SAND2014-17445C

# Toward an Evolutionary Task Parallel Integrated MPI + X Programming Model

Richard F. Barrett, Dylan T. Stark, Courtenay T. Vaughan, Ryan E. Grant, Stephen L. Olivier and Kevin T. Pedretti
Sandia National Laboratories*
Albuquerque, NM, USA
Email: rfbarre,dstark,ctvaugh,regrant,slolivi,ktpedre@sandia.gov

***Abstract*—This paper presents some results from a research project designed to address some performance issues attributable to the bulk synchronous parallel programming model seen at large processor counts. Over-decomposition of the domain, operated on as tasks, is designed to smooth out utilization of the computing resource, in particular the node interconnect and processing cores, and hide intra- and inter-node data movement. Our approach maintains the existing coding style commonly employed in computational science and engineering applications. Although we show improved performance on existing computers, the effectiveness of this approach on expected future architectures will require the continued evolution of capabilities throughout the codesign stack. Success then will not only result in decreased time to solution, but would make better use of the hardware capabilities and reduce power and energy requirements, while fundamentally maintaining the current code configuration strategy.**

***Index Terms*—scientific and engineering applications; high performance computing; programming models.**

## I. Introduction

The Bulk Synchronous Parallel programming model (BSP) is the dominant strategy for implementing high performance portable parallel processing scientific and engineering applications. As widely available parallel processing architectures focused node interconnect performance on bandwidth (relative to latency), code developers often aggregated data from various structures into single messages [3] in order to reduce the communication cost. Although many such applications have performed well even up to peta-scale, the situation appears to be changing. For example, as illustrated in Figure 1, when using a simple difference stencil code from the Mantevo suite, we are seeing degradations in scaling performance, found to be caused by very small synchronization effects cascading as the number of MPI ranks increased, exacerbated by the "bursty" use of the interconnect. Briefly, with more details below, this figure represents the performance of a difference stencil code. Careful attention to process placement moved the scaling degradation to the right, but this can be a significant hindrance to throughput in a shared resource production environment and may not be tractable for other algorithms. Regardless, this bursty communication strategy requires increasing global bandwidth support, a meaningful impediment to the need to reduce power and energy consumption, a stated requirement of exascale roadmaps [13].

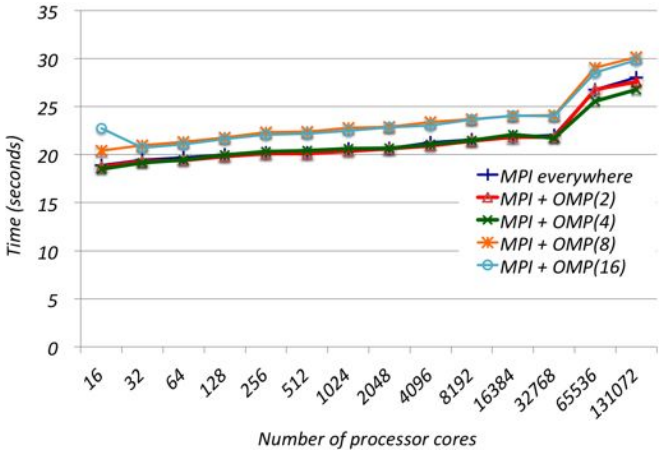



Fig. 1. Difference stencil code weak scaling on a Cray XE6. MPI + OMP($n$) represents $n$ OpenMP threads for each of the (Number of processor cores / $n$) MPI processes.

This has led to a renewed focus on non-traditional architectures, providing new opportunities for exploring alternatives to this programming approach. The emerging trend of lighter-weight cores, each with access to a decreasing per core amount of memory, combined with the expanding capabilities of the interconnect compels a rethinking of basic programming strategies, with special emphasis on inter-process communication.

In this paper we report on our work in developing a task parallel programming approach to "MPI + X" [24] that combines computation and communication requirements into a task in a way that maintains the traditional coding style, yet spreads the workload more evenly across the computing resource in comparison to the bulk synchronous parallel model. This is enabled by exposing a greater level of parallelism by dividing the workload into smaller pieces, presented to a task model that couples computation and communication requirements, allowing the runtime system to manage the work as a function of the computing environment capabilities.

Task parallel approaches have been available previously

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

to code developers, but acceptance has been limited for a variety of reasons, perhaps most strongly by the requirement of significantly rewriting code. We demonstrate our approach using the ubiquitous difference stencil. Although a relatively simple computation, it composes a broad range of modeling approaches of physical phenomena in science and engineering, and leads to optimism for the principles being applied to more challenging algorithms.

## II. Related Work

Over-decomposition strategies are being effectively applied in linear algebra algorithms, where it is commonly referred to as *tiling* [25], [15].

Unlike these computations, difference stencil algorithms have no meaningful data re-use. Further, they continue to maintain a strong separation between computation and communication phases. We break up the workload of each beyond the core view and provide a node view to the smaller individual workloads. By including inter-process communication requirements within a task, the task can be swapped out while the inter-MPI rank data is moved, new work swapped in, and the MPI work swapped back in once the data motion is complete.

