© 2016 Philip B. Miller

REDUCING SYNCHRONIZATION IN DISTRIBUTED PARALLEL PROGRAMS

BY

PHILIP B. MILLER

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Dr. Anshu Dubey, Argonne National Laboratory
Professor William D. Gropp
Associate Professor Luke Olson

Abstract

Developers of scalable libraries and applications for distributed-memory parallel systems face many challenges to attaining high performance. These challenges include communication latency, critical path delay, suboptimal scheduling, load imbalance, and system noise. These challenges are often defined and measured relative to points of broad synchronization in the program's execution. Given the way in which many algorithms are defined and systems are implemented, gauging the above challenges at synchronization points is not unreasonable. In this thesis, I attempt to demonstrate that in many cases, those synchronization points are themselves the core issue behind these challenges. In some cases, *the synchronizing operations cause a program to incur the costs from these challenges.* In other cases, the presence of synchronization potentially exacerbates these problems.

Through a simple performance model, I demonstrate that making synchronization less frequent can greatly mitigate performance issues. My work and several results in the literature show that many motifs and whole applications can be successfully redesigned to operate with asymptotically less synchronization than their naïve starting points. In exploring these issues, I have identified recurrent patterns across many applications and multiple environments that can guide future efforts more directly toward synchronization-avoiding designs. Thus, I attempt to offer developers the beginnings of a high-level play-book to follow rather than having to rediscover application-specific instances of the patterns.

In memory of Daniel Harry Schreiber.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Modeling the Performance Impact of Desynchronizing Execution 5

Chapter 3 Patterns for Reducing Synchronization in Distributed-Memory Parallel Programs 12

Chapter 4 Atmospheric Data Input in ISAM 30

Chapter 5 Desynchronizing Parallel File Output 38

Chapter 6 Dense LU Factorization 51

Chapter 7 Tree-Structured Adaptive Mesh Refinement 73

Chapter 8 Desynchronizing and Optimizing the Chombo AMR Framework 95

Chapter 9 Conclusion 129

References 131

Chapter 1

Introduction

Developers of scalable libraries and applications for distributed-memory parallel systems face many challenges to attaining high performance. These challenges include communication latency, critical path delay, suboptimal scheduling, load imbalance, and system noise. A wide range of tools and techniques have been developed to analyze and address these concerns.

Among the various responses, these problems are often defined and measured relative to points of broad synchronization in the program's execution. As many algorithms are defined and systems are implemented, this is not an unreasonable approach. In this thesis, I attempt to demonstrate that in many cases, those synchronization points are themselves the flip side of the coin behind these challenges. In some cases, the synchronizing operations themselves cause a program to incur the costs from these challenges. In other cases, the presence of synchronization potentially exacerbates these problems.

1.1 Load Imbalance

A great deal of work in the parallel computing literature references load imbalance to synchronization [1]. For an explicit instance, we find the definition "In the most general sense, a load imbalance in a parallel code is the difference in work on two or more processes between two of their synchronization points" [2]. In contrast, other areas of distributed computing use measures of load, such as time-averaged processor utilization or service response time, that do not reference coordinated activities between separate processes. This distinction is quite natural, since parallel computing of the kind under consideration is often applied to problems and solution methods that involve tightly coupled interactions among various parts of the computation. Mea-

sures of utilization and responsiveness thus do not capture the effects of load imbalance in this setting[1].

In two limit cases, the time-averaged view and the synchronized view inherently coincide. In an embarrassingly parallel application, all processors work independently, but the job is incomplete until the last processor produces its final results; there is effectively synchronization at the very beginning and end of execution, but at no other points in between. The other case arises when examining load at the finest resolution between consecutive synchronization points with nothing intervening.

Over multiple points of global synchronization, aggregate utilization will equal a load measure from the bounding synchronization points if relative load among the processes is equal between each pair of consecutive synchronization points - i.e. if the load pattern repeats itself. If the load pattern varies, utilization less than 100% can reflect load imbalance even if the total work in each process is equal when summed between the two end points. Absent the intermediate synchronization, the processes would perform their equal work in equal time. Thus, we can say that the synchronization itself caused the program to incur a cost of load imbalance.

1.2 Noise

Petrini et al. related the impact of noise in an application with regular synchronization [3]. In that paper, they cited an earlier observation of noise in NAMD as a curiosity that did not impact performance as substantially [4], because at the time NAMD didn't have the global FFTs in PME for long-range force calculation that would impose global synchronization. Once that was added, noise became a much greater concern for NAMD, and usage has shifted toward leaving a processor core idle on most systems where interference cannot be avoided.

More recent work on modeling and simulating the impact of noise is specifically geared toward capturing the implementation details of heavily-synchronizing collective operations, because that is where noise is felt most severely [5].

[1]At a suitable level of measurement, e.g. neglecting polling/spinning on communication, utilization level does capture many other performance effects.

1.3 Fault Recovery

Another problem facing large-scale parallel systems is efficiently addressing system faults. One way of tolerating faults is through message-logging protocols [6, 7]. These protocols replace expensive rollback of the entire system to a global checkpoint with local recovery and re-execution of just a single failure domain, such as a hardware node or group of nodes [8]. With sufficient information about applications' control flow and communication structure, they can run with low additional memory demands and minimal execution overhead [9, 10]. More sophisticated protocols can even omit re-execution of some tasks if the underlying environment can prove that their results have been fully communicated [11].

During recovery, unaffected processes can continue execution independent of the failed process(es) up to a point where they depend (possibly transitively) on re-executed work from the recovering process. The extent to which this is possible depends on both the application's point-to-point communication structure and the frequency with which it calls for global synchronizing operations. When live processes reach a point in their execution where they wait for the completion of such operations, they must idle until the failed process(es) catch up and fulfill those dependencies. Thus, looser synchronization can reduce the impact of faults.

1.4 Summary

The preceding observations have been previously offered to motivate the use of asynchronous and task-based programming and execution models. In as much as these models avoid creating inadvertent program-order dependencies among logically independent computations, they reduce instances of synchronization that are irrelevant to correct execution. This thesis focuses at a higher level, examining the synchronization presented by algorithms and applications themselves.

My work and several results in the literature show that many motifs and whole applications can be successfully redesigned to operate with asymptotically less synchronization than their naïve starting points. The adaptations I've contributed to include

- per-step input of atmospheric forcing data in the Integrated Science

Assessment Model (ISAM), used to study land-surface/atmosphere interactions (Chapter 4),

- parallel output of particle trajectories in NAMD and ChaNGa (Chapter 5),
- dense LU factorization (Chapter 6),
- unknown-count uses of the TRAM framework for streaming many-to-many communication, used in EpiSimdemics ([12, 13, 14], described in Chapter 3),
- a tree-structured AMR mini-application (Chapter 7), and
- the Chombo framework for patch-structured AMR (Chapter 8).

We see varying degrees of performance improvement from implementing these changes. We also see that in systematically reducing synchronization, a specifically asynchronous environment is neither necessary (Chapter 4) nor sufficient (Chapter 8).

In exploring these issues, I have identified recurrent patterns across many applications and multiple environments that can guide future efforts more directly toward synchronization-avoiding designs. These are described in Chapter 3. Thus, I attempt to offer developers the beginnings of a high-level play-book to follow rather than having to rediscover application-specific instances of the patterns.

Chapter 2

Modeling the Performance Impact of Desynchronizing Execution

Motivated by the connection of synchronization to the costs of load imbalance, noise, and critical path delays, this chapter explores a simple performance model to demonstrate that less frequent synchronization can mitigate these problems.

Consider an iterative parallel program running on p processors. In each step, every processor typically executes a basic work unit of length w. Some fraction f of the work units, evenly distributed across time and processors, take longer than w by a ratio $r = 1 + \Delta > 1$. The processors must synchronize their execution every k steps (e.g. in a convergence test, or the data distribution seen in ISAM in Chapter 4). Unlike Valiant's Bulk Synchronous Parallel model [15], the synchronizing operation itself is taken to be free, and communication cost is neglected. The application will run for s steps in total, which is assumed to be large. An intuitive illustration of this model, with varying k, can be seen in Figure 2.1.

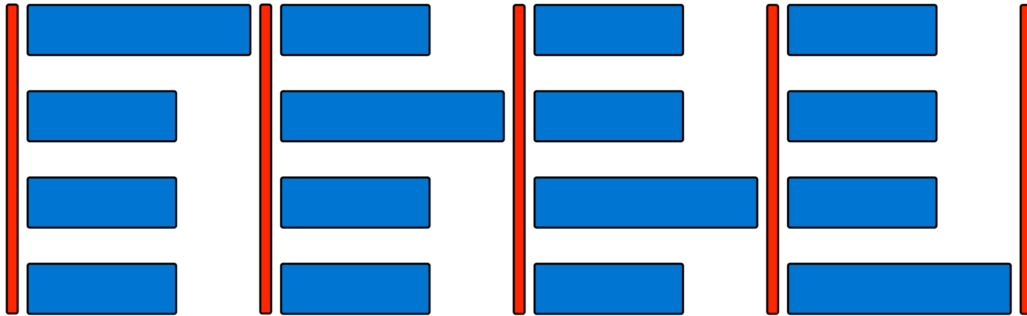
In all cases, the total work is given by the expression

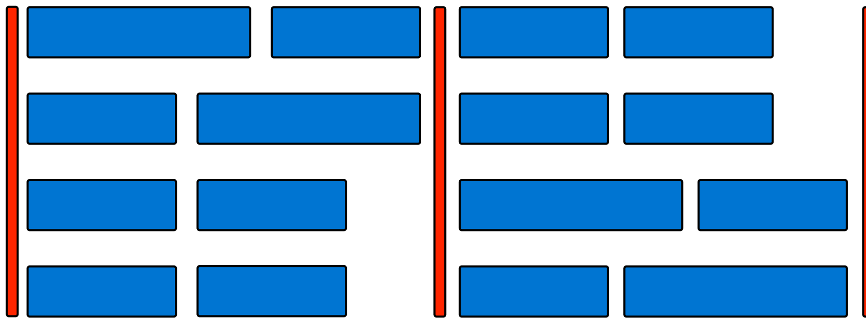
$$spw(1 + f\Delta)$$

Given the uniform distribution of overload, the ideal execution time simply divides the work by the processor count:

$$T_{ideal} = sw(1 - f + fr) = sw(1 + f\Delta) \quad (2.1)$$

Since we are primarily concerned with large-scale parallel systems, assume that $p \gg 1/f$, so that some processor can be expected to experience an overload in each step.





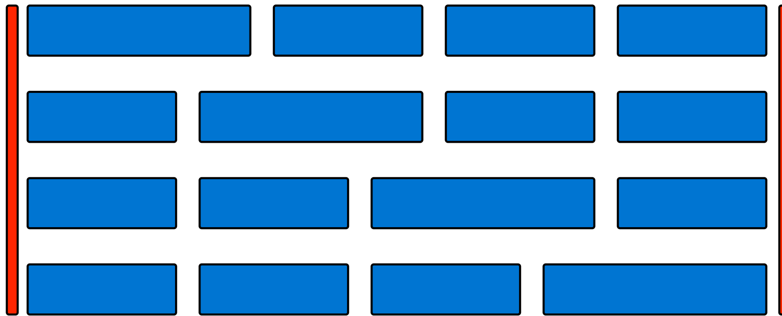


Figure 2.1: An illustration of the intution behind this model's predictions that eliminating synchronization can mitigate performance impacts of load imbalance, noise, and other undesirable effects. The rows of blue boxes represent work on separate processors, and red bars represent points of global synchronization among them. As synchronization is eliminated, less time is lost.

In the case where the application synchronizes at every step, $k = 1$, the elapsed time per step is simply the maximum over all processors, wr, and so the total elapsed time is

$$T_{k=1} = swr = sw(1 + \Delta) \tag{2.2}$$

Effectively, the program behaves as if $f = 1$ in Equation 2.1.

Next, we consider the case where synchronization is slightly less frequent: $k = 2$. In very unlucky windows, some processor will have two long steps back to back, making the window take $2wr$ time[1]. Except at processor counts that are large relative to f^2, the majority of windows can be expected to take time $w(1+r)$ and thus the total elapsed time (with 2 steps in each window) is approximately

$$\begin{align} T_{k=2} &= s\frac{w(1+r)}{2} \tag{2.3} \\ &= sw\frac{1+(1+\Delta)}{2} \tag{2.4} \\ &= sw\frac{2+\Delta}{2} \tag{2.5} \\ &= sw\left(1+\frac{\Delta}{2}\right) \tag{2.6} \end{align}$$

We can see that just widening the synchronization window by one step nearly halves the effect of the overload, relative to Equation 2.2.

In the limit where $k > s$, every processor is expected to experience fs long steps of time wr and $(1 - f)s$ steps of time w. Thus the total time to completion is given by

$$\begin{align} T_{k>s} &= fswr + (1-f)sw \tag{2.7} \\ &= sw(1 - f + fr) \tag{2.8} \\ &= sw(1 + f\Delta) \tag{2.9} \end{align}$$

This matches the ideal time given by Equation 2.1. Entirely removing mid-run synchronization thus fully mitigates any excess cost of the intermittent overload.

[1]This probability may not be entirely negligible at $k = 2$, but it decreases geometrically as f^k, with larger k being of greater interest

More generally, we can compute the expected time for any value of k. The program will execute s/k synchronization windows each encompassing k steps. Of those, $\lceil kf \rceil$ will be long, and the rest normal. Thus, the total time will be approximately

$$\frac{s}{k} \cdot (\lceil kf \rceil wr + (k - \lceil kf \rceil)w) \tag{2.10}$$

In the limit as k grows large in this equation, we can again see that it converges to the ideal time given in Equation 2.1. This formula as a function of k is plotted in Figure 2.2. As a speedup relative to $k = 1$, we find that

$$\text{speedup}_{f,\Delta}(k) = \frac{1 + \Delta}{1 + \frac{\lceil kf \rceil}{k}\Delta} \tag{2.11}$$

This is illustrated in Figure 2.3.

The overall effect of fully eliminating synchronization in this model is to provide a maximum speedup of

$$\text{speedup}_{\Delta,k \geq s}(f) = \frac{1 + \Delta}{1 + f\Delta} \tag{2.12}$$

Since noise is typically observed to occur in a time-dependent manner rather than a step-dependent manner, the model requires some adaptation[2]. Consider unsynchronized noise occuring with a period t and amplitude a. If $t \leq w$, then it is high enough frequency to affect essentially all processors equally from this model's perspective. Thus, $f = 1$ and $\Delta = \frac{w}{t} \cdot a$.

In the lower-frequency case, where $t > w$, then $f \approx t/w$ and $\Delta \approx a/w$. These are approximate values, because the stretching of a step on one processor may delay others enough that they suffer noise as well, or noise may occur when the processor was idle waiting for a message to arrive [5].

The model as described above assumes that the relevant synchronization is global among all processors. When synchronization is not global, but involves only a subset, the model must be adjusted accordingly. The fraction of

[2]Trace-based performance instrumentation with fixed output buffers is an exception to this. In Charm++, event logs for the Projections tool get flushed to disk after a fixed number of entries. Since steps of execution typically generate similar numbers of events repeatedly, but in different volumes on different processors, flushing acts like step-based noise.

