	64	npartdom	1
mpsi	90	nsteps	150
mthetamax	640	numberpe	1 to 256
mgrid	32449	mi	64718 to 258872

TABLE VIII: Parameters used for validating the model.

H. Model Validation

To quantify the performance benefits of removing global barriers and overlapping computation steps in GTCX, we evaluated our model with the parameters shown in Table VIII.

Figure 9 shows a strong-scaling plot of GTC against GTCX for an assumed, fixed degree of overlap (δ) for each of the phases. The execution times for the computation and communication steps of GTC were determined empirically by the output data of an actual run. The model errors were found to be within 15% of the execution time.

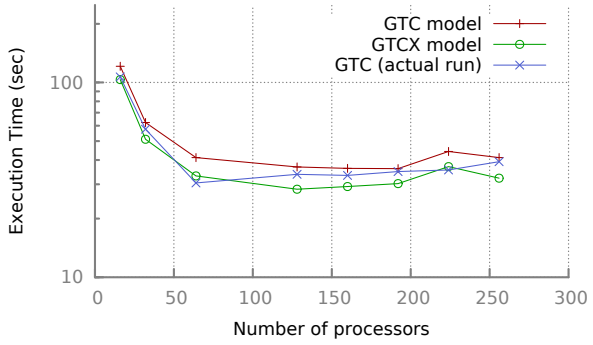


Fig. 9: Comparative analysis of the GTC and GTCX performance model.

VI. Conclusions

Implementing GTC in HPX-3 and SST/macro for HPX-3 has highlighted some key characteristics of many-tasking runtime systems while at the same time exposing some performance deficiencies. The removal of global barriers, the increase in overlapping phases of computation, and the presence of implicit load balancing all helped to extract more parallelism in GTCX while the increased overhead due to oversubscription and context switching mitigated the impact of those improvements. The legacy migration application path for GTCX enabled overlapping computational phases and intelligent scheduling of threads but did not take advantage of any code re-writes or data restructuring that would have more directly benefited from a many-tasking execution model. The GTCX and GTC code performance generally resembled each other both in the full implementation and in SST/-macro with only small performance and scaling gains registered in GTCX at the area where GTC had already stopped scaling. Comparative analysis based on the performance model enabled us to quantify the benefits due to overlapping computation phases.

Acknowledgments

We would like to thank Gilbert Hendry and Hartmut Kaiser for their technical assistance.

References

[1] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, "Oversubscription on multicore processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –11.

[2] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.64

[3] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[4] T. El-Ghazawi, F. Cantonnet, and Y. Yao, "Evaluations of UPC on the Cray X1," in *CUG 2005 Proceedings*, New York, NY, USA, 2005, p. 10. [Online]. Available: http://www.gwu.edu/~upc/publications/cug05.pdf

[5] F. Cappello and D. Etiemble, "Mpi versus mpi+openmp on the ibm sp for the nas benchmarks," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 12–12.

[6] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur, "Hybrid parallel programming with mpi and unified parallel c," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787323

[7] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, 1st ed. O'Reilly Media, Jul. 2007.

[8] M. D. McCool, A. D. Robison, and J. Reinders. (2012) Structured parallel programming patterns for efficient computation. Waltham, MA.

[9] 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*, sept. 2009, pp. 394 –401.

[10] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling, "Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model," *International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 319–332, 2012. [Online]. Available: http://hpc.sagepub.com/content/26/3/319.abstract

[11] 2012, http://cilkplus.org/.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[13] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[14] C. Yang, K. Murthy, and J. Mellor-Crummey, "Managing asynchronous operations in coarray fortran 2.0," in *Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[15] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for Programming with Millions of Lightweight Threads," in *International Parallel and Distributed Processing Symposium. IEEE Press*, 2008.

[16] C. Hewitt and H. G. Baker, "Actors and continuous functionals," Cambridge, MA, USA, Tech. Rep., 1978.

[17] J. Robert H. Halstead, "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, 1985.