The performance of stencil computations has been an ongoing focus area in high performance scientific computing, for example as described in [2], [7], [9], [10], [21]. In fact many of these ideas could be incorporated into our approach.

The advent of multicore processors compelled the increase of injection rates and bandwidth from the node onto the interconnect. This inspired and enabled our recent work in message passing strategies: rather than aggregate data from each of say 40 variables into single messages (in a multi-material physics code), we performed the halo exchange as each variable was updated, resulting in a 40 times increase in the number of messages, each 1/40th the size [5]. Our work herein takes this approach to a much more challenging level.

A large body of past work has addressed computation and communication overlap [16], [14], [11] in MPI. Our work leverages the benefits of such overlap as a consequence of its task-parallel approach. Past approaches to automatically provide overlap [22] have been proposed. Our approach using tasks provides similar benefits without the need for manual or automatic code transformation, or runtime methods of overlap prediction and scavenging.

Prior work supporting dynamic parallelism and MPI message passing includes integration of an on-node task parallel runtime with MPI with SMPS [18] or Habanero [8], and with use of application-level frameworks such as Uintah [19]. Our work is closely related to these approaches, though it is more limited in focus than either of these two efforts. We deliberately avoid imposing a broader task parallel programming model on the user, and we do not require language extensions or compiler support. Both decisions were made to decrease any re-writing of existing MPI codes to take advantage of special MPI calls or features of a particular model. Likewise, approaches like Uintah also requires code rewriting, leaving legacy MPI code behind and recasting algorithms to their high-level framework.

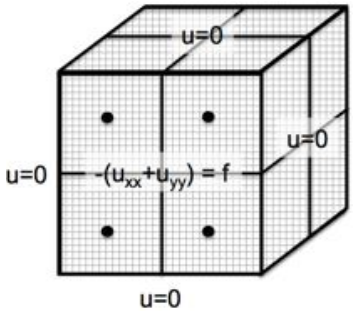
## III. A Task Parallel Over-Decomposition Strategy for Finite Difference Stencils

The algorithmic structure of the finite difference method maps naturally to the parallel processing architecture and single-program multiple-data (SPMD) programming model. On parallel processing architectures, these stencil computations require data from neighboring processes. This inter-process communication is typically abstracted into some sort of functionality that may be loosely described as *boundary exchange* (also called ghost-exchange or halo-exchange). The general form of this may be expressed as

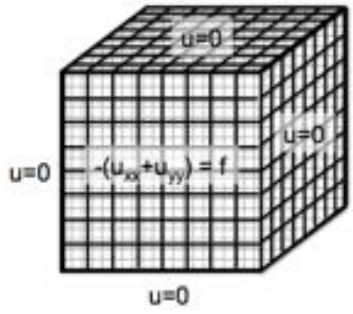
```
int stencil ( ... )  {
   Exchange_boundary_data ( ... );
   Apply_boundary_conditions ( ... );
   Apply_stencil ( ... );
}
```

including the application of physical boundary condition computation.

A data-parallel implementation of difference stencils decomposes the domain such that each core is assigned a single contiguous block, illustrated in Figure 2(a). This requires




(a) Data parallel




(b) Task parallel over-decomposition

Fig. 2. Domain decomposition approaches to finite differencing

that the communication requirement be completed before the computation may begin.

The task-parallel version over-decomposes the domain that has been assigned to a set of cores sharing memory, and relies on a dynamic runtime system to schedule blocks to a set of processing cores spanning a shared memory region, illustrated

in Figure 2(b). This approach maintains the basic form of these operations, allowing computation and communication requirements to be included in the same task, but with the individual task workloads significantly reduced. Here, when a communication event reaches a blocking state, the task may be swapped out, allowing another task to be swapped in. When that task completes, or is itself swapped out while blocking on a communication event, another task may be swapped in, and so on. For a high enough level of over-decomposition, most tasks will require computation only, and thus quickly complete. At some point, the communication-blocked task is swapped back in and its work continues. This process is illustrated in Figure 3. We abbreviate this task parallel over-

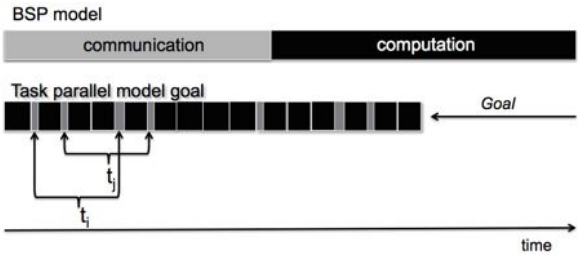



Fig. 3. Computation and communication interaction

decomposition programming model as *TPOD*.

The amount of over-decomposition, defined by the size of the subblocks, is chosen as a balance of processor capabilities, memory hierarchies, node interconnect capabilities, and the runtime scheduling system's ability to manage the tasks.