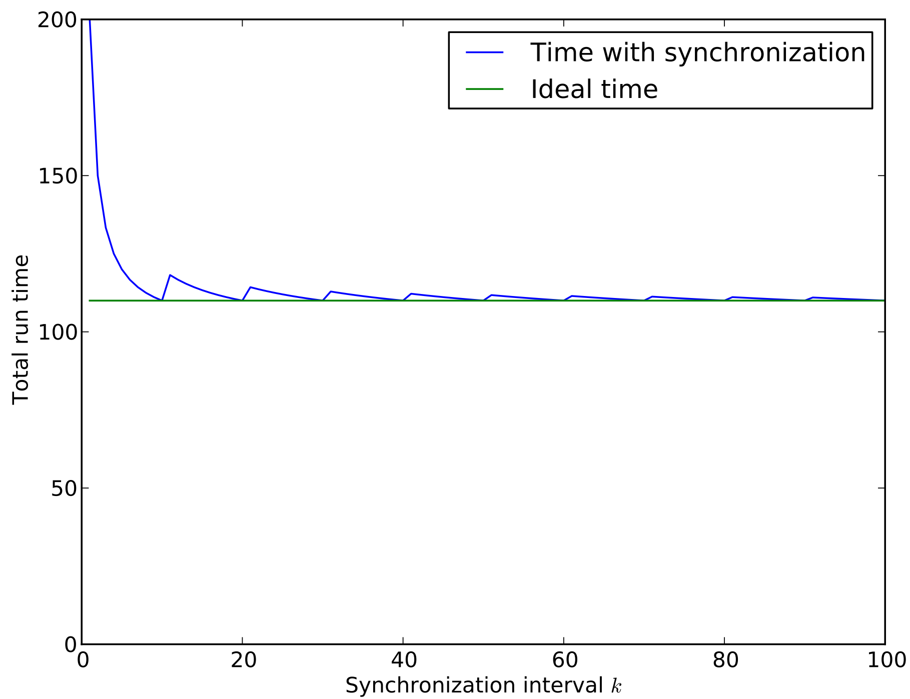


Figure 2.2: Predicted running time of a program of 100 steps, with $\Delta = 1$ and $f = 0.1$ as a function of k. We can see that as the synchronization window k increases, the running time converges to the total workload.

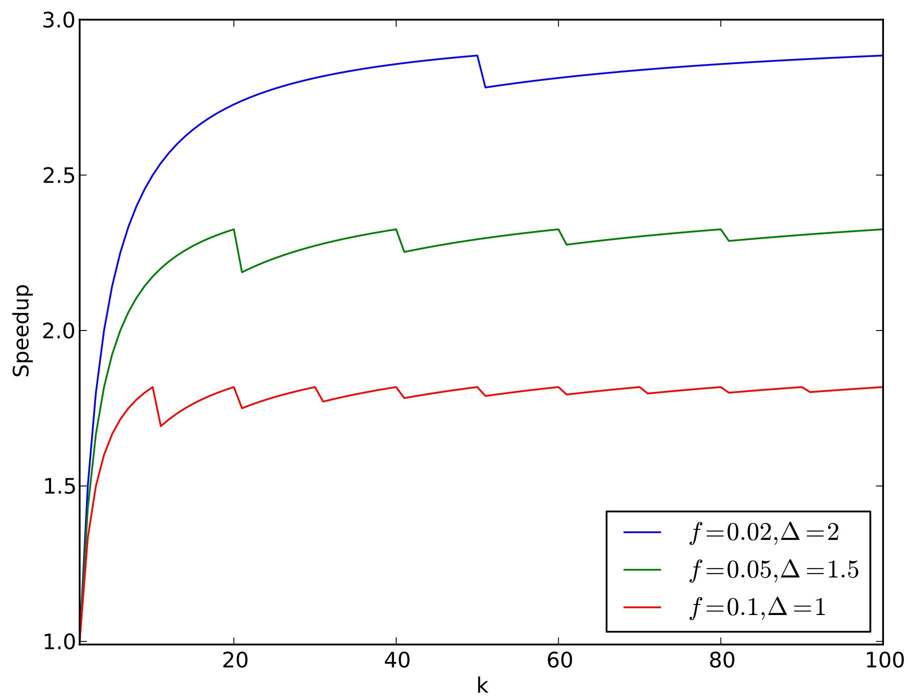


Figure 2.3: Speedup provided by widening the synchronization window from $k = 1$. Essentially ideal speedup is reached when $k \geq 1/f$.

processors that synchronize at any given point can be used to scale the value f. This reflects that in any given synchronization group, a processor will be overloaded proportionately less often as in the entire system. This structure may be found in neighbor exchange operations, especially where each processor is responsible for more than one independent unit of computation.

Programs that exhibit 'phased' behavior within larger steps should approximately follow this model. This is observed, for instance, in the original version of Chombo, with updates to each level of refinement constituting a phase. At different levels, different processors are over- or under-loaded, but across phases they partially average out. As described in Section 8.2, desynchronizing those phases improved overall performance.

Chapter 3

Patterns for Reducing Synchronization in Distributed-Memory Parallel Programs

Looking at the applications described in this thesis and elsewhere, we can identify several common patters for weakening or eliminating synchronization in distributed-memory settings:

- Batch (blocking on / waiting for) collectives (§ 3.1)
- Communicate more with each collective, to reduce frequency (§ 3.2)
- Send and consume data expected from a collective incrementally (§ 3.3)
- Separate communication from coordination (§ 3.4)
- Replace synchronizing collectives with coordination schemes (§ 3.5)
- Replace synchronizing collectives with p2p messages that achieve the desired effect (§ 3.6)
- Semantic object naming (§ 3.7)

This chapter characterizes these patterns, and attempts to generalize and abstract them so that they can potentially be applied more broadly and consciously.

This effort is in the spirit of various design pattern and pattern language efforts, originating in the work of (building) architect Christopher Alexander [16], imported into computer science and software engineering by the 'Gang of Four' Gamma, Helm, Johnson, and Vlissides [17] and broadly explored in the 'Pattern Languages of Programs' (PLoP) conference, and specifically introduced to parallel programming by Tim Mattson et al. in a book [18] and the ParaPLoP workshop series.

A common characteristic in many parallel applications is that broad synchronization is typically associated with calls to collective operations. This is partially inherent in the natural dependences those operations express,

such as in all-to-all communication patterns. It can also result from dependences introduced by the algorithms used in the implementation of those operations [5]. Thus, many of the patterns illustrated in this thesis involve modifying or replacing collective communication operations with various alternatives.

In some cases, an operation causing synchronization conveys additional semantic information relevant to the applications control flow beyond any direct payload of data that it transfers. In the limit of operations that carry no payload, such as MPI barriers and Charm++ quiescence detection, the only purpose they can serve is to transfer this more abstract state across processes and objects. In those cases, if we wish to weaken or eliminate synchronization, we must find other ways to let applications draw the same conclusions.

At the level of implementation considerations, synchronization often conveys useful information that cannot be overlooked. At the point of the call, the posting of a send operation indicates several facts:

1. the payload data has been fully computed
2. the payload data is available in a specific designated buffer
3. the sender is willing and able to commit network resources to that transfer

Analogous versions of the latter two points also apply to explicit receive operations.

1. a specific designated buffer is available to receive the payload data
2. the receiver is willing and able to commit network resources to the transfer

Since collectives generally combine sending and receiving data, both sets of facts apply, among all participants in the collective. These facts are essential to the efficient execution of optimized collective algorithms that rely on cooperation of processes between the originator of some input data and its final recipient.

In one-sided operations, these necessary facts must be handled very differently, essentially with coordination schemes of various sorts. Readiness

of data must be conveyed through synchronization separate from the communication operation, such as fences or barriers. Buffer addresses must be negotiated in advance or explicitly communicated. Network resource commitment, where taken into consideration, also requires explicit synchronization aside from the transfer.

Consider an MPI application that wants to ensure that all of a set of receives are posted before their corresponding sends, or vice versa, to ensure that it does not fail or suffer performance penalties due to implementation or hardware limits. This is a fundamental example of synchronization for coordination without any payload or higher-level semantic content. It's solved by posting a set of non-blocking operations all in one direction (e.g. all sends), calling for a barrier, and then posting all of the opposite operations (e.g. all receives).

In general, we informally consider synchronization as a third factor alongside communication and computation when analyzing the design and deployment of various parallel algorithms. An algorithm or configuration that can achieve its desired result by reducing one of these terms either without increasing the others or by limiting the increase to a reasonable tradeoff can be expected to offer better overall performance. Similarly, with increasing processor count or problem size, slower growth in the synchronization term will predict lower overall cost. Thus, although applicable in different contexts and applied in different forms, the patterns below all aim at reducing synchronization relative to other elements of a parallel program's execution.

3.1 Batch (Blocking On / Waiting For) Collectives

AKA Use synchronization from one collective to avoid incurring synchronization penalties from subsequent collectives.

Applicability There is a large interval between when data to be communicated is or can be ready to send, and when it must be available to consume. There is a large portion of available data whose communication can be shifted substantially earlier or later to be communicated in bulk. The synchronization associated with the unit collectives does not provide useful information to the rest of the application.

Description Consider an application that performs several collective operations interspersed with its computation in each step. For instance, the application may execute in several delineated phases, with some of those phases calling collectives. If the result of those collectives isn't immediately needed to continue work within that phase, nor does the fact of their completion provide semantic information, then the synchronization those collectives impose may incur avoidable costs as described in Chapter 2.

To avoid those costs, it's preferable to rearrange execution (and potentially program structure, as necessary) so that as many collective operations as possible are performed back-to-back, rather than interspersed with computation. After one synchronizing operation, batching more in a 'convoy' avoids introducing opportunities for imbalanced load, noise, or extra critical path delays to intervene and delay completion of later operations. Even with nominally uniform work for every process to perform between synchronizing operations, increasingly variable processor clock speeds across individual chips and nodes means that any extraneous work will likely be imbalanced [19].

With non-blocking collectives, this pattern is still potentially relevant. Barring specialized hardware, interrupt-driven operation, or a dedicated thread, processor time must still be made available to allow non-blocking collectives to make progress. Batching uses the synchronization of one operation to guarantee the necessary resource availability for however many follow it in (ideally) quick succession.

Known Uses

CharmLU: The exclusive scheduling of active panel factorization updates and reductions ensured that other operations would not interfere with this highly-synchronous work on the critical path. Essentially, from the perspective of the active panel, the large-block DGEMMs in trailing updates acted as noise lasting several times the duration of each column's work. The optimization to batch up the synchronous active panel work is described in Sections 6.2.5 and 6.2.6.

Chombo: Though not its logical endpoint, the work on Chombo's stable timestep calculations described in Section 8.2 shows the effect of batching reductions. By doing all of a timestep's reductions at its end, and then combining them, we obtained better performance than running separate reductions between timestepping of each level of refinement.

Related Concepts This pattern will tend to appear in concert with explicitly increased concurrency and parallelism in nearby portions of execution. In Chombo, application of this pattern exposed latent concurrency among computation on multiple levels of refinement.

3.2 Communicate More With Each Collective, To Reduce Frequency

AKA Condensing collectives

Applicability The same general conditions as in batched collectives apply. Additionally, the communication and computation to be performed must fit in a single defined operation.

Description Abstractly, the idea of this pattern is to rearrange communication, and possibly the computation to feed that communication, so that more communication occurs in association with each synchronizing operation. If that synchronization was not otherwise informative, then this rearrangement should lead to a proportionate decrease in how often synchronizing operations must occur.

One instance of this pattern would be replacing repeated gather or scatter operations over a common set of processes but with varying roots with fewer all-to-all or all-gather operations.

This pattern is something of an anomaly. Where other patterns suggest to replace or weaken collectives, this one proposes to strengthen them. The tradeoff is that a single heavy-weight operation may carry a great deal of synchronization (in the form of excess dependencies), but much of that synchronization can be overlapped within a single narrow window of execution, leaving a many times broader window of execution between synchronization.

Known Uses
A clear example of this pattern is seen in optimizations of the ISAM land surface modeling code, described in Chapter 4. In that setting, repeated scatter operations occurring every k steps were transformed to much rarer all-to-all operations every kp steps, whose period scales with the number

of processors used in a given run. The total set of true dependences and which processor provided the data to satisfy them was unchanged by this transformation. However, all computational steps beyond the first one after the collective have their dependences satisfied much earlier.

Meanwhile, many false dependences between independent computational steps were broken apart. In this case, the frequent synchronization from scatter calls provided no extra information that the application itself actually needed. However, the coordination of network resource usage those synchronized collectives provided avoided creating network hotspots that would have appeared in a naïve use of one-sided operations instead. Using an all-to-all operation preserved the communication efficiency benefits of synchronized network resource allocation.

Ultimately, by applying the transformation from frequent iterated scatter operations to rare all-to-all operations, the synchronization window was drastically widened. As predicted by the model described earlier and observed in experiments, this had the effect of fully mitigating periodic load imbalance.

In the Chombo AMR framework, the time-advancing operation on each level is expected to globally reduce and return a stable timestep value usable for the following step. As described in Section 8.2, all of the per-level stable timestep reductions ultimately fed into a single reduction of the value to use across all levels of refinement. By forcing synchronization between the updates of separate levels, the framework incurred the costs of load imbalance and communication latency at every level. Moving all of these reductions to the end of each whole step eliminated that excess synchronization, and allowed all the reductions to be consolidated into a single operation.

Charm++ has recently implemented a consolidated 'summary statistics' reduction operation. It takes multiple inputs, and computes all of the minimum, maximum, mean, and standard deviation. It is built on a more general tuple reduction design that would allow condensation of arbitrary related reductions, though that has not yet found other uses.

As described in Section 6.3, the 'tournament pivoting' algorithm used in Communication-Avoiding LU factorization also demonstrates this pattern.

Related Concepts As in the previous pattern, condensed collectives will tend to appear in concert with explicitly increased concurrency and paral-

lelism in nearby portions of execution. Cause and effect in this relationship can flow in either direction, depending on details of the application. In ISAM, parallelization of the atmospheric input reading and pre-processing was necessary to subsequently make this pattern applicable.

Alternatives Non-blocking collectives are applicable in circumstances very similar to this pattern - a substantial time interval between availability of some data and the demand for it. When applied exactingly, they have the advantage of more precisely expressing the dependencies of related computation on the data they convey. They have the prospective disadvantage of increasing the necessary amount of control flow and volume and lifetime of state necessary to manage them. This contrast is particularly noticeable when the combined data grows with one of the application's scaling factors, such as processor or step count. This is the case in ISAM, which would call for up to $\mathcal{O}(P)$ non-blocking scatters to be in flight at once. Non-blocking collectives also require an effort to ensure their progress, which may not be trivial if the application otherwise does little or no communication.

3.3 Avoid Collective Synchronization by Sending and Consuming Data Incrementally

Applicability The data to be communicated by a collective can be broken down into independent units, and the computation can incrementally generate and/or consume those units in coarse enough work grains to not suffer high relative overhead from fine-grained communication.

Description As noted by Hoefler et al. [5], various collective operations encode different dependency patterns among participating processes. By using a collective to communicate the data to fulfill those dependencies, an application is saying that *all* data to be sent depends on all preceding computation, and all subsequent computation depends on all data received. In some cases, such as transpose-based parallel FFT implementations [20], this is a very practical choice.

In many cases, however, each unit of data to send might be generated by an independent unit of computation, and each unit of data received may expose

independent units of computation. This pattern suggests that the program should be structured to allow those computations to occur incrementally, with data tansfer interleaved. Thus, the program can avoid the strong synchronization entailed by using a collective to transfer it all at once.