[18] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress language specification, version 1.0," March 2008.

[19] 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 cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, October 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[20] HPC University and the Ohio Supercomputer Center, "Report on high performance computing training and education survey," available from http://www.teragridforum.org/mediawiki/images/5/5d/HPCSurveyResults.FINAL.pdf.

[21] T. Stitt and T. Robinson, "A survey on training and education needs for petascale computing," available from http://www.prace-project.eu/IMG/pdf/D3-3-1_document_final.pdf.

[22] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu, "Parallex: A study of a new parallel computation model," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–6.

[23] S. Ethier, W. M. Tang, and Z. Lin, "Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 1, 2005. [Online]. Available: http://stacks.iop.org/1742-6596/16/i=1/a=001

[24] D. Tskhakaya, "The particle-in-cell method," in *Computational Many-Particle Physics*, ser. Lecture Notes in Physics, H. Fehske, R. Schneider, and A. Weie, Eds. Springer Berlin Heidelberg, 2008, vol. 739, pp. 161–189. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74686-7_6

[25] K. Antypas, J. Shalf, and H. Wasserman, "Nersc-6 workload analysis and benchmark selection process," National Energy Research Scientific Computing Center Division Ernest Orlando Lawrence Berkeley National Laboratory, Tech. Rep. LBNL 1014E, August 2008. [Online]. Available: http://www.nersc.gov/assets/pubs_presos/NERSCWorkload.pdf

[26] S. Ethier, W. M. Tang, and Z. Lin, "Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 1, 2005. [Online]. Available: http://stacks.iop.org/1742-6596/16/i=1/a=001

[27] X. Wu and V. Taylor, "Performance modeling of hybrid mpi/openmp scientific applications on large-scale multicore cluster systems," in *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, 2011, pp. 181–190.

[28] G. Hendry and A. Rodrigues, "Sst: A simulator for exascale co-design," in *Proc. of the ASCR/ASC Exascale Research Conference*, 2012.

[29] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[30] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A comparison of context-oriented programming languages," in *International Workshop on Context-Oriented Programming*, ser. COP '09. New York, NY, USA: ACM, 2009, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/1562112.1562118

[31] T. Gilmanov, M. Anderson, M. Brodowicz, and T. Sterling, "Application characteristics of many-tasking execution models," in *Proc. of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2013.

[32] S. Olivier and J. F. Prins, "Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs." *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 341–360, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijpp/ijpp38.html#OlivierP10

[33] K. Madduri, K. Z. Ibrahim, S. Williams, E.-J. Im, S. Ethier, J. Shalf, and L. Oliker, "Gyrokinetic toroidal simulations on leading multi- and manycore hpc systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 23:1–23:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063415

[34] A. Koniges, R. Preissl, J. Kim, D. Eder, A. Fisher, N. Masters, V. Mlaker, S. Ethier, W. Wang, M. Head-Gordon, and N. Wichmann, "Application Acceleration on Current and Future Cray Platforms," in *CUG 2010, the Cray User Group meeting*, May 2010.

[35] T. Hoefler, W. Gropp, M. Snir, and W. Kramer, "Performance Modeling for Systematic Performance Tuning," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), SotP Session*, Nov. 2011.

[36] G. Hendry, "Decreasing Network Power with On-Off Links Informed by Scientic Applications," in *the Ninth Workshop on High-Performance, Power Aware Computing*, May 2013.

[37] T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," in *In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05) - Workshop 15*, 2005.

[38] M. M. Mathis, D. J. Kerbyson, and A. Hoisie, "A performance model of non-deterministic particle transport on large-scale systems," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 324–335, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.future.2004.11.018

[39] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - simulating large-scale applications in the LogGOPS model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, Jun. 2010, pp. 597–604.

[40] G. Hendry and A. Rodrigues, "Simulator for exascale co-design," available from http://sst.sandia.gov/publications.html.

[41] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Comput.*, vol. 20, no. 3, pp. 389–398, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(06)80021-9