In this work, we employ miniGhost [6], a miniapp from the Mantevo suite (mantevo.org) that models heat diffusion across a homogenous three dimensional surface by applying difference stencils. It has been demonstrated to represent the scaling behavior of full application programs (for example, [4]) and has been previously used to explore alternative inter-communication strategies [5].

Miniapplications provide a proven application-relevant context for our experiments [4]. Designed for modification and experimentation, they are open source, self-contained, stand-alone code, with simple build and execution systems. They enable investigation of different programming models and mechanisms, existing, emerging, and future architectures, and enabling investigation of entirely new algorithmic approaches for achieving effective use of the computing environment within the context of complex application requirements.

### A. Code Discussion

The computational science and engineering community has a significant investment in application programs. For example, the United States Department of Energy has a multi-billion dollar investment across many years of application development, resulting in application programs that exceed one million source lines of code. Many of these applications make strong use of stencil computations. Our task-based model requires only minor reconfiguration, leaving the fundamentals in-place: loops over blocks of grid points with nearest neighbor interprocess communication of face data. That is, this is an evolutionary approach with the potential for revolutionary consequences. Moreover, rooted in the fundamentals of threading, this model provides a natural mapping to different architectures (e.g. many-core, gpu-acceleration, etc) and programming mechanisms (e.g. OpenMP, CUDA, etc).

The definition and ordering of the over-decomposition is abstracted to an initialization function, say `Over_decompose`, resulting in code that looks like this

```
ierr = Over_decompose ( blks,  ... );
for ( i=0; i<numblks; i++ )
    spawn ( stencil ( blks[i], ... ));
```

Now rather than constraining the ordering of the application of the stencil (typically to an `(i,j,k)` nesting), the ordering is defined within this function, allowing exploration of different strategies for presenting the workload to the scheduler.

For the results presented herein, we used a simple random shuffling of the block order, with the goal of spacing out the tasks that include communication while requiring nothing from the code developer. Additional orderings, defined for example in terms of a directed acyclic graph (DAG), perhaps informed by some known workload characterization, could be inserted. The code developer now has a higher level view of the computation, looping over sets of grid points defined by the blocks, with the actual computation and communication code remaining the same (with the exception of the new indexing set).

This does require a modification to the starting and ending indices. Whereas the data parallel version sweeps across all grid points, and thus indices span the entire dimension from $1, \ldots, n$ (the $0^{th}$ and `n+1` indices are ghost space), over-decomposition of miniGhost requires defining task workload indices. Similar indexing would be needed for the application of boundary conditions. But again, the fundamental structure of user code remains unchanged.

### B. MPI+X Discussion

In order to evaluate the potential of the task parallel over-decomposition approach described herein, we extended the Qthreads task-parallel runtime system [26] to support direct calls to MPI from within tasks (user-level threads). Qthreads supports a variety of task creation and synchronization constructs, but does not require that the user adopt language extensions or robust programming models, such as with Cilk and other approaches. In fact, TPOD requires only two calls to the Qthreads C API: a straightforward spawn function (`qthreads_spawn`) creates new tasks, and a task-level barrier (`qt_sinc_wait`) enforces task completion for each time step.

This integration of MPI+Qthreads (MPIQ) manages the scheduling of tasks across the available cores, independent of the application. Tasks without communication can execute on any core, with a work stealing scheduler ensuring that no core is idle while there is more available work. Tasks that include communication must be scheduled with respect

to the thread support level provided by the MPI implementation. MPIQ does not require a specific thread level and will adapt the scheduling of tasks with communication to guarantee correct usage of MPI. For instance, when running with `MPI_THREAD_FUNNELED` support, a task that attempts to call into MPI will be migrated by the runtime to the execution context where MPI was initialized. Because TPOD may induce considerably more concurrent tasks with MPI calls than parallel execution contexts, MPIQ can swap out tasks on blocking MPI calls. Further, because a correct ordering of those calls that guarantees no dead- or live-locking is not known by the runtime system, blocking MPI calls are converted to their non-blocking equivalents. These are then managed using a combination of cooperative yielding of tasks while waiting for the request to be satisfied. If an MPI function does not have non-blocking equivalent, we create auxiliary OS threads that can manage the blocking events without starving a core.

The TPOD approach provides multiple opportunities for reducing MPI communication costs. It spreads communication over a longer time period, lessening network pressure. This allows for shorter communication queues, which can improve receive side message matching performance. It also facilitates computation/communication overlap, which is a factor in improving overall runtime. Finally, this approach integrated with the runtime/tasking system can leverage the fast-path in MPI implementations by using `MPI_THREAD_FUNNELED` mode instead of `MPI_THREAD_MULTIPLE`. This helps to avoid much of the locking inherent in multi-threaded MPI implementations, which in turn provides performance similar to that of single threaded MPI (and for some implementations `MPI_THREAD_FUNNELED` is identical to `MPI_THREAD_SINGLE`). This is an important factor in ensuring that this approach will be applicable to future systems, as the performance of traditional single-thread MPI implementations is well known to be excellent at scale. Additional details of the MPI + X integration can be found in [23].