Known Uses

CkIO, an implementation of parallel output in Charm++, demonstrates this pattern [21]. As data fills a buffer representing a write stripe, the write operation can be initiated immediately, independent of any other data that may be expected. This is described further in Chapter 5.

CharmLU also exhibits a variant of this pattern, as described starting in Section 6.2.1. Due to its memory scheduling, blocks of matrix data are not transferred in conventional multicast operations to be consumed at some point later. Rather, they're transferred when there is space set aside for them. Computation proceeds according to priority, and releases storage holding input blocks as the need for those blocks on a given process ends.

EpiSimdemics: Another Charm++ application demonstrating this pattern is EpiSimdemics [14]. This is an agent-based code for simulation of contagion phenomena in semi-discrete time. Typical simulations focus on the spread of infectious diseases in day-long steps. The simulation is modeled by a bipartite graph of objects. The objects on one side of the graph represents groups of individual agents. The objects on the other side of the graph represent groups of locations where those agents may interact over the course of each simulated day. In each day, the agents dynamically generate a schedule of locations they will visit, and transmit the times they will be at each of those location to the location objects. When the location objects have received the full set of visits, they compute the interactions between agents who were present at overlapping times during that day, and send the interactions back to the agent objects. When the agents have received the full set of interactions they experienced during the day, they update their state (e.g. become infected) and prepare for the next simulated day.

The initial implementation of EpiSimdemics in MPI executed each day in a cycle of compute-communicate phases. Each communication was implemented as a variable all-to-all collective. For various reasons, this design encountered scaling limits at approximately 512 cores. Allowing for incre-

mental communication and processing in the CHARM++ port (along with other changes) improved scalability to thousands of processors. Ultimately, subsequent optimizations enabled it to scale to hundreds of thousands of cores on Blue Waters and various Blue Gene Q installations.

CHOMBO: A mild example of this pattern can be seen in the MPI implementation of CHOMBO. Its standard communication pattern for block-to-block boundary exchange, interpolation, and down-sampling uses a loop of point-to-point nonblocking send and receive operations between processors holding neighboring or overlapping boxes, followed by waiting for all of those operations to complete. This behavior is precisely modeled by the neighborhood all-to-all operations added in the MPI-3 standard. However, by treating each message independently, receivers are able to do necessary processing on each one (e.g. copying from the communication buffer into its final location; accumulating the received values with existing data; interpolating the received data in time or space either alone or with existing data) as it arrives, rather than waiting for all of the messages expected by the collective that could nominally replace these individual operations.

A more fully elaborated form of this pattern appears in porting CHOMBO to CHARM++, described in Chapter 8. Each high-level communication operation in the original CHOMBO code was treated as a collective. In this design, computation on any box on a processor is made to wait on completing the communication for all other boxes also assigned to that processor. By running the control flow of CHOMBO for each box independently, we gain incremental processing of communicated data. As soon as each box's dependencies are satisfied, its computation proceeds.

An implementation of parallel sorting in CHARM++ by Solomonik [22] exhibits this pattern. Earlier implementations identify 'splitter' or 'separator' keys that would produce a sufficiently uniform distribution of values across processors, redistribute all of the data according to those distinguished keys in a bulk all-to-all operation, and only then rearrange the data locally on each process to its final sorted order. In the newer implementation, data is partitioned and transferred incrementally as acceptable splitter keys are identified, and each recipient starts merging partial results into locally sorted order as the rest of the data moves and the sorting process proceeds.

Related Concepts Streaming communication and computation is a more general idea that somewhat subsumes this one. The distinction is that incrementalizing a collective is based on having a pattern of communication that has inherent uniformity in what data is being communicated, and what entities are sending and receiving it. Streaming deals with data that would not have generally been transferred through collectives, except in artificial attempts to follow a bulk-synchronous design.

After all the individual units of data have been received and processed, there may be subsequent work dependent on the whole set. If the count of messages to be received is known *a priori*, then that computation can be locally triggered. If not, the next pattern is applicable as well.

3.4 Separate Communication from Coordination

Applicability The application gains useful semantic information from the communication operation beyond the content explicitly passed by the operation. That semantic information is not inherently necessary for efficient implementation of the communication itself.

More concretely, this patterns applies when there is uncertainty in the number or size of messages being sent and received. One example would be a scalar `MPI_Alltoall` to form the receive counts and displacements arguments to a subsequent variable `MPI_Alltoallv`. Another would be a reduce-scatter to indicate how many messages each process must wait to receive in a communication phase before moving on.

A more general scenario with uncertainty of both sends and receives is where receiving one message may lead to the generation of others. Thus, the program requires some consensus mechanism to ensure that each process can tell when it has received every message that has been or will be sent to it.

Description Send messages using elementary point-to-point mechanisms, or using optimized schemes such as routing and aggregation [13]. These can be fairly general, and non-specific to the application. Separately, identify the characteristics of the communication pattern or the algorithm it supports that need to be signalled. Implement or deploy a mechanism to detect that characteristic. If a more general mechanism doesn't entail an excess cost in efficiency, it is likely suitable.

Implementation Mechanisms

Nonblocking Barrier or Reduction: This simplest mechanism can be lightweight and very efficient, since it requires only a simple, bounded communication pattern to implement. However, it is limited in that it requires message senders to be identified before communication starts, and message senders to know idependently when they have sent all of their traffic. An implementation of this pattern, Dynamic Sparse Data Exchange [23], was offered as motivation to add non-blocking barrier collectives to the MPI standard.

Quiescence Detection (QD) is one of the most general mechanisms to fill this role. It detects when all messages sent within a parallel program have been received and processed, and no further messages remain in transit in the network or queued on any processor. Thus, it can support use cases with unpredictable senders, variable message counts, and dependent messages generated as a result of receiving previous messages. Charm++ includes a highly efficient implementation [24]. A potential downside of its use, noted below, is that by detecting a global property, it conflicts with concurrent composition of multiple modules the idependently need to know when their own communication has finished, without regard to activity in the others.

Modular Quiescence Detection: 'Completion Detector' (CD) One means of providing this out-of-band coordination is the 'Completion Detector' library in Charm++ [25]. This library uses waves of reductions and broadcasts over a spanning tree of the system to count senders finished and messages sent and received, signaling completion when all messages from all senders have been delivered. It is similar in design to Charm++'s quiescence detection module, except that it has been made modular to not require that the rest of the program fall idle for it to make progress.

Parental Responsibility 'Termination Detection' (TD) is a design for recognizing the completion of 'diffusing computations' on a distributed system, originally developed by Djikstra [26]. It builds a spanning tree over processing elements that have sent messages, with responsibility for detecting the end of all activity within a subtree lying with its root. As processors receive messages, they send acknowledgments to the sender, except for the first message they receive which is not acknowledged until all messages they have sent get back acknowledgments.

Known Uses

EpiSimdemics ([14], also described in § 3.3) illustrates this pattern. Specifically, the number of messages each communication end-point will send or receive in a step is not known *a priori*, since it is entirely data dependent. As noted above, setup work can occur as messages are delivered, but each timestep cannot be processed by an object until it knows that it has received all of the messages sent to it for that step. In this case, each sender knows when it has sent all of its messages for a step. Thus, senders can potentially tell a coordination mechanism when they are done, and how many messages they sent.

As the communication pattern in EpiSimdemics is an instance of dynamic sparse data exchange [23], that design would have been applicable to the MPI implementation. The same design is possible in Charm++, but with much greater syntactic and runtime overhead.

Quiescence detection worked well for this purpose in early versions of EpiSimdemics. However, its global nature was problematic when the application evolved to run multiple instances of the simulation within each job to allow dynamic scenario testing. At that point, QD coupled progress of the independent scenarios, and hence synchronized them. Substituting an instance of the Completion Detector for QD in each instance of the simulation resolved this difficulty.

Topological Routing and Aggregation of Messages (TRAM [13]): While the basic mechanisms of TRAM can convey the data in question from sender to receiver, both TRAM and the application need additional coordination information for correctness. TRAM needs to know when all messages generated by a step have been sent, so that it can start to flush its buffers. TRAM must also know when all messages at a given layer of its routing system have been received, so that that layer can proceed to flush its buffers to the next layer. Finally, the application needs to know when it has received all of the messages that were sent, so that it can proceed to compute on the entire set.

TreeAMR: In the TreeAMR code described in Chapter 7, we initially replaced global collective communication with point-to-point messages carrying all of the data, and a separate synchronization mechanisms to indicate when consensus on how to adjust the mesh was reached. This stage of evolution does not eliminate synchronization, but it weakens the dependencies

carried by it, and enabled further adaptations that do eliminate broad synchronization. A second stage of evolution in TreeAMR carried this further: by carefully coordinating object insertions and subsequent control flow, one of two synchronizations at each regridding step was further eliminated.

DSDE: Another example can be seen in recent solutions to the 'dynamic sparse data exchange' problem [23] that helped justify the non-blocking barrier operation added to MPI-3. A naive implementation of this communication pattern uses a variable all-to-all operation (possibly preceded by a simple all-to-all to convey buffer sizes), with many send/receive counts set to zero. To replace the sparse all-to-all operations with lighter weight communication patterns, Hoefler et al. combined non-blocking synchronous sends with a non-blocking barrier used to indicate that all messages had been delivered, and so no process would need to wait on the chance that more would arrive.

Related Concepts Valiant's Bulk Synchronous Parallel model [15] assumes that all communication is uncoordinated, and uses separate global synchronization for receivers to know that messages have arrived. This pattern partially mimics BSP, with major caveats. Relative to BSP, the presumed ability of modern systems to process messages as they arrive offers substantial 'extra' expressive power within each superstep, since each received message can potentially generate subsequent dependent messages. Thus, more powerful mechanisms than barriers are necessary, but not necessarily any more costly.

Standalone communication without conveying broader coordination is likely to also be received incrementally, and hence available to process incrementally as well. Thus, this pattern is closely related to the pattern of replacing collectives with incremental communication and processing, described in Section 3.3.

3.5 Replace Synchronizing Collectives with Coordination Schemes

Applicability The application gains useful semantic information from the synchronizing operation beyond the content explicitly passed by the operation. That semantic information is not necessary for efficient implementation of the communication itself.

Description The semantic information provided by synchronization can be obtained by some other application-specific means, or the application can profitably be adapted to use a substitute for or alternative form of that information.

Known Uses
CharmLU: HPL, the reference LINPACK implementation used for Top500 benchmarking [27], uses synchronized multicasts over rows and columns of a 'process grid' to transmit factored blocks for use in further computation. The synchronization in these broadcasts effectively coordinates two kinds of resources: communication bandwidth and progress, and available memory. CharmLU (described in Chapter 6) does not use synchronized multicasts, and thus coordinates these resources through other means [28, 29, 30].

HPL has a configurable fixed 'lookahead depth' that controls how many steps ahead of computation data can be distributed. By doing so, it limits the memory footprint of remote data that each process holds as input for local computations. CharmLU replaces the fixed lookahead depth with a dynamic per-process value based on actual available memory. Within the bounds of available memory, each processor sends explicit requests for blocks of data that it will need in the near future. How it does this, efficiently and without creating deadlock situations, is described in Section 6.2.2.

Because a given factored block is needed by all blocks beyond it in the factorization, the request scheme created hotspots of injection bandwidth on the nodes holding each block as many requests arrived in close succession. This revealed the value of coordinating network resource usage from the synchronized multicasts. Since the set of processors with a pending request at any given time is dynamic, we implemented a scheme to send the message via dynamic spanning trees by appending the destination list to the block

message to a few processors, and have them each forward the message to further subsets. This is described in more detail in Section 6.2.4.

CkIO also demonstrates this pattern. Collective output has been used to synchronize processes to ensure that the recipient of data to be written would be ready to handle it when provided. With message driven asynchronous execution, concurrent composition, and suitable coordination, the synchronization becomes unnecessary. The results of this work are described in Chapter 5.

Related Concepts Alpert and Philbin described an analog of this concept for a Bulk Synchronous Parallel setting, which they call 'Counting BSP' [31]. They observe that if receivers can anticipate the number of messages they will receive in a superstep and count them as they arrive, then no actual barrier to synchronize the end of the superstep is necessary.

3.6 Replace Synchronizing Collectives with Point-to-Point Messages That Achieve the Desired Effect

Applicability The application gains useful semantic information from the synchronizing operation beyond the content explicitly passed by the operation. That semantic information is not necessary for efficient implementation of the communication itself. Enough data is available locally in each process to form a substitute through targeted point-to-point messages.

Description When the coordination information drawn from synchronization is not necessarily global, individual processors can communicate directly with each other to synthesize a local substitute. This may imply changes in the higher-level algorithm's operation to use that local information rather than the global version previously available.

Known Uses
CharmLU's use of directed request messages to indicate memory availability for specific blocks on particular processors demonstrates this pattern, as described in Section 3.5 and Section 6.2.2.

In concert with 'separating communication from coordination' (Section 3.4), broad synchronization in the mesh evolution phase of TreeAMR ([32], Chapter 7) can be eliminated entirely. Both the first and last stages stages of optimizations to mesh evolution push the knowledge of general state from broadly interdependent collective operations about regions to be refined or coarsened, objects to be created or destroyed, and neighbors added or removed into local point-to-point messages with narrow dependencies. The first stage replaces collectives that described mesh evolution globally with locally consistent and stable mesh evolution marked by consensus detection. The last stage replaces globally detected and signaled consensus with locally provable consensus based on convergence of lower and upper bounds.

This pattern is potentially applicable to numerically stable time step calculation in any simulation of a system in which computed effects have finite propagation speed through the simulation domain. For instance, compressible fluid flows satisfy this, while incompressible models do not. They can achieve this by locally computing and applying stable time step lengths within each portion of the domain, and coordinating values at interfaces where the time step differs. The impact this approach would have over the prevalent global reduction method in Chombo is explored in Section 8.2.3. Implementation of this new design is on the roadmap for the Enzo-P AMR cosmology/astrophysics application using the Cello framework [33].

3.7 Semantic Object Naming

AKA Unique global names, coordinate-based indexing

Applicability The application requires a *dynamically evolving* set of objects that must be generally reachable for communication. These objects are counted, numbered, sorted, placed, or located. The application is built to run in an environment that provides fully distributed and asynchronous location metadata lookup (e.g. Chare Arrays in Charm++ [34]).

Description Every object to be created has its identifier constructed based on some inherent characteristic of its place or role in the application. How that identifier is formed may be very application-dependent. For instance:

- a spatial coordinate within the simulation domain,
- the path to or position of the object in some distributed structure representing the computation
- the object's unit of responsibility in a data decomposition
- how much progress the computation has made

This obviates the need for globally synchronizing steps to do things like counting the objects to create, scanning that count across processors, numbering the objects, sorting the object list, or mapping object-level neighbor lists to processor-level communication lists.

Known Uses

There are two implementations of tree-based structured AMR in Charm++ that apply this pattern. Both implement bit-vector indices for boxes giving the path from the tree root to the named node, as described in Section 7.2.4 in the context of a tree AMR mini-application. The Cello framework extends the scheme to a forest-of-trees structure, by prepending the top-level index of each box's containing tree to the bit-vector. A similar design is on the roadmap for ChaNGa's Barnes-Hut tree.

In CharmLU, matrix blocks are indexed by their position in the matrix. This enabled easy experimentation on their mapping [35], and would support the request-driven dynamic multicasts described in Section 6.2.4 even if blocks were dynamically relocated (such as for load balance) during execution.