## IV. Experimental platforms

New high performance architectures are showing signs of increased thread parallelism, NUMA effects, heterogeneity, and network opportunities and restrictions. We selected two systems, Cielo and Volta, that we see representing these issues and their expansion onto future architectures. These machines allow us to begin to evaluate the model and answer fundamental scaling questions related to task granularity. For example, what is the appropriate abstraction between the application and the system? And how do we best control computation and communication granularity for effective system utilization?

Cielo [12], an instantiation of a Cray XE6, is composed of 8,944 compute nodes, connected using a Cray custom interconnect named Gemini, and a light-weight kernel (LWK) operating system called Compute Node Linux. Each compute node, illustrated in Figure 4, consists of two oct-core AMD Opteron Magny-Cours processors, for a total of 143,104 cores. Each core has a dedicated 64 kByte L1 data cache, a 64 kByte L1 instruction cache, and a 512 kByte L2 data cache, and the cores within a NUMA node share a 6 MByte L3 cache (of which 5 MBytes are user available). Each Magny-Cours processor is divided into two memory regions, called NUMA nodes, each consisting of four processor cores and 8 GBytes of DDR3 1333 MHz memory. Thus each compute node consists of 16 processor cores, 32 GBytes of memory, evenly divided among four NUMA nodes, which are connected using HyperTransport version 3. The links between NUMA nodes run at 6.4 GigaTransfers per second (GT/s).

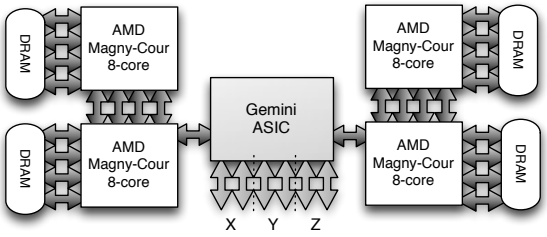



Fig. 4. Cielo node architecture

Cielo compute nodes are connected using Cray's Gemini 3-D torus interconnect. Relative to its SeaStar predecessor used in the Cray XT series, Gemini provides an improvement to the achievable asymptotic bandwidth for point-to-point communication. There are two potential bottlenecks to consider: injection bandwidth and link bandwidth. Injection bandwidth is limited by the speed of the Opteron to Gemini HyperTransport link, which runs at 4.4 GT/s.

The computing environment was set using module `PrgEnv-intel/4.2.34`, which includes the Intel icc compiler version 14.0.02 (using `-O3 -fast`) and the Cray MPI implementation `cray-mpich/7.0.1`.

Volta, a Cray XC30 Cascade system, consists of 52 compute nodes, connected using a Cray Aries custom interconnect [1], at this scale configured with all-to-all connectivity. Each node is composed of two 12 core Intel Xeon E5-2670 Sandy Bridge processors, for a total of 1,248 cores, running the CLE operating system. Each core has a dedicated 32KB L1 data cache, a 32KB L1 instruction cache, and a 256KB L2 data cache, and the cores within each socket share a 20 MB L3 cache. The computing environment was set using module `PrgEnv-intel/5.2.25`, which includes the Intel icc compiler version 14.0.01 (using `-O3 -fast`) and the Cray MPI implementation `cray-mpich/7.0.1`.

## V. Experiments

Our focus here is on Cielo, which allows us to scale up to processor counts where we see the performance issue with the BSP implementation. Volta provides a smaller scale alternative for additional data points. Future work targets additional platforms, including a large scale Cray XC, where the Aries interconnect is configured with a dragonfly topology [17].

We configured a weak scaling problem set that consumes a large proportion of the memory available in each shared

| Block dimension | Cache lines per dim | Number of blocks | Blocks per face | Msg size (bytes) |
|---|---|---|---|---|
| $720, 360^2$ | 90,45 | 4 | 2,4 | 4e6,1e6 |
| $360^3$ | 45 | 8 | 4 | 1.04e6 |
| $256^3$ | 32 | 8 | 4 | 5.24e5 |
| $128^3$ | 16 | 125 | 25 | 1.31e5 |
| $64^3$ | 8 | 1331 | 121 | 3.28e4 |
| $32^3$ | 4 | 10648 | 484 | 8.19e3 |
| $24^3$ | 3 | 27000 | 900 | 4.61e3 |
| $16^3$ | 2 | 91125 | 2025 | 2.05e3 |
| $8^3$ | 1 | 729000 | 8100 | 5.12e2 |

TABLE I
OVER-DECOMPOSITION SPECIFICATIONS

memory region, i.e. a socket or NUMA node, and assigned one MPI rank to each. The problem was solved using different amounts over-decomposition, as defined by the sub-blocks (three dimensional cubes) of the full domain assigned to each MPI rank. Although not necessary for our regular grid, we focus on cubic blocks in preparation for Adaptive Mesh Refinement (AMR), where cubes are a natural structure of computation. Block dimensions are defined as factors of a cache line, ranging from a single line (64 bytes, or 8 grid points in double precision, on Cielo) in each dimension, increasing at a rate of one cache line in each dimension, up to half of the dimension defined for the MPI rank. We apply a 7-point stencil in three dimensions, and therefore from the MPI perspective there are six inter-process communication partners (fewer, of course, on physical boundaries). Once the over-decomposition creates a reasonable number of sub-blocks, an individual block will have at most three communication partners.