The port of Chombo to Charm++ partially implements this pattern. It encodes the generation and refinement level of each box into parts of their index, as described in Section 8.1.3. The final element of each box's index that's currently generated by global numbering could be replaced with coordinate-based indices to avoid global numbering and sorting.

Bock and Challacombe's implementation of *Sparse Approximate Matrix-Matrix Multiplication*, SpAMM [36], logically extends the 3D parallel matrix-multiplication of Agarwal et al. [37]. They use the sparsity structure to omit computing portions of the block-wise convolution space that will not make

noticeable contributions to the final product. The basic 3D algorithm could be implemented to apply this pattern both in 2D to blocks of the input and output matrices, and in 3D to the intermediate blocks summed to form the product. SpAMM implements a recursive tree structure within these indices, representing subdivisions of the matrix that it will actually operate on.

Alternative: In applications with well-defined and smoothly-evolving neighborhood structure, process-local object numbering and distributed update protocols can similarly avoid global operations to name and connect objects (at least after initialization). AMR simulations can be made to fit this case, and can be implemented as such [38]. The challenge of this approach as opposed to pure semantic naming and distributed location lookup is that it requires more specialized application logic to implement. In both cases, each application must invent its own means to label relevant and potentially evolving entities. Without distributed lookup for unique global names, applications must also define its own algorithms for locating and connecting those objects, rather than relying on a common synchronization-avoiding infrastructure. Since a universal implementation of that infrastructure exists [34], it can be effectively used with this pattern in all but extraordinary cases.

Chapter 4

Atmospheric Data Input in ISAM

List of Patterns Illustrated:

3.1 Batch (blocking on / waiting for) collectives (§ 4.1)

3.2 Communicate more with each collective, to reduce frequency (§ 4.3)

The Integrated Science Assessment Model (ISAM) is a whole-Earth and single-site land surface model code used to study biogeophysical and biogeochemical phenomena and their interaction with the larger climate and ecosystem. It couples fluxes of carbon, nitrogen, water, and energy between the near-surface atmosphere, plant growth, and material accumulation above and below ground. ISAM has been used extensively in various modeling and synthesis studies.

In previously published work in collaboration with Michael Robson [39][1], we adapted ISAM to substantially improve its performance and scalability on large supercomputers. Relative to the performance of a suitable baseline version of the code at its peak of $1k$ cores, we obtained a 6.6× speedup on the Hopper Cray XE6 system and a 2.8× speedup on the Edison Cray XC30 system, both hosted at NERSC. These same changes also enabled strong scaling up to $32k$ cores on each of these systems, with efficiency limited only by a documented serial bottleneck described by Amdahl's Law. This chapter reviews the portion of the previously published work relating to the model's atmospheric data input (also called the 'climate forcing'), describes the substantial excess synchronization that our improvements removed, and reanalyzes the results obtained in light of the model presented in Chapter 2.

[1]The text and figures of this chapter are adapted from the cited paper with permission. ©2014 IEEE

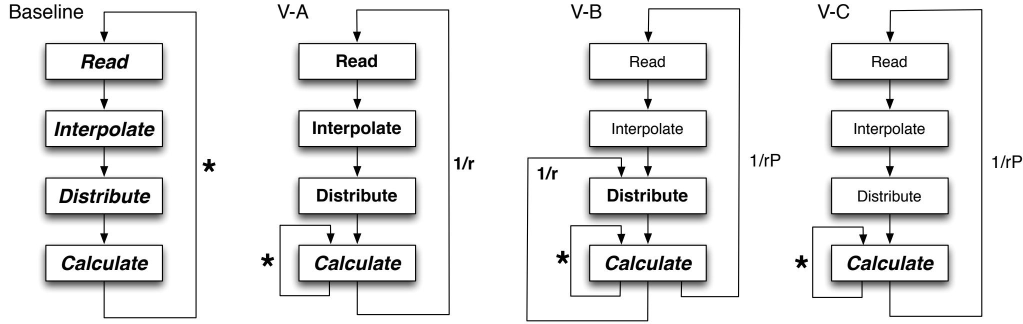


Figure 4.1: An illustration of the process by which various versions of ISAM acquire and consume atmospheric input data. Each flow chart represents one version of the code, with various optimizations implemented. The optimizations are described in the section corresponding to the label on each flow chart. In the baseline version, input is read, interpolated, and distributed in each step (shown by '*' loops). Later versions reduce redundant work by a factor of $3 \leq r \leq 12$, the ratio of model time steps to atmospheric input time resolution ($1/r$ loops). Finally, non-computational work is parallelized and desynchronized in two stages ($1/rP$ loops).

The basic structure of the climate forcing data input procedure involves three steps. A process reads the appropriate point in the time series provided by the input files, using the NetCDF library [40, 41]. It then computes the spatial interpolation to the simulated land surface points. Finally, the interpolated data are distributed to the processes according to which grid points they are responsible for. The mapping of points to processors is done in a round-robin fashion to obtain rough load balance. This structure, and the changes to it described in this chapter, are illustrated in figure 4.1.

The initial design of this input presented many impediments to scalability. The remainder of this chapter discusses how these limitations have been largely eliminated. All data are shown in figures 4.2 and 4.3. The individual curves and bars are keyed 'A', 'B', and 'C' by the successive optimizations they depict, and 'R' denotes the baseline with no input optimizations, but with the round-robin mapping described in the original paper.

In collecting the data reported here, we coarsely separated time spent on collectives from idle time waiting at collectives. For each dynamic instance of a collective operation, we recorded the time elapsed from call to return in each process. We record the minimum observed time across all processes as the actual collective time of each instance, and the excess time beyond that minimum on each process as idle time. Given the approximate nature of

this methodology, it may be more appropriate for present purposes to simply consider collective and idle time in combination, rather than independently.

4.1 Matching Atmospheric Timestep with Model Timestep

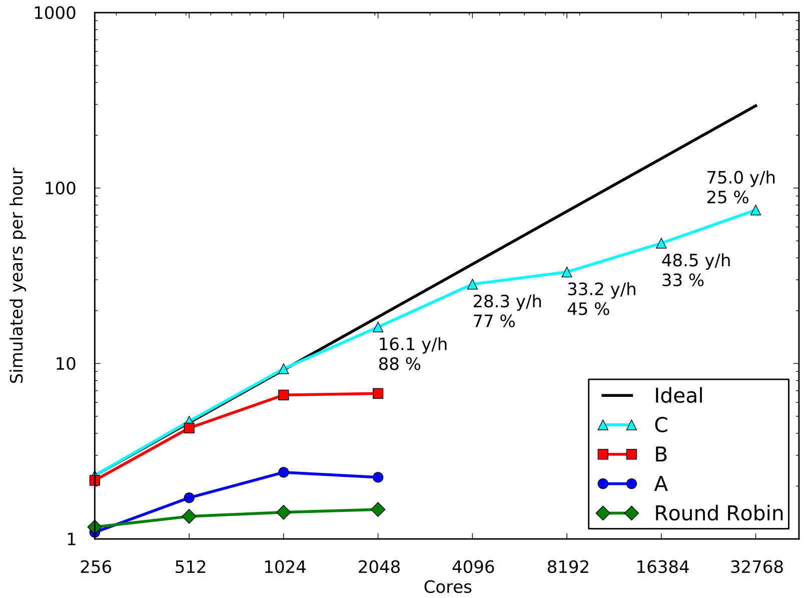
The ISAM model is typically run with a timestep of 30–60 minutes. The atmospheric data are provided at time intervals of 3–6 hours. In the initial implementation of ISAM developed by our collaborators, every step contained a call to the input procedure, which read the most recent atmospheric data from the source files on rank 0, interpolated it, and distributed it. Thus, the latency of filesystem access, interpolation, and `MPI_Scatterv` was on the critical path of successive model timesteps. Given that the same data would be provided for 3–12 steps in a row, this repetition was both redundant and created excess synchronization.

By modifying ISAM to reuse already-prepared data, we improved performance by a factor of 1.7× on 1024 ranks of Hopper and 1.2× on 1024 ranks of Edison. This improvement comes from both reduced time spent performing collectives, and reduced imbalance time waiting on heavily-loaded cores to reach each collective.

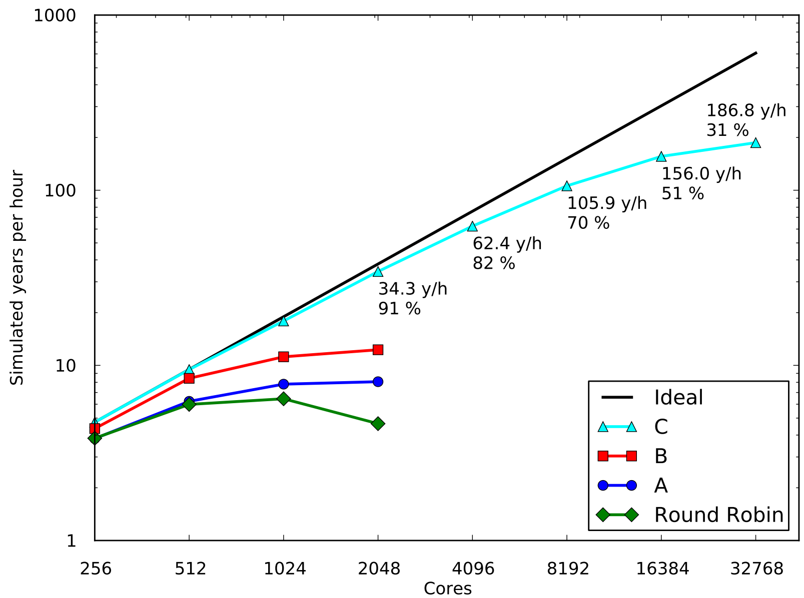
On 1024 cores of Hopper, where the round-robin and present optimized versions of the code obtain their best performance, the optimized code spends 88% less CPU time performing collectives and 24% less CPU time idling. The decline in collective time accounts for 62% of the 1.7× speedup and the decline in idle time accounts for a further 34% of the speedup.

On 1024 cores of Edison, where the round-robin and present optimized versions of the code also obtain their best performance, the optimized code spends 30% less CPU time performing collectives and 18% less CPU time idling. The decline in collective time accounts for 64% of the 1.2× speedup and the decline in idle time accounts for a further 33% of the speedup.

In as much as the work of several collectives was consolidated in time into a single one, this can be seen as an instance of the ‘batch collectives’ pattern (§ 3.1). The effect on execution structure is precisely analogous: widening the window between synchronizing collectives.



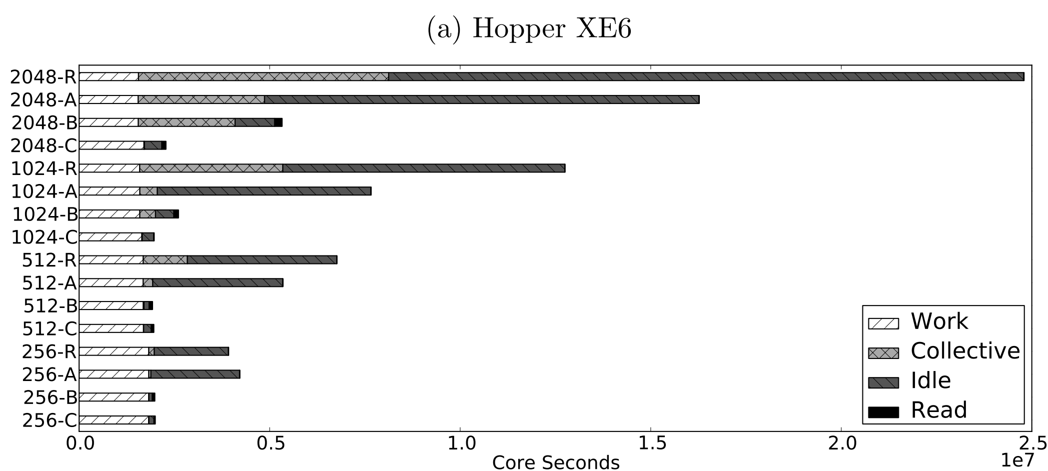
(a) Hopper XE6

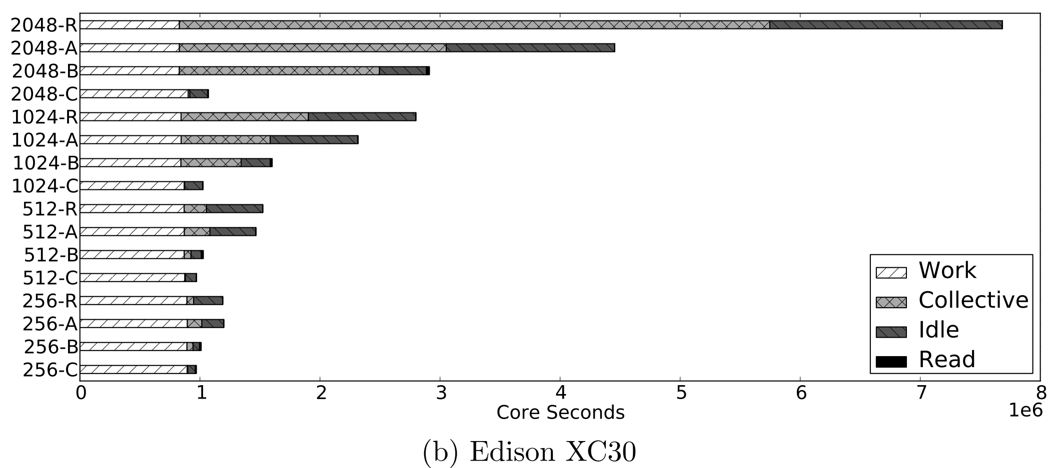


(b) Edison XC30

Figure 4.2: Overall scaling of ISAM as successive optimizations are applied to the input process for the climate forcing data. The graphs show years of simulated time per hour of execution wall time. Higher is better. Runs were for 5 years of simulated time. Labeled points show precise performance values and parallel efficiency relative to the most optimized code version 'C' on 256 cores.

(a) Hopper XE6





(b) Edison XC30

Figure 4.3: Breakdown of core-seconds spent by ISAM on different activities, as a function of scale and applied optimizations of the climate forcing data input process. Runs were for 5 years of simulated time. Lower is better.

4.2 Parallel-In-Time Reading and Interpolation

With input data read from the filesystem every few steps, the time per step scales poorly due to an Amdahl's law bottleneck on the time to access the filesystem and interpolate the data. Additionally, contemporary supercomputers offer high-bandwidth parallel filesystems to support their computational capabilities. By reading input data using only a single rank, ISAM was limited to the bandwidth of a single node.

Thus, our next optimization to ISAM's input procedure is to read and interpolate many steps worth of input data in parallel. At model timesteps where data must be read, each process reads and interpolates data for a step computed by incrementing the current timestep by its rank. At each subsequent step, the responsibility for distributing data cycles across the ranks until every rank has served as the root once.

In theory, this can reduce the elapsed wall time spent on reading and interpolation by $\mathcal{O}(P)$, since P such steps are performed in parallel. This is potentially limited by available bandwidth both in accessing the file data from the filesystem and in interpolating it in memory. It also presents the potential for interference and contention between these processes or other jobs on resources such as the Lustre metadata server. At larger scales, we observe these effects as decreasing average bandwidth per core and node and increased variability, as described in the original paper.