A global barrier is inserted at the end of each time step to represent some sort of reduction operation expected there, but without intruding on the basic stencil computation. The Dirichlet boundary conditions require a negligible amount of computation, used here only to ensure correctness: flux out plus the current sum across the grid must remain equal to the initial conditions.

For this example, we assign a grid of size $720 \times 720 \times 720$ to each MPI rank (for a variety of reasons, this is different from the problem set used in Figure 1) and iterate for 20 time steps. This consumes a significant amount of memory (more than six of the available eight GBytes) while also providing a significant number of evenly divisible over-decomposition factors. Table I lists some details of interest for the decomposition dimensions we present here.

As discussed above, ultimately our work is motivated by our understanding of future architectures. Therefore the MPI-everywhere model, whereby one MPI rank is assigned to each processor core, is not seen a viable option.

The figure of merit for understanding the overall parallel performance of a finite difference algorithm is grind time, defined as

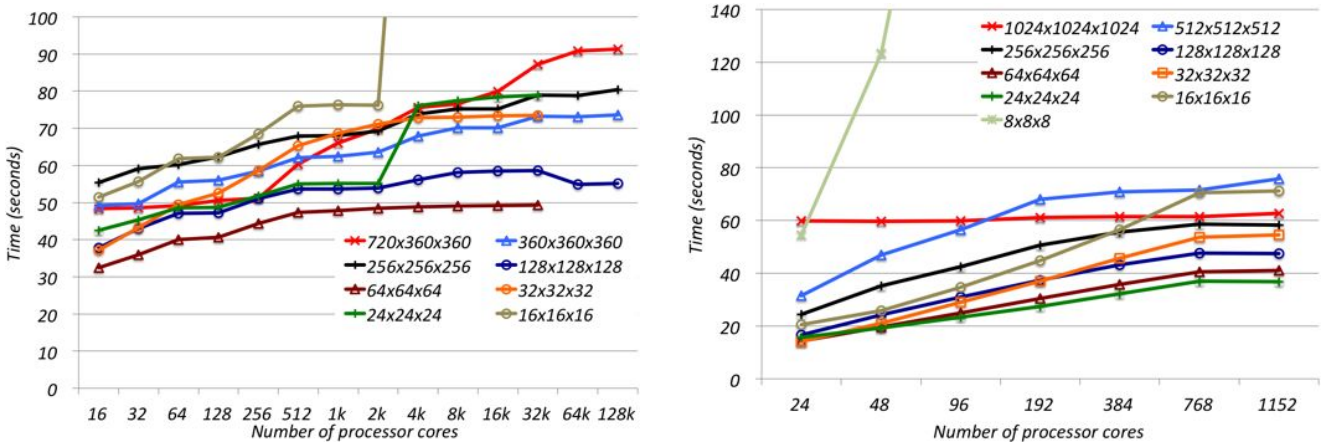
$$g = t/n$$

for $g$ = grind time, $t$ = total time, and $n$ = total number of grid points. That is, this is the time to operate on an individual grid point, and therefore good performance results in a decreasing grind time. This is also a useful metric for this particular work, since we configure all decompositions to be even factors of the block dimension (again, the natural approach for AMR). So for example, for a block dimension of 64, there will be 11 blocks in each dimension, resulting in a domain of $704 \times 704 \times 704$ assigned to each MPI rank. This also means that in order to make a fair comparison of scaling, we multiply grind time by the same number of grid points for each over-decomposition.

Figure 5 shows the performance of these over-decomposition configurations. For this weak scaling experiment, the workload increases linearly with the computing resource, and so perfect scaling is a flat line. For small amounts of over-decomposition there is little opportunity for communication overlap, though still, performance improves. As the over-decomposition increases, overlap opportunities increase, and the overall performance improves to its best point with the $64^3$ decomposition, when time begins to increase. The $16^3$ decomposition performs relatively well at the beginning, but then degrades terribly at 4k cores.

In comparison with the MPI + OpenMP version in Figure 1, where scaling degrades at high core counts (this with careful process mapping), the TPOD approach stays flat, and this without the attention to process mapping. We see similar behavior on Volta (Figure 5(b)), except that the single-block case starts, and remains, flat, indicating the increased capabilities of the interconnect and the processor. (Plans calls for testing this trend at much higher scales on a much larger Cray XC.)