Note that the memory load imposed by this adaptation scales with the number of ranks per node, rather than the number of ranks in the entire job. The code does not distinguish between rank 0 and all other ranks; thus they all have the capacity to read and interpolate input. In a setting where the total memory on a node is insufficient to buffer a step's input per rank, we could adjust the scheme to only read and interpolate on every kth rank instead. This simply adjusts the above theoretical impacts by a constant factor of k, without changing the conclusion of improved scalability.

The improvement provided by this optimization over that described in section 4.1 is 2.76× on 1024 cores of Hopper and 1.3× on 1024 cores of Edison. In both cases, the reduction in idle time accounts for the bulk of the improvement. On both systems, this optimization allows the code to continue to gain performance at scales up to 2k cores, with efficiencies of 39% and 34% respectively, relative to the 256 core baseline.

4.3 Simultaneous Distribution of Multiple Steps

Having minimized wasted CPU time by fully parallelizing the reading and interpolation steps, the largest non-work portion of the execution time at the scaling limit of the code from section 4.2 is spent in collectives. On 2048 cores, these consume 46% of CPU seconds on Hopper and 57% of CPU seconds on Edison. On both systems, the increases in collective times account for the bulk of increased time relative to runs on 1024 cores.

To overcome this impediment, we observe that at the first scatter operation after climate forcing data is read and interpolated, the P processors each have data available for an upcoming timestep. However, in each scatter, only the cyclically selected root processor actually provides it. This misses a substantial opportunity for increased parallelism in usage of network resources.

We take advantage of this opportunity by replacing the per-step calls to `MPI_Scatterv` with an `MPI_Alltoallv` operation performed every P time steps. Rather than spatially scattering data representing the climate forcing at a single point in time, we now transpose the data from its provided temporal distribution (each core sends a distinct timestep for every point) to a spatial distribution (each core receives the time series for the points it owns). Once this is done, each core can independently execute P time steps with no communication. Thus, we note that this transformation is an application of the 'do more with each collective operation' pattern described in Section 3.2.

At first glance, this pre-distribution of input data may seem to dramatically increase memory usage on every node. However, this is not the case. To see why, we first observe that the additional memory consumption is a constant, independent of P. Suppose there are n points in total, and each one requires b bytes of memory for a single time step's climate data. Each core is responsible for n/P of those points. The data each core reads from disk as in section 4.2 is bn. In the transposition, each core receives the bn/P bytes for one future time step from each of the P cores. Thus, the total received data is just bn – exactly as much as every core read from disk. For the NCEPQ climate data set, $b = 24$ and $n = 192 \times 94 = 18,048$, totaling 423 kilobytes. For the CRU_NCEP data set, $b = 32$ and $n = 720 \times 360 = 259,200$, totaling 8 megabytes.

	Optimization	Lines added	Lines deleted	Total
(A)	Matching input steps	53	35	88
(B)	Parallel input	133	53	188
(C)	Simultaneous distribution	68	12	80

Table 4.1: Volume of code change required to implement each optimization

The effects of this optimization are striking. Where previously roughly half the execution time was spent in collectives at just 2k cores, this optimization reduces that time to less than 1% on both systems. Additionally, idle times also fell by over 50% on both systems, due to the longer period between synchronization points and greater opportunity for dynamic load variation to average out. Moreover, read times (though representing only a small proportion of execution) also fell substantially because of this optimization. We conjecture that this decrease is due to reduced contention when accessing the filesystem, since different cores can reach this phase across a wider timespan, as opposed to nearly simultaneously. Overall, this provides a 2.4× speedup on 2k cores of Hopper, and a 2.9× speedup on 2k cores of Edison. It also allows us to scale with continued speedups to 32k cores.

4.4 Summary

From our baseline code, we have obtained speedups of 6.58× on 1024 cores of Hopper and 2.78× on Edison. With all of the optimizations applied, we strong scale from 256 cores to 2048 process with an efficiency of 88% on Hopper and 91% on Edison. We accomplished this dramatic performance improvement through transformations that

- removed redundant collective calls (§ 4.1),
- parallelized input to use all processors instead of just one (§ 4.2), and
- exploited the parallel availability of much more input data to asymptotically reduce the frequency of calls to synchronizing collectives (§ 4.3).

In doing so, we also enabled essentially perfect strong scaling to more than an order of magnitude more processors than the code had previously been able to use.

Chapter 5

Desynchronizing Parallel File Output

List of Patterns Illustrated:

3.3 Send and consume data expected from a collective incrementally (§ 5.1)

3.4 Separate communication from coordination (§ 5.1)

3.5 Replace synchronizing collectives with coordination schemes (§ 5.3)

3.7 Semantic object naming (§ 5.2, 5.3)

Parallel file systems, such as Lustre and GPFS, are common to large scale parallel computers. They offer very high write bandwidth to keep up with the data generation of applications running on those systems. To obtain good bandwidth, applications and their supporting libraries must arrange operations so that each 'stripe' unit of a file is only written to by a single processor, ideally in large, aligned block units. Typically, computations are not structured so that the resulting data will be arranged this way naturally. Thus, in line with the description in Chapter 3, the code must at a minimum

1. have a means to designate the block-processor mapping,
2. ensure buffers are allocated on each such processor to hold pending data, and
3. communicate that the writing processors are ready to receive the data from the processors generating it.

MPI collective write operations satisfy these needs through synchronized operation. For the cost of that synchronization, they offer very high perfor-

mance in relocating the provided data and delivering it to the parallel file system.

The CkIO library was developed to provide CHARM++ applications with parallel filesystem support comparable to MPI-IO. This library has been through two full design generations to improve its performance and implementation flexibility. The later generation design is in production use in the CHANGA N-body particle simulation code used for cosmology and astrophysics. It is also in beta use for the NAMD classical molecular dynamics code for biomolecular research.

In the design progression of CkIO, we can see the incremental application of several patterns described in Chapter 3. In the first generation [21][1], the library separated communication from coordination (§ 3.4) and processed data incrementally (§ 3.3). The second generation partially applied semantic naming (§ 3.7), but not to the extent that it could eliminate coordination through synchronized operations. After describing the two generations of CkIO that have been built so far, the final section of this chapter outlines a potential future design that fully applies these lessons to eliminate the need to synchronizing operations when performing parallel output.

5.1 Initial Design

We have implemented an output forwarding layer for parallel applications written on top of the Charm++ parallel runtime system. As shown in Figure 5.1, the application's work is divided among several objects on each processor. The objects communicate with each other by asynchronous method invocation in an active-messaging scheme. Each processor can be executing work in one object while transmitting or receiving messages for others. This overlap of communication and computation is important for high performance.

Normal application development practices in Charm++ suggest the use of numbers of objects that correspond to a 'natural' decomposition of the problem being solved or the system being simulated, without direct regard for the number of processors in the system. The runtime can then map these objects

[1]The text and figures describing this design in Section 5.1 are adapted from the cited paper with permission. ©2011 IEEE

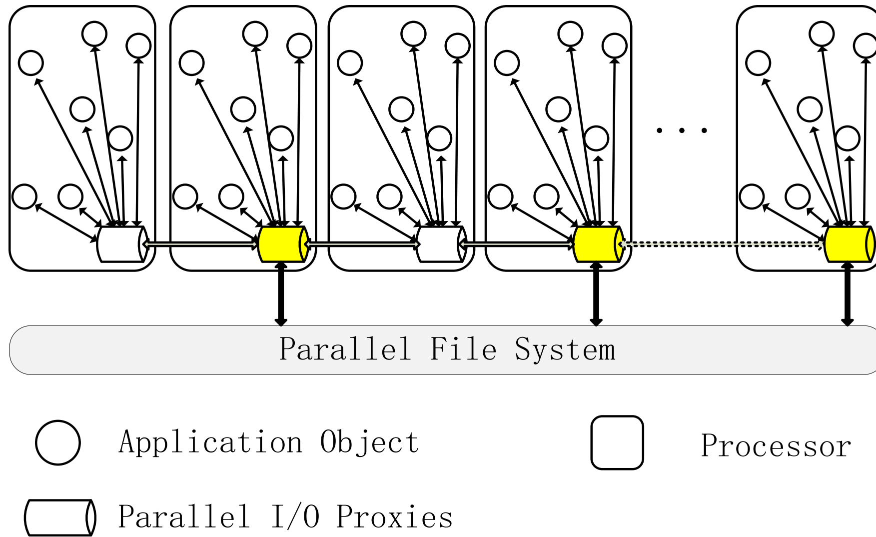


Figure 5.1: Architecture: Application objects communicate with local IO proxy objects, which exchange data amongst themselves and interact with the filesystem.

to optimize for load balance and communication patterns. However, for an I/O library, the many considerations of process- and node-level buffering, connection and contention limiting, and others drive toward an implementation that explicitly considers how many processors are available and how work is distributed among them.

The central element of our implementation is a one-per-processor collection of objects (known as a *Chare Group* in Charm++) that we will interpose between application-level objects that own the data and the underlying parallel filesystem. Groups communicate by the same asynchronous mechanisms as other Charm++ objects, but are addressed by the processor on which they reside, rather than by an abstract index. The 'Parallel I/O Proxy' objects of Figure 5.1 are implemented as a chare group, instantiated at application startup. The interface to this group, including the message entry points and sequencing logic, can be seen in Figure 5.2. The corresponding implementation code can be seen in Figure 5.3.

When the application wants to perform output, it signals the chare group to prepare for that process (`Manager::prepareOutout()`). The group signals its readiness to the application through a callback (`ready`), through which it delivers an opaque handle that the application should pass in along with the data. That handle acts as a 'parallel file descriptor,' allowing the proxy

```
1   group Manager {
2     entry void prepareOutput_central(std::string name, size_t bytes,
3                                      CkCallback ready, CkCallback complete,
4                                      Options opts);
5     entry void prepareOutput_distrib(int handle, std::string name,
6                                      size_t bytes, Options opts);
7     entry void prepareOutput_readied(CkReductionMsg *m);
8
9     /// Serialize setting up each file, so that all PEs have the same sequence
10    entry void run() {
11      for (filesOpened = 0; true; filesOpened++) {
12        if (CkMyPe() == 0)
13          when prepareOutput_central(std::string name, size_t bytes,
14                                     CkCallback ready, CkCallback complete,
15                                     Options opts) atomic {
16            // Default setting and error checking omitted
17
18            nextReady = ready;
19            thisProxy.prepareOutput_distrib(nextHandle, name, bytes, opts);
20            files[nextHandle] = FileInfo(name, bytes, opts);
21            files[nextHandle].complete = complete;
22          }
23
24        when prepareOutput_distrib[filesOpened](int handle, std::string name,
25                                                size_t bytes, Options opts) atomic {
26          if (CkMyPe() != 0) {
27            files[handle] = FileInfo(name, bytes, opts);
28          }
29
30          // Open file if we're one of the active PEs
31          if ((CkMyPe() - opts.basePE) % opts.skipPEs == 0 &&
32              CkMyPe() < lastActivePE(opts)) {
33            int fd = open(name.c_str(),
34                          O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
35            if (-1 == fd)
36              CkAbort("Failed to open a file for parallel output");
37            files[handle].fd = fd;
38          }
39
40          contribute(CkCallback(CkIndex_Manager::prepareOutput_readied(0),
41                                thisProxy[0]), filesOpened);
42        }
43
44        if (CkMyPe() == 0)
45          when prepareOutput_readied[filesOpened](CkReductionMsg *m) atomic {
46            delete m;
47            nextReady.send(nextHandle++);
48          }
49      }
50    };
51
52    entry void write_forwardData(int handle, const char data[bytes],
53                                 size_t bytes, size_t offset);
54    entry void write_dataWritten(int handle, size_t bytes);
55  };
```

Figure 5.2: The interface definition and coordination code for the I/O proxy group

```
1   struct Options {
2     /// How much contiguous data (in bytes) should be assigned to each active PE
3     size_t peStripe;
4     /// How much contiguous data (in bytes) should a PE gather before writing it out
5     size_t writeStripe;
6     /// How many PEs should participate in this activity
7     int activePEs;
8     /// Which PE should be the first to participate in this activity
9     int basePE;
10    /// How should active PEs be spaced out?
11    int skipPEs;
12  };
13
14  struct FileInfo {
15    std::string name;
16    Options opts;
17    size_t bytes, total_written;
18    int fd;
19    CkCallback complete;
20  };
21
22  class Manager : public CBase_Manager {
23    /// Application-facing methods, invoked locally on the calling PE
24    void prepareOutput(const char *name, size_t bytes,
25                       CkCallback ready, CkCallback complete,
26                       Options opts = Options()) {
27      thisProxy[0].prepareOutput_central(name, bytes, ready, complete, opts);
28    }
29
30    void write(int handle, const char *data, size_t bytes, size_t offset) {
31      Options &opts = files[handle].opts;
32      do {
33        size_t stripe = offset / opts.peStripe;
34        int pe = opts.basePE + stripe * opts.skipPEs;
35        size_t bytesToSend = min(bytes, opts.peStripe - offset % opts.peStripe);
36        thisProxy[pe].write_forwardData(handle, data, bytesToSend, offset);
37        data += bytesToSend;
38        offset += bytesToSend;
39        bytes -= bytesToSend;
40      } while (bytes > 0);
41    }
42
43    /// Internal methods, used for interaction among IO managers across the system
44    void write_forwardData(int handle, const char *data, size_t bytes, size_t offset) {
45      // Omitted error checking and interruption handle code for simplicity
46      pwrite(files[handle].fd, data, bytes_left, offset);
47      thisProxy[0].write_dataWritten(handle, bytes);
48    }
49
50    void write_dataWritten(int handle, size_t bytes) {
51      files[handle].total_written += bytes;
52
53      if (files[handle].total_written == files[handle].bytes)
54        files[handle].complete.send();
55    }
56
57    int filesOpened;
58    int nextHandle;
59    std::map<int, FileInfo> files;
60    CkCallback nextReady;
61
62    int lastActivePE(const Options &opts) {
63      return opts.basePE + (opts.activePEs-1)*opts.skipPEs;
64    }
65  };
```

Figure 5.3: The implementation of the I/O proxy group

objects to look up the parameters (which processors, stripe size, offsets, etc.) associated with each particular target file. Once the system is ready, the application objects will pass their portions of the data to the local element of the group (`Manager::write()`), which will redistribute the data according to the plan and perform write operations as whole stripes are assembled.

Let us consider how this design fits the the patterns of separating communication from coordination and transferring data incrementally. The methods named `prepareOutput_*` provide all of the necessary coordination in a non-blocking fashion, with no dependence on the data to be written. Thus, file handles and buffers can be prepared while the computation generates the data. Then, the `write` methods transfer the data between the processor on which it's generated and the processor that will be responsible for writing it. For a processor that will write data, any chunk received is sufficient to write it out. There is no dependence on that processor receiving other data, or acting as an intermediary for data transfers to other processors.

Control Flow For each file, our IO forwarding layer takes as parameters a stripe size, a number of processors to use, a starting processor, and a numeric separation between those processors. It applies these parameters in a straightforward fashion to direct data provided by the application to the processor that will eventually pass it to the filesystem.

When application code is ready to write data to persistent storage, it calls the IO forwarding layer with a file name, size, and the parameters listed above. It also passes two callbacks, for signaling readiness and completion. These callbacks can represent a function to call or, more commonly, the target of a subsequent message send. The IO forwarding layer communicates internally to ensure that all processors know how that file is to be handled. Every processor acknowledges these instructions by contributing to a parallel reduction operation. When the reduction reaches the root processor, it triggers the ready callback, passing a handle object used to look up the file. These steps are illustrated in Figure 5.4.