A typical next step in examining the behavior of a BSP model code is to examine computation and communication times. Here, though, this exposes some issues regarding deeper profiling of the TPOD model. In the BSP model, its relatively easy to capture, interpret, and understand the behavior of computation and inter-process communication requirements *because* of the synchronous execution model. But the TPOD model, by design, has a far more complex execution model, so the standard timing method of starting and stopping timers surrounding the computation and communication activities will include things not seen in the BSP model. For example, as illustrated above in Figure 3, the goal of hiding communication costs by swapping tasks due to communication events would result in communication times reported that exceed the total run time. And we see this effect for these experiments, graphically illustrated in Figure 6. This does not really provide much actionable information for understanding execution, but it does provide confidence that our task swapping approach is resulting in communication hiding. It also illustrates the interaction between computation and communication as over-decomposition changes. The computation cost increases with increasing over-decomposition, as does the reported communication time, but the total run time decreases to a point, here $64^3$, at which point the ability to hide the increasing




(a) Cielo (b) Volta

Fig. 5. Weak scaling performance

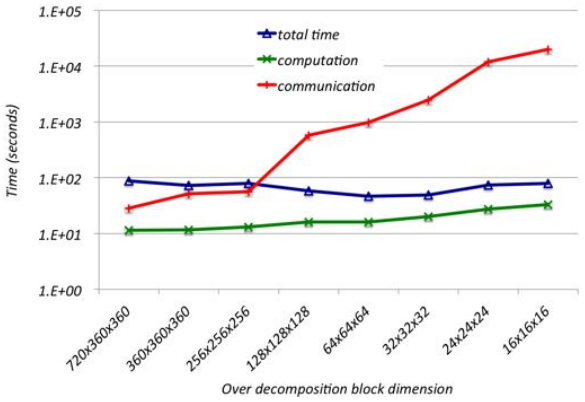



Fig. 6. Computation and communication for 32k processor cores on Cielo

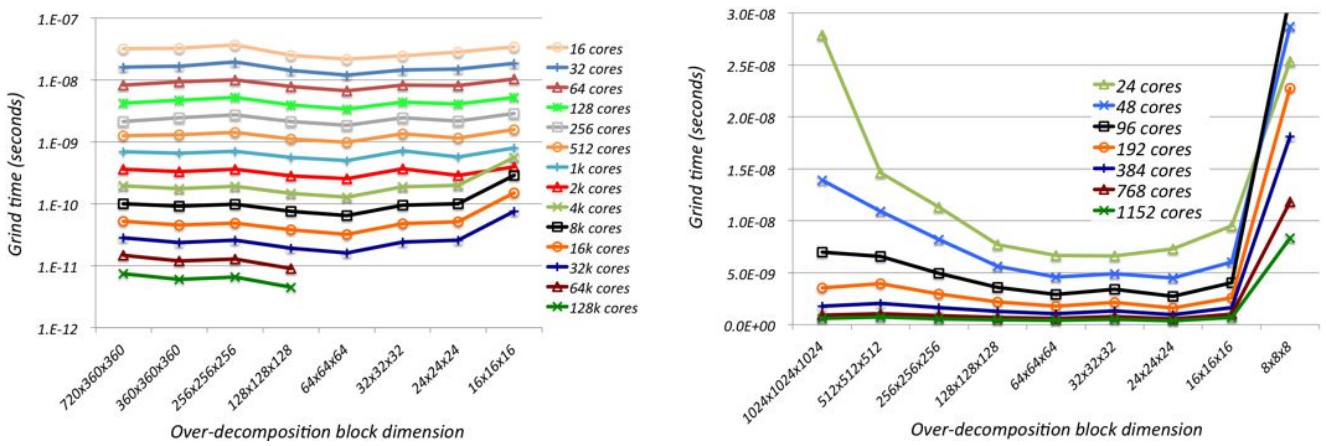
communication cost is diminished.

A different view of this performance data is shown in Figure 7. Here, each line represents a fixed number of cores and grid size, with the x-axis varying the over-decomposition block dimension, so that we may see the impact of the over-decomposition on the grind time. In some sense, then, each line may be considered strong scaling and the different lines weak scaling. We see that for many block dimensions, the grind time decreases as over-decomposition increases, possible only if some communication is being hidden by computation.

One of the key goals of our over-decomposition scheme is to spread network communication over a longer period of time, which for a constant problem size will result in lower point-in-time load on the network. Doing this should reduce the network bandwidth requirement for a given problem size and also reduce network congestion, which when present can substantially increase message latency and overall communication time. We examined the Cray Gemini network performance counters [20] in an effort to empirically measure whether or not these benefits were being achieved in practice.

Results suggest that point-in-time network load is being reduced. First, we examined the per-link bytes transmitted counter to measure the total communication volume injected into the network for each block size, which corresponds to the total bytes (including header data) sent by all MPI ranks. Illustrated in Figure 8(a), successively larger core counts result in larger communication volume, causing the curves in the figure to stack nicely on one another. The large dips for the $256^3$ and $128^3$ block sizes, attributable to their overall problem size being approximately 2.5 and 1.5 times smaller due to the cache line factor blocks, show that network traffic decreases by the expected amount for the smaller problem sizes, providing confidence that these counters are working properly. For the actual over-decomposition targets on the right half of the graph, the general trend is for the bytes injected to increase slightly for successively finer-grained over-decomposition dimensions. This is expected since increased over-decomposition should result in a larger number of smaller messages, and each message has a fixed overhead cost (i.e., the message headers and other per-message wire overhead). The empirical data shows this effect, confirming that per-message overhead on the Cray Gemini network is reasonable.

Next, we examined network stalls per byte transmitted, a proxy for network congestion. In general, as more load is placed on a sparse network like the Gemini's 3-D torus, more head of line blocking occurs, more network router buffers fill up, and higher message latencies result. Congestion causes stalls, meaning messages must wait in a router buffer before they can continue to their next hop in the network. Measuring total stalls at all allocated Gemini network ASICs, if over-decomposition spreads out communication traffic, we expect to see a reduced point-in-time network load and fewer stalls. We compute the average number of stalls by dividing measured bytes injected into the network. The general trend, illustrated in Figure 8(b), shows a decrease in average stalls as over-decomposition increases, suggesting that over-decomposition




(a) Cielo (b) Volta

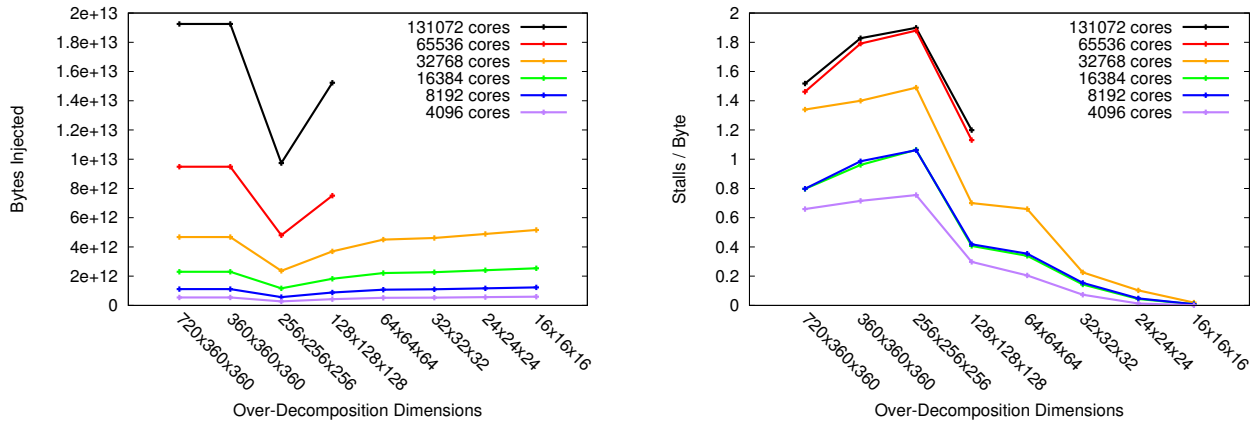
Fig. 7. Exploring block sizes

is having the desired impact.

## VI. Summary and Future Work

Scaling issues encountered using the BSP programming model on current architectures and inspiration from emerging and expected future architectures motivated an exploration of alternative coding strategies for some computational science and engineering applications of interest. The task parallel over-decomposition programming model provides a theoretically viable alternative, but we were intent on maintaining the general structure of existing application coding strategies. We were somewhat surprised to see the TPOD approach outperform the BSP model on current, traditional cluster-like architectures. Although results presented herein were run in dedicated resource mode, i.e. no other programs were executing at the time, we also saw that performance was seemingly not adversely impacted on shared resources, the standard resource management model that production computing operates in. This provides encouragement that this approach will have an even greater impact on more challenging applications on more challenging architectures.

Looking forward to new architectures, we expect that the number of processor cores on a node will approach the hundreds and even thousands, each of lighter weight (e.g. slower clocks and less memory per core) than those on current multicore processors. Interconnect technologies, such as fat trees, dragonfly, etc, and future optical, are designed to provide more efficient bandwidth, but latencies are in some sense more constrained. Therefore the increasing core threading capabilities could be combined with an *off-load* model of inter-node communication to provide stronger support for the TPOD strategy. The tasks are spawned onto the node, assigned as a task to a thread on a core. If required, MPI communication is initiated by the core, but as much of the subsequent work is performed on a separate processor assigned the task: transmission and reception of the data, the source




(a) Bytes injected into 3-D torus (b) Ratio of network stalls per byte injected

Fig. 8. Measured network traffic for Cielo

and destination matching, etc. We further believe that this approach maps well to GPU architectures, which we intend to investigate as well. This will, however, present challenges with regard to moving data between the host and device if that model persists.