As shown in Figure 5.5, the application code sends this handle to any objects with data to be written. Each of these objects call the IO proxy object local to the processor on which they reside, passing the handle along with their data buffers. The local proxies forward data to proxies on other processors as needed, via the `Manager::write_forwardData()` method. From the

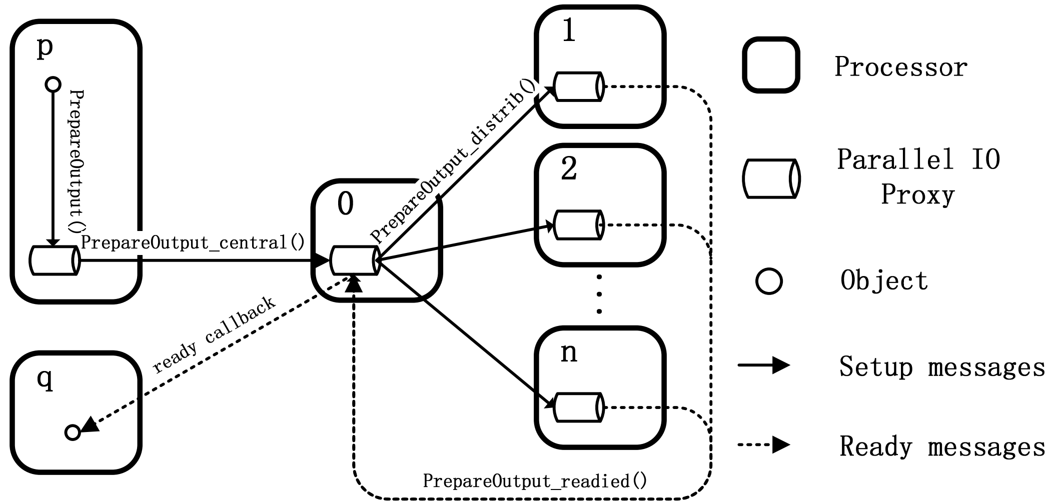


Figure 5.4: Flow of execution to prepare for writes to a file

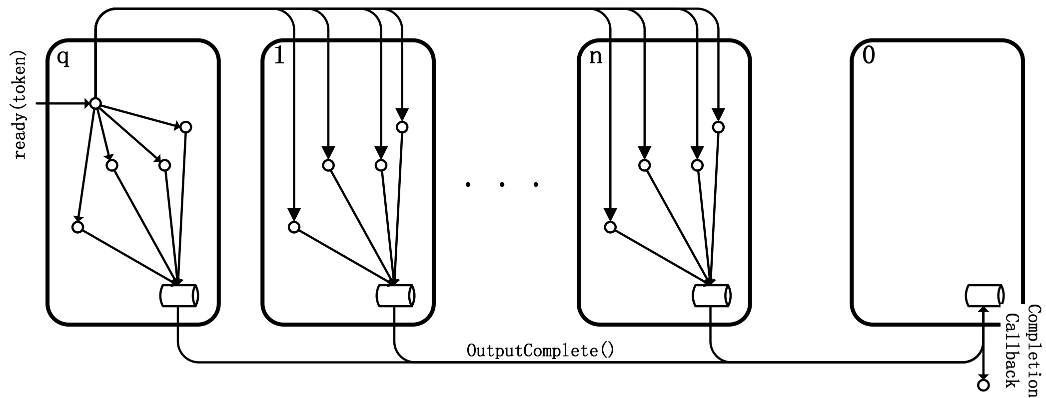


Figure 5.5: Flow of execution once a file is ready to be written

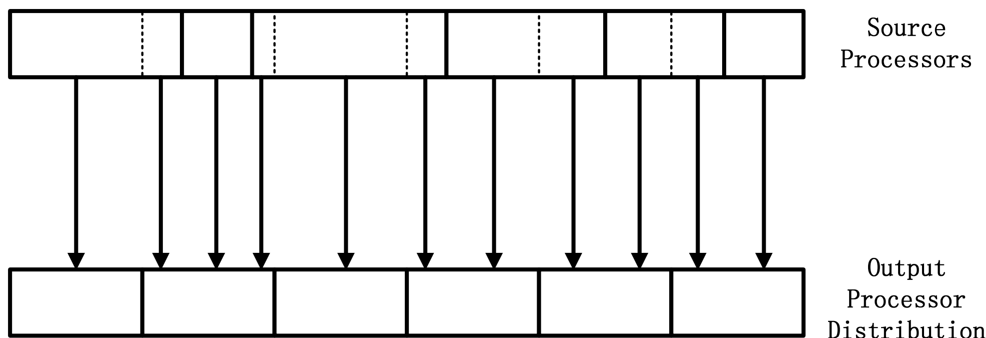


Figure 5.6: Mapping chunks of a file from different processors to whole stripes

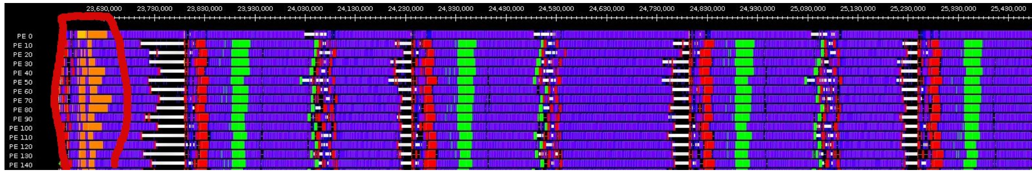
application's point of view, the forwarding process completes immediately, and buffers can be reused or discarded without delay. When an IO proxy has written all the data it is responsible for, it notifies the master. When the master has heard from the IO proxies on every processor writing data, it signals the completion callback.

Striping Given the substantial documented effect of matching application writes to filesystem striping, the optimization priority is to constrain each output processor's writes to distinct stripes. For each segment of data that the application wishes to write, we compute which stripes it intersects based on the stripe size parameter (Figure 5.3 line 33), as shown in Figure 5.6. Then, for each stripe, we compute which processor is responsible for writing that stripe to the filesystem (line 34). We send messages containing each stripe chunk to the IO proxy on its respective processor (line 36). When the IO proxy receives the data, it passes it to the filesystem (line 46), secure in the knowledge that it will not contend with other processors for access to that stripe, and notifies the root processor (line 47) so that we can tell when the process is complete (lines 50-55).

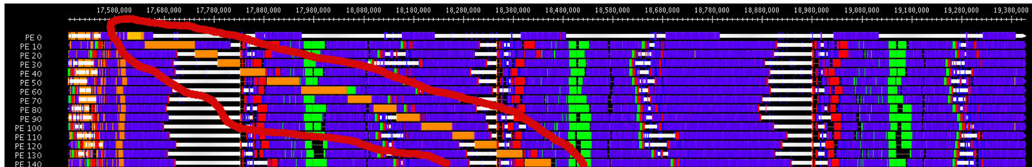
5.1.1 Evaluation

In order to evaluate the effectiveness of our approach, we have adapted NAMD, an existing Charm++ application, to use our output framework. NAMD is a popular ($> 40,000$ users) code for classical (i.e. Newtonian) simulation of large (up to 100 million atoms) biomolecular systems at atomic

(a) Multiple files, dummy scheme



(b) Single file, one at a time



(c) Single file, simultaneously

Figure 5.7: Execution timelines of unmodified NAMD performing an output step generated by the PROJECTIONS tool (with comparable time scales).

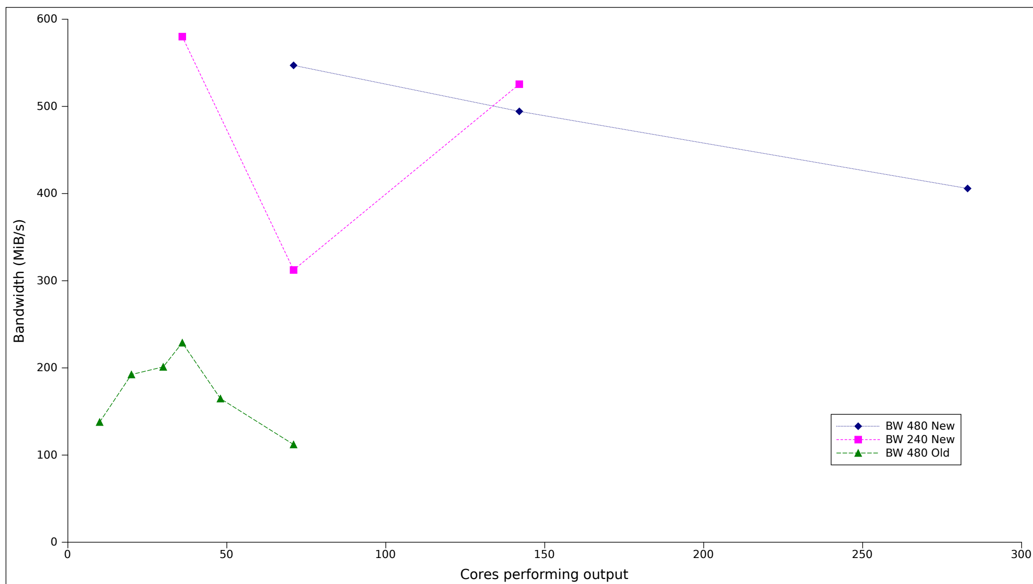


Figure 5.8: NAMD's output bandwidth on 240 and 480 processors of Jaguar and Kraken, with varying numbers of processors touching a single output file.

scale. Its behavior of periodically dumping the state of the computation (i.e. the positions and velocities of all particles) to disk is typical of many scientific applications.

In its present version, NAMD contains mechanisms to do parallel output. Specifically, the particles' positions are collated on a subset of the processors, which coordinate to write their data to the filesystem. This coordination amounts to a control on how many of them will actually make `write()` calls simultaneously. This scheme was implemented to enable scalability to large target systems without exceeding the available memory on individual nodes. It makes some expedient choices to attain acceptable performance, and leaves several knobs for the user to set 'appropriately'. It takes no account of the type or parameters of the filesystem on which it runs.

NAMD's parallel output scheme [42] introduces a layer of indirection between the application objects and the IO processors, to balance the IO and memory load among the processors performing IO. Depending on how many processors are involved in output and when they perform their operation, performance can vary wildly. Figure 5.7 shows execution traces of NAMD for simulating a 2.8-million-atom virus molecule on 32 nodes of Jaguar PF (using 10 cores/node and one output processor per node) before our modifications. In the figure, the highlighted dark yellow bars represent the time spent on output, while bars in other colors represent the time of different types of force computation. Figure 5.7a shows each processor writing to a separate file, a scheme not supported by surrounding tools or NAMD's own input reader, as a point of reference; Figure 5.7b shows all writes going to one file one after another while figure 5.7c shows all writes going to one file but simultaneously. Note that these traces are presented on approximately the same timescale, illustrating that an incautious 'all-at-once' mode of output can be disastrous.

The disappointing performance in NAMD's working output code can be explained by the various kinds of contention that it creates in the storage subsystem. Among the processors performing output, data is divided among them evenly, for the sake of load balance. Since the simulation data set does not precisely divide into neat power-of-two size chunks, this means that there is no alignment of each processor's writes to filesystem or storage hardware boundaries. Additionally, because each processor's responsibility crosses stripe boundaries, there are substantially more connections to the

storage nodes than are necessary.

In Figure 5.8, we present measurements of the average bandwidth obtained by coordinate outputs from a 10-million atom simulation on Jaguar, a Cray XT5 with a Lustre-based storage system. Each output step wrote 283MiB of data, and the data presented are the average of 8 output steps each. We can see that NAMD's existing implementation peaks around 220MiB/s using 36 out of 480 processors for output, and falls off rapidly. In contrast, when NAMD is adapted to use our library, it sees substantially better performance on both 240 and 480 processors. Measured bandwidth ranges from 305MiB/s to 580MiB/s. Moreover, our attained bandwidth does not decay as rapidly as one leaves the 'sweet spot' of output processor count. Thus, it is less reliant on the user to choose a good value for the number of output processors. Finally, because it successfully uses a larger number of cores, we are able to run much larger simulations (with a correspondingly larger memory footprint) without output times increasing sharply.

Measurements by the machine's operators [43] suggest that the underlying Lustre filesystem can offer bandwidth of at least 1.5 GiB/s to hundreds of processors used here. Thus, there is still a long ways to go in minimizing time spent performing output.

5.2 Deployed Design

The second generation design for CkIO introduces several improvements. It implements configurable stripe buffering so that blocks of file data filling a whole stripe of the underlying file system get aggregated and written in full, rather than as each increment of data arrives. This trades some pipelining for lower overhead interaction with the backing storage system (particularly, avoidance of a read-modify-write cycle of partial stripes on RAID volumes).

The later design demonstrated early examples of a few patterns from Chapter 3. It separates coordination from communication at two levels. Files can be opened once, with many subsequent sessions writing to disjoint offsets within them. Each session only needs to declare its offset and length within the target file to obtain a handle that the application can use to submit data. Thus, the session setup (object construction, buffer allocation, etc.) can occur before any data to be written has even been computed, or concurrent

with its computation. It also moves more toward semantic naming of the objects that will aggregate the output data into aligned stripes and write it to the file system.

This version of the library's design has not yet been published. It was studied in a performance modeling report for a class project by Ronak Buch and Sam White [44]. Their experiments found that the library was capable of transferring data at several gigabytes per second. The results presented likely underestimate its performance in actual usage, because they timed the entire operation, including all setup, as the denominator for the measured bandwidth. As noted earlier, it is in use by ChaNGa and will shortly be integrated for use in NAMD to replace the lower levels of its current parallel output implementation [42].

5.3 Design for Fully Desynchronized Coordinated Output

CkIO synchronizes to coordinate initial offset and total data size for each write session on a given file. Data transfer is then incremental and independent. Final flushing and outcome reporting is again synchronized.

Taking an asynchronous execution model as a pre-requisite, we can go further. With better object naming (absolute file offset rather than session-relative) and completion detection, the size and offset of a session would not need to be provided up front. With that modification, session initiation would not need to be synchronized across callers or the backing callee objects. Instead, independent writers could indicate a logical session sequence number that each write contributes to. All data filling in whole blocks could be written out opportunistically. The system could then use the mechanisms described in Section 3.4 to detect when the full data of the session has been transferred, ensure the potentially incomplete boundary blocks get written out, call suitable filesystem flush operations, and optionally report completion.

In the case of variable amounts of data from each contributor, or a number of contributors or size of contributions that is not known to other contributors, the standard strategy is to perform a reduction and/or scan (parallel prefix) as necessary to obtain that information, prior to submitting the data.

An alternative could have processors submit data independently with suitably generated ordered keys, using termination detection of some form (as described in Section 3.4), and do the necessary coordination independent of the specific processors providing the data to be written.

Chapter 6

Dense LU Factorization

List of Patterns Illustrated:

3.1 Batch (blocking on / waiting for) collectives (§ 6.2.5–6.2.6)

3.2 Communicate more with each collective, to reduce frequency (§ 6.3)

3.3 Send and consume data expected from a collective incrementally (§ 6.2.4)

3.5 Replace synchronizing collectives with coordination schemes (§ 6.2.2)

3.6 Replace synchronizing collectives with p2p messages that achieve the desired effect (§ 6.2.4)

Dense LU factorization is commonly used as a means to solve dense linear systems. A set of n linear equations in n variables is solved by performing LU factorization and solving the resulting triangular systems. The algorithm has a few different variants, one of which is Crout's algorithm which performs an in-place factorization. Numerical stability is achieved via partial pivoting.

Most parallel formulations of LU are blocked algorithms with underlying sequential operations delegated to a high performance linear algebra library (e.g. an implementation of BLAS). The matrix is typically decomposed into square blocks of size b^2 (shown in Figure 6.1) and distributed across a set of processors.

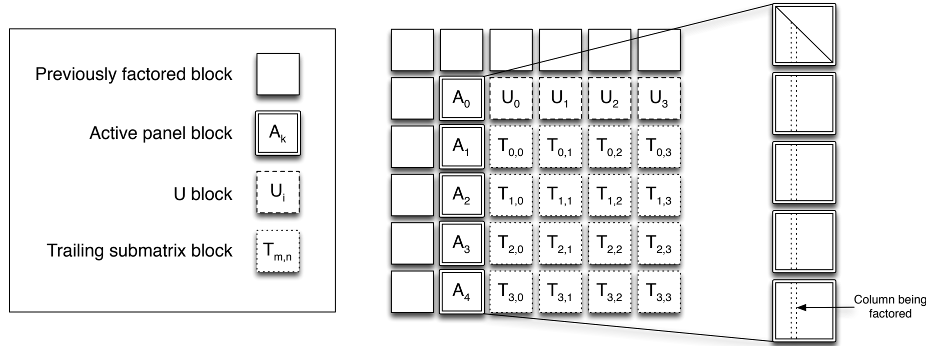


Figure 6.1: The matrix is decomposed into square blocks, which take on different roles as the factorization proceeds.

6.1 Background

This section provides background information on the problem of parallel dense LU factorization.

6.1.1 Algorithm

The parallel factorization process can be described as follows:

1. for *step* in $0..\frac{n}{b} - 1$:

   Active panel blocks are those at/below diagonal block *step*

   (a) Partial Pivoting for *column* in $0..b$: (on each active panel block)

      i. Each block identifies its maximum value below the diagonal in the current *column* within that block and contributes to a reduction among the active panel blocks.

      ii. The result of the reduction identifies the *pivot row*, which is swapped to the diagonal position and broadcast to all of the active panel blocks.

      iii. Each active panel block performs a rank-1 update of the section after *column* with multipliers from *column* and the pivot row.

   (b) The sequence of pivot exchanges is broadcast to the blocks of U

and the *trailing submatrix*, which communicate to apply the same swaps as the active panel.

(c) Active panel blocks send their contents, each a portion of L, to the blocks to their right.

(d) U blocks to the right of the diagonal each perform a *triangular solve*, and send the result to the blocks below them.

(e) Blocks in the trailing submatrix each compute a *trailing update* as the product of the L and U blocks they have received.

6.1.2 Granularity Spectrum

The factorization presents a challenging spectrum of computation and communication grain sizes. The trailing updates comprise the bulk of the computation in a dense LU solver. Each trailing update is an $\mathcal{O}(b^3)$ matrix-matrix multiplication (i.e. a call to the `dgemm()` level-3 BLAS routine). The triangular solves (via `dtrsm()`) are of similar computational cost. Each trailing update or triangular solve takes tens of milliseconds for the block sizes common on today's architectures. Large messages drive the heavy computation kernels. In contrast, the active panel is communication intensive, with b small-message pivot reductions and broadcasts occurring in rapid succession, interspersed with smaller computations. Each of these fine-grained steps on the active panel take hundreds of microseconds to single digit milliseconds.

6.1.3 Lookahead

Ideally, every processor would remain busy during the entire factorization process. However, in each step, only a subset of processors own blocks that participate in the active panel. Thus, to avoid idling processors, work from multiple steps must be overlapped. The extent of the overlap (specifically, the number of steps that the active panel runs ahead of trailing updates) in an implementation of dense LU is known as its *lookahead depth* [45].

6.2 Charm++ Implementation

In our Charm++ implementation of dense LU factorization, each block is placed in a message-driven object, driven by coordination code written in Structured Dagger [46]. The coordination code describes the message dependencies and control flow from the perspective of a block. Thus, every block can independently advance as it receives data and bulk synchrony is avoided by allowing progress in the factorization when dependencies have been met. With many blocks per processor, the Charm++ [47] runtime system inherently provides dynamic overlap of communication and computation by scheduling blocks that have received the necessary data. In general, the system ensures high utilization, since some blocks on each processor should always have work. Others implementations dynamically interleave the work performed on various blocks, either by introducing task parallelism to HPL [48] or by spawning many light-weight threads in UPC [49].

This style of message-driven programming allows a clear and concise representation of the algorithm without explicit buffering of messages. When a message arrives, the Charm++ runtime system invokes a method on an object or buffers it if the object is not ready to process the message.

By representing each matrix block as a separate object, the description of the parallel algorithm is separated from the particular details of its execution. Additionally, the control flow executed for each block is directly visible in the code; it is linear and effectively independent of other activity on its host processor.

Due to the simplicity of expression in the locally message-driven style, the source code for our implementation of the factorization library is approximately 1,650 lines long [30].[1] This is shorter than HPL, which is around 12,000 lines and the UPC implementation [49], which is around 4,000 lines of code. Furthermore, the distribution of blocks to processors is not embedded in the expression of the parallel factorization algorithm, but is instead localized to discrete mapping functions. The flexibility this provides was previously used to study some atypical mappings in an earlier non-pivoting version of this code [50] and to study optimized mappings of pivoting LU factorization on modern multicore cluster and supercomputer nodes [35].

[1] As counted by David Wheeler's SLOCcount.

Prioritization

On each processor, the work units for which input data has arrived are placed in a priority queue. The priorities are set by the type of work a unit represents and the index of its target block in the matrix. The basic priority scheme gives high priority to active panel work and U triangular solves (to generate work quickly), and lower priority to trailing updates.

6.2.1 Dynamic Lookahead for Greater Overlap

Bulk synchronous implementations, such as HPL [51], require a fixed lookahead depth and restrict the overlap of steps to that amount. This restriction is due to memory limits of the machine; delaying the computation by increasing the lookahead depth means that memory for input blocks accumulates and then must be controlled. Due to implementation complexity and performance portability issues, the ScaLAPACK library [52] does no lookahead (i.e. its lookahead depth is 0).

In an asynchronous, dataflow parallel programming model, the availability of input data immediately triggers the next steps in the algorithm that depend on it. For typical, iterative, scientific algorithms, the amount of parallelism in the computations remains more or less steady as the algorithm progresses. Such algorithms can be expressed in pure dataflow semantics and can exploit asynchronous execution models without other concerns. However, the LU factorization has varying amounts of parallelism at different stages of the computation. When expressed in the dataflow model, it can cause unbridled spikes in memory usage because early steps in the algorithm trigger large amounts of data movement to feed the subsequent steps. For factorizations involving large matrices relative to the size of available memory, this can cause premature and unsuccessful termination of the execution. Hence, although lookahead is a natural consequence of using the dataflow model, it still needs to be moderated by a continuous awareness of memory and bandwidth utilization. This leads to a reality where the dataflow semantics are adaptively throttled by a system that monitors memory usage and other system parameters.

In a message-driven, asynchronous environment, LU can be implemented to allow dynamic lookahead: the diagonal can progress without a bound

before the rest of the matrix finishes updating. Our solver implements dynamic lookahead, using a dynamic *pull-based* scheme to constrain memory consumption below a given threshold.

To implement the pull-based scheme, each processor has a distinguished *scheduler object* in addition to its assigned blocks. The scheduler maintains a list of the blocks assigned to its processor, and tracks what step they have reached. Within the bounds of the memory threshold, it requests blocks from remote processors that are needed for local triangular solves and trailing updates. To eliminate the possibility of deadlock, the order in which operations are executed, and hence remote blocks requested, must be carefully selected. Husbands and Yelick point out [49] that selecting updates in step order is deadlock-free, but suggest that there may be a general solution for finding a deadlock-free selection order of trailing updates using the dependencies between blocks.

In an instance of the 'coordination schemes instead of synchronizing collectives' pattern (§ 3.5), Section 6.2.2 describes the dependencies between the blocks and how our implementation uses this structure to safely reorder the selection of trailing updates to execute. We paired this with an instance of the P2P replacement (§ 3.6) and incremental transfer (§ 3.3) patterns in the form of on-the-fly dynamic multicast operations to transfer the large matrix blocks, described in Section 6.2.4.

6.2.2 Dependence Scheduling of Large Block Operations

To achieve high machine utilization, and hence good performance, the active panel and trailing update calculations must be overlapped. Specifically, the active panel for a step t should finish early enough before the trailing updates from step $t-1$ such that no processor idles while waiting for input data for step t's trailing updates. In strong scaling scenarios and in the large weak-scaled runs, each active panel may take longer to factor than all of the trailing updates it generates. Thus, to maintain overlap throughout the factorization, active panels should be executed as eagerly as possible while staying within memory limits.

In a matrix decomposed into $N \times N$ blocks, the factorization of active panel t enables $(N-t)^2$ trailing updates. However, only $N-t$ of those updates must complete before the factorization of active panel $t+1$ can start. Despite

this, the UPC implementation allocates memory for these updates in strict step order, as a conservative means to avoid deadlock. Thus, with a matrix that is large relative to available memory, it must execute most of each step's updates before making space for the next step, and lookahead is very limited until late in the factorization, when little of the matrix remains to be updated.

In order to explore less conservative scheduling policies, we formalize the dependence structure in terms of *planned operations*, those for which memory has been reserved. These include both triangular solves and trailing updates, but not pivoting, since it consumes a minimal amount of memory.

For each block (x, y), major operations on it are denoted as a triple (x, y, t). Every block will go through trailing updates

$$(x, y, t) \quad | \quad 0 \leq t < min(x, y).$$

Blocks below the diagonal, $x > y$, become part of the active panel after their last trailing update and so have no more operations to plan. Blocks above the diagonal, $x < y$, complete their trailing updates and then perform a triangular solve, whose triple will always be of the form (x, y, x). For simplicity, this formulation conservatively subsumes pivoting operations into whatever major operation follows them, since they consume little additional memory (obviating the need to plan them explicitly).

A triangular solve (x, y, x) depends on its final trailing update

$$(x, y, x-1) \prec (x, y, x)$$

the final trailing updates to its associated active panel

$$(i, x, x-1) \prec (x, y, x) \quad | \quad x \leq i < N$$

and (due to pivoting) the previous step's trailing updates on the column below it

$$(i, y, x-1) \prec (x, y, x) \quad | \quad x \leq i < N.$$

A trailing update (x, y, t) directly depends on the previous update to that block

$$(x, y, t-1) \prec (x, y, t) \quad | \quad t > 0$$

and the triangular solve of its U input

$$(t, y, t) \prec (x, y, t).$$

The dependence on U creates a transitive dependence on the corresponding block of L, and so need not be considered explicitly.

If operations are planned strictly in step order, with triangular solves preceding trailing updates, these dependencies are effectively expanded to include the entire trailing submatrix at step $t-1$ for every step t triangular solve. Under that policy, it is clear that all dependencies will be planned before their dependents, and this will create a deadlock-free schedule. Moreover, this is a policy that every processor can follow independently, without communication to coordinate decision-making. This is the policy followed by the UPC implementation.

We have two desiderata for a less conservative scheduling policy. First, it should enable overlap to the greatest extent possible. Second, it should require little or no non-local information to operate correctly, because coordinating multiple processors operating asynchronously can be expensive, error-prone and difficult to reason about.

Given a set of operations S that can be considered done or planned at some point in time, the operations $E(S)$ eligible for planning can be determined by which dependencies are satisfied:

$$\begin{aligned} E(S) = \quad & \{(x, y, t) | \quad t < min(x, y) \quad \wedge \\ & \qquad ((x, y, t-1) \in S \vee t = 0) \quad \wedge \\ & \qquad (t, y, t) \in S\} \quad \cup \\ & \{(x, y, x) | \quad x < y \wedge (x, y, x-1) \in S \quad \wedge \\ & \qquad (\{(i, x, x-1), (i, y, x-1) | x \leq i < N\} \\ & \qquad \subseteq S \vee x = 0)\} \end{aligned} \tag{6.1}$$

If these precise dependencies are applied on a local, per-processor basis, deadlock can result, as shown in figure 6.2. This occurs because the trailing updates necessary to pivot data for some triangular solve can be mutually blocked by other trailing updates across two or more processors. Step-order planning does not give rise to cases like this; specifically, it guarantees that the trailing updates on one processor needed to generate pivots for a trian-

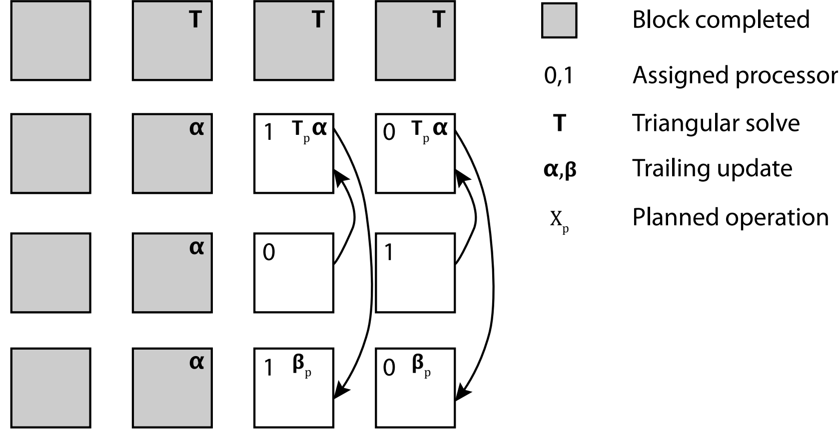


Figure 6.2: Possible deadlock situation if only local dependencies are considered. This is an example with two processors and an allowed planning depth of two. The two processors that try to execute β_p are dependent on their T_p triangular solves. These two solves are dependent for pivoting on the two non-local blocks that have not completed α. Since these blocks are not planned, deadlock ensues.

gular solve would be planned before any trailing updates that may depend on that triangular solve's output.

Suppose that processor has the information that the triangular solve $(t + 1, y, t + 1)$ has completed, despite some of its blocks in column y not having been updated to step t, and thus unable to pivot with $(t+1, y)$. This means that those blocks contained no pivot rows for that step. The completion of the triangular solve with no contribution from those blocks lets that processor delay planning updates to those blocks, in favor of step-wise later updates that might be closer to critical for the active panel.

Our baseline implementation follows the conservative step order that avoids the possibility of deadlock. However, we deviate from step order by exploiting the information about finished triangular solves in a limited fashion. When a U block on the first super-diagonal does its triangular solve, it broadcasts a notice of this progress to the scheduler objects on all of the processors. That broadcast is used to release conservatively set dependences that would hold back the next active panel. In its limited form, the benefits of this are modest: about a 0.5% increase in throughput as a fraction of the system's peak.

A more complete implementation would make the same release notification from every triangular solve, allowing columns that are a few steps away from being on the active panel to run further ahead of other columns further to

the right in the matrix. The challenge, then, would be for each scheduler to determine how much memory to allocate to updates on which part of the matrix, given a need to balance fastest immediate progress with having work to do when otherwise idle.

6.2.3 Experimental Setup

The experiments described in the remainder of this section were performed on two supercomputers.