Our next application target involves Adaptive Mesh Refinement (AMR). AMR is a strategy for focusing attention on areas of intensive activity in computational science and engineering applications while still constraining the overall workload. This comes at the cost of added complexity, including more challenging computational and communication patterns. AMR provides a naturally changing workload that can, in a strong sense, be viewed and managed as domain over-decomposition, illustrated in Figure 9. We are also targeting other application

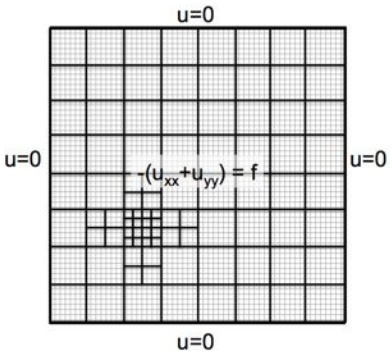



Fig. 9. AMR dynamic workload.

domains for applying this approach.

Our initial work has exposed questions that must be addressed if this approach is to provide a viable alternative for full application programs on new architectures. Stronger means for understanding run time behavior at a finer grain is required to make stronger causal relationships in this execution model. Strong integration of the inter-node and intra-node (MPI and "X", respectively) is necessary to effectively manage and coordinate communication and computation. Our future work will include addressing these and other issues so that we may fully explore the potential for this alternative programming model.

## REFERENCES


[1] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray XC Series Network. Technical Report WP-Aries01-1112, Cray Inc., 2012.

[2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC12, 2012.

[3] R.F. Barrett, S. Ahern, M.R. Fahey, R. Hartman-Baker, J.K. Horner, S.W. Poole, and R. Sankaran. A Taxonomy of MPI-Oriented Usage Models in Parallelized Scientific Codes. In *The International Conference on Software Engineering Research and Practice*, 2009.

[4] R.F. Barrett, P.S. Crozier, M.A. Heroux, P.T. Lin, T.G. Trucano, and C.T. Vaughan. Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications. *Journal of Parallel and Distributed Computing*, 2014. To appear.

[5] R.F. Barrett, C.T. Vaughan, S.D. Hammond, and D. Roweth. Reducing the Bulk of the Bulk Synchronous Parallel Model. *Parallel Processing Letters*, 23, December 2013.

[6] R.F. Barrett, C.T. Vaughan, and M.A. Heroux. MiniGhost: A Miniapp for Exploring Boundary Exchange Strategies Using Stencil Computations in Scientific Parallel Computing. Technical Report SAND2011-5294832, Sandia National Laboratories, May 2011.

[7] L. Bongo, B. Vinter, O. Anshus, T. Larsen, and J. Bjorndalen. Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors. In *Proceedings of the 2006 International Conference Workshops on Parallel Processing*, ICPPW '06, 2006.

[8] S. Chatterjee, S. Tasırlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. *Department of Computer Science, Rice University, Technical Report TR12-07*, 2013.

[9] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, pages 129–159, 2009.

[10] R. de la Cruz and M. Araya-Polo. Towards a multi-level cache performance model for 3D stencil computation. *Procedia CS*, 4:2146–2155, 2011.

[11] D.W. Doerfler and R. Brightwell. Measuring MPI send and receive overhead and application availability in high performance network interfaces. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 331–338. Springer, 2006.

[12] D.W. Doerfler, M. Rajan, C. Nuss, C. Wright, and T. Spelce. Application-Driven Acceptance of Cielo, an XE6 Petascale Capability Platform. In *Proc. 53rd Cray User Group Meeting*, 2011.

[13] J. Dongarra and P. Beckman et. al. The International Exascale Software Roadmap. *International Journal of High Performance Computing*, 25(1), 2011.

[14] R.L. Graham et al. Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010.

[15] B. Hadri, E. Agullo, and J. Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *24th IEEE International Parallel and Distributed Processing Symposium (submitted)*, 2010.

[16] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'07)*, 2007.

[17] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[18] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

[19] Q. Meng and M. Berzins. Scalable Large-Scale Fluid-Structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms. *Concurrency and Computation: Practice and Experience*, 26(7):1388–1407, 2014.

[20] K.T. Pedretti, C.T. Vaughan, K.S. Hemmert, and R.F. Barrett. Using the Cray Gemini Performance Counters. In *Proc. 55th Cray User Group Meeting*, 2013.

[21] S. Rahman, Q. Yi, and A. Qasem. Understanding Stencil Code Performance on Multicore Architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10, New York, NY, USA, 2011. ACM.

[22] M.J. Rashti and A. Afsahi. A speculative and adaptive MPI rendezvous protocol over RDMA-enabled interconnects. *International Journal of Parallel Programming*, 37(2):223–246, 2009.

[23] D.T. Stark, R.F. Barrett, R.E. Grant, S.L. Olivier, K.T. Pedretti, and C.T. Vaughan. A Dynamic Runtime with Co-Scheduling of Work and Communication Tasks for Hybrid MPI+X Applications. Under review.

[24] R. Thakur et al. MPI at Exascale. In *Scientific Discovery through Advanced Computing (SciDAC 2010)*, July 2010.

[25] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[26] K.B. Wheeler, R.C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS 2008: Proc. 22nd IEEE Intl. Symposium on Parallel and Distributed Processing, MTAAP Workshop*, pages 1–8. IEEE, 2008.