The first was the the Intrepid Blue Gene/P system at Argonne National Laboratory. Each node of Intrepid had 4 PowerPC 450 cores running at 850 MHz. The peak performance of each core is 3.4 GFLOP/s. On Intrepid, we used matrix blocks of 300×300 doubles up to 1k cores on Intrepid, and 150×150 doubles at larger scales, where not otherwise specified.

The second was the Jaguar Cray XT5 system at Oak Ridge National Laboratory. Each node of Jaguar had a pair of 6-core AMD Opteron 'Istanbul' processors running at 2.6 GHz. The peak performance of each core is 10.4 GFLOP/s. We used matrix blocks of 500×500 doubles for experiments on Jaguar where not otherwise specified.

Experiments to study the impact of particular optimizations were performed with all other optimizations described here and in a paper on mapping this code [35] enabled. Thus, the results present the criticality of each optimization to the overall performance obtained.

6.2.4 Limiting Network Contention with Dynamic Multicasts

Dense LU factorization is not generally considered a network-intensive parallel operation, since its computation asymptotically dominates its communication. However, it presents communication patterns that involve moving large volumes of data (matrix blocks) in a 'bursty' fashion from a few source processors to many recipients. In a synchronous implementation, these bursts of communication can be implemented as efficient collective broadcasts to statically known subsets of processors (e.g. the 'process rows' and 'process columns' in HPL). In a pull-based implementation, however, recipient processors may request blocks at any time, and the owner of a block will need to respond quickly, so that the requester does not run out of work and idle.

Testing shows that responding to these requests one-by-one as they arrive leads to network saturation on processors owning blocks that are in high demand. This saturation stretches the time the sender spends responding, and delays arrival of the response on requesting processors.

To address network saturation, we dynamically batch block requests to efficiently multicast blocks and spread the network load. Requests for a block arriving before that block is ready are batched and sent in a single multicast when the block's computation is complete. However, requests arriving later have no inherent method for batching into multicast groups. Thus, we limit the number of large outgoing messages that each processor may have in flight at a time. When a request for a block arrives, the requesting processor is put on a list of requesters for the block, and the block puts itself in a send queue. Eventually, as sends complete, each queued block will reach the head of the queue.

When a block reaches the head of the send queue, it will have accumulated a list of several processors that have requested the block since the last time that block was sent. The list of requesting processors participating in a multicast is transmitted by constructing a binary spanning tree on the fly. This enables dynamic, asynchronous collective communication with negligible additional latency and little message size overhead.

Figure 6.3 shows that our multicast scheme substantially outperforms point-to-point responses to each request. This scheme illistrates the patterns of replacing synchronous collections with a purpose-specific coordination scheme and point-to-point messages. It avoids making processors that would be near the leaves of a static broadcast tree wait for interior nodes of that tree. Thus, we avoid paying for late arrival and noise on those intermediate processors.

6.2.5 Exclusive Scheduling Classes

In applications that mix large grains of sequential execution with latency-sensitive communication operations, there is a tension between computational throughput and responsiveness: a single processor's work tends to execute most efficiently when presented in large chunks; however, when such compute kernels are running, reacting to incoming messages is difficult or impossible.

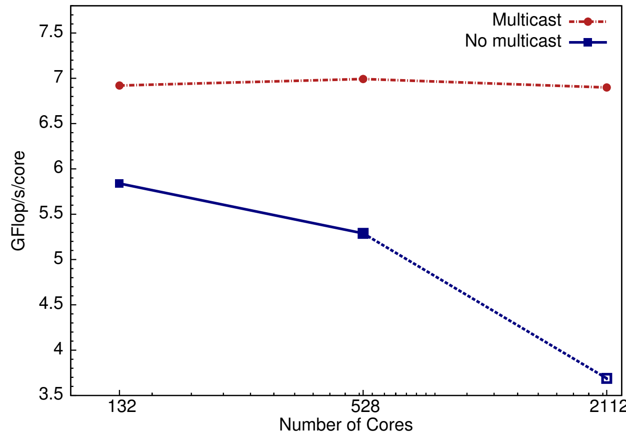


Figure 6.3: Performance effects of agglomerating work and multicasting it on-the-fly to destinations. The 2112 point of the no multicast curve did not finish in the allotted time; hence it represents the maximum performance that configuration could have achieved.

In many asynchronous programming models, work is decomposed into units and each processor draws from a local queue of available work units. When a processor finishes executing a work unit, it will select the highest priority work unit available in its queue as the next. In general, as long as work units are available it is beneficial to execute them to avoid idle time and maintain high utilization. However, if the highest priority work unit available is not on the critical path and is relatively long, it may delay execution of a critical work unit that will arrive soon. Therefore, it may be beneficial for that processor to idle briefly, waiting for the higher priority in-flight work unit, rather than opportunistically executing the already available work unit.

In any asynchronous execution model that is opportunistic, ensuring that specific classes of work execute uninterrupted is a challenging problem. The problem is exacerbated if there are large grain size variations across these classes. Decreasing the interleaving of a critical class with grains from other classes may be important for ensuring that the critical path computation or communication proceeds quickly.

Existing applications and runtime environments resolve this tension using a variety of methods:

- *Interrupts/Preemption*: Long stretches of execution can be interrupted when a latency-sensitive event occurs, with the reaction preempting the ongoing computation. This method can achieve excellent responsiveness, but requires low-level hardware or runtime support, may be

overhead prone, and is difficult to program.

- *Polling*: The code for a long stretch of work can be adapted to explicitly poll for the arrival of a critical message and respond to it before resuming execution. This in-line interruption introduces overhead, but it also presents deeper issues of determining polling frequency. Moreover, it is not always desirable or possible to poll from within optimized compute kernels like those found in BLAS libraries.

- *RDMA*: If the critical operation is purely a data transfer operation on precomputed data, this problem can be resolved using remote direct memory access. With hardware support, this can be very efficient, since the ongoing computation can continue executing unaffected. However, only very simple operations are possible. Hardware and programming environment support are also necessary, limiting its portability

A straightforward message-driven implementation of dense LU factorization exhibits this problem because it carries a mix of latency-sensitive messages on the active panel, and mostly latency-insensitive work in the trailing submatrix. The former take microseconds to single-digit milliseconds per matrix column, whereas the latter take tens of milliseconds each.

When work on the active panel is available on a processor, it is given priority over all other parts of the factorization process. However, because new active panel work only arrives after the previous one has been completed, the intervening gap between these units gives the processor an opportunity to schedule large grain trailing updates or triangular solves. If such large grains are scheduled, the processor's participation in the next unit of active panel work is delayed, affecting all the processors involved in the panel factorization. This considerably slows down this class of work which lies on the critical path. With sufficient delays, processors will exhaust their backlog of trailing updates before the current panel is factorized and data for the next batch becomes available.

The synchronous execution structure of HPL prevents this problem from arising. Based on lookahead depth, all processors know what work they are expected to complete before participating in the next row/column broadcast or active panel factorization. Until recent hardware generations, this workload was naturally balanced, since the operation counts and processor speeds

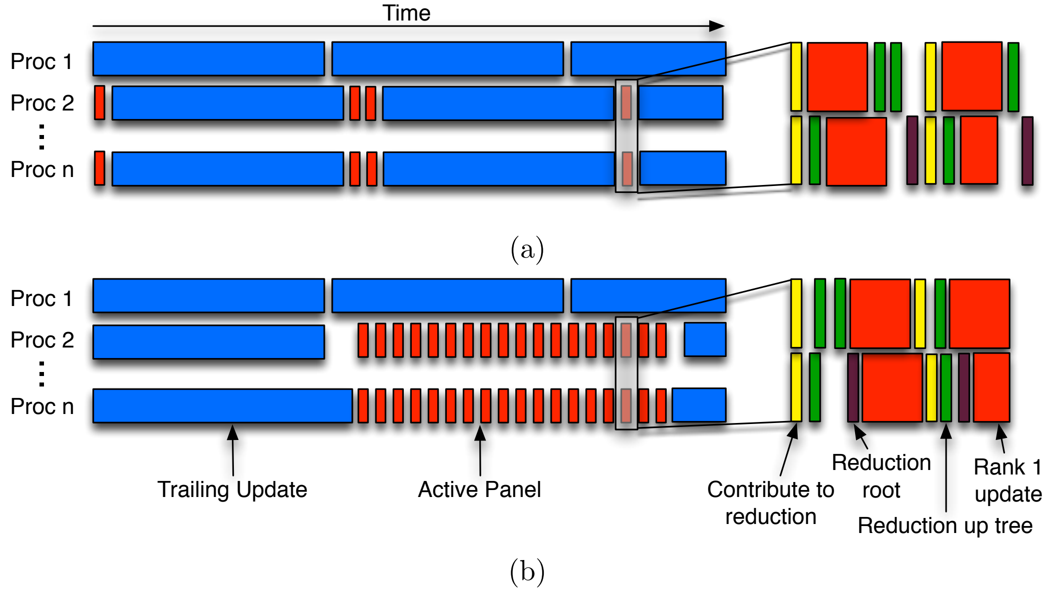


Figure 6.4: Two different time progressions of dense LU: 6.4a displays execution with interleaving of various grain sizes; 6.4b shows execution with isolation. If the smaller grains are interleaved with larger grains, the critical path is prolonged.

were uniform. With variable processors now becoming ubiquitous [19], that can no longer be relied upon to maintain high utilization. Thus, this synchronization will cause faster processors to wait when they could have done more work without causing a delay.

A possible method to decrease this interference is to separate work units into *exclusive scheduling classes*. During execution, the scheduler is set to some exclusive scheduling class. Work units of lower classes in the local queue will be held back in favor of higher class work units. Such stratification of work units allows the scheduler to selectively choose only the work units that are suitable for execution, depending on the currently active scheduling class. The active scheduling class is determined by the application; it instructs the scheduler to transition to a different scheduling class when appropriate.

This methodology has the advantage of maintaining the desired variation in grain sizes while using a general scheduling methodology to solve the problem, thereby improving performance. Moreover, the intricacies of using application-specific polling or interruption/preemption can be avoided by segmenting work into scheduling classes.

To achieve high overall performance in dense LU, we simulate a scheduling-

class scheme on top of the Charm++ runtime's priority-based scheduler. When work of one class is selected for execution on a processor, other work in lower scheduling classes is held back to avoid introducing unnecessary latency. This technique is analogous to scheduling classes in realtime systems and microprocessor interrupt levels: the delay or preemption of the latency-sensitive factorization is prevented by temporarily disabling execution of lower-class coarse grain work. Figure 6.4 shows two different possible executions, both with and without isolation using exclusive scheduling classes enabled.

6.2.6 Isolation of Active Panel

The most apparent class distinction in dense LU is between the active panel factorization and the bulk work (triangular solves and trailing updates). This separation is enforced by keeping a processor-local counter of the blocks currently participating in the active panel. When this counter is non-zero, bulk work is not enqueued into the runtime scheduler's queue. Instead, it is placed into an application-level queue, to be re-scheduled when the active panel completes. Bulk work units that are waiting in the runtime's queue are removed and placed in the same application-level queue. To maintain this counter, each block on the active panel increments this counter after contributing to the first column's pivot reduction and receiving the broadcast that results.[2] They decrement the counter when active panel work is complete.

The benefits of isolating the active panel from the bulk work can be seen in figure 6.5. As the application weak scales with the active panel isolated, performance remains consistently high. However, without isolation, performance drops sharply.

6.2.7 Isolation of Triangular Solves

Among the larger work units, there are two different tasks: triangular solves on U blocks and trailing updates. Because triangular solves generate additional concurrent work, we generally prefer to perform triangular solves

[2] The increment must wait for the first column to finish to prevent deadlock: some other block on a processor may need to perform a trailing update before it can participate in the active panel.

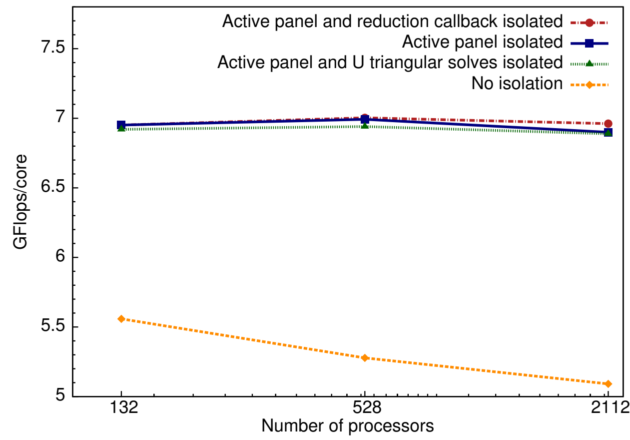


Figure 6.5: Performance effects of enforcing various exclusive scheduling classes on XT5 with weak scaling.

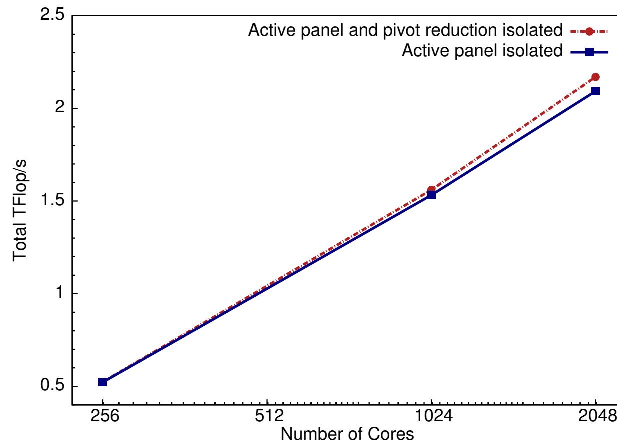


Figure 6.6: Performance effects of enforcing the pivot reduction exclusive scheduling class on BG/P with strong scaling from 256 to 2048 processors. As we scale, isolation has a greater impact on performance.

before trailing updates. Thus, we have also considered delaying trailing updates when the data to perform triangular solves is expected to be available. This occurs when an active panel is completed, and the diagonal block and all pivoting instructions have been broadcast. As figure 6.5 shows, this separation is actually slightly detrimental to performance. Performance degrades in this configuration because each triangular solve depends on pivot data from one or more blocks in the trailing submatrix below it, some of which may not have completed their updates for the previous step. Thus, the triangular solves wait longer than the execution time of several trailing updates before becoming ready to execute, and the processor idles.

Instead of a class separation, simple prioritization of ready-to-execute triangular solves ahead of any trailing updates provides the best performance. A more elaborate prioritization scheme might still prefer some trailing updates, such as to blocks that are in the next active panel, over triangular solves, especially those far to the right in the matrix.

6.2.8 Isolation of Asynchronous Reductions

The final work class distinction considered in this paper lies within each active panel process. Our steps for the factorization of each column of the matrix include: pivot identification via asynchronous reduction amongst all the participants in the active panel factorization; broadcast of a fragment of the pivot row to all participants; and a rank-1 update of the remaining unfactorized sub-blocks that are on the active panel. Performance gains were realized by splitting the rank-1 update into two separate updates: one for the immediate next matrix column and the other for the remaining sub-block. This allows earlier participation in the next pivot identification which is critical to progress and overlaps this communication with the rank-1 update computations.

The runtime performs the pivot reductions by constructing a spanning tree amongst the participant processors. These reduction operations along the spanning tree are fine-grain, while the rank-1 updates are large in comparison. When these rank-1 updates were scheduled on a processor before the reduction moved past it along the spanning tree, the overall progress was impaired by the delay in the reduction (inset of figure 6.4a). Thus, we place the reductions in a higher class than the rank-1 updates.

We modified Charm++'s reduction mechanism to signal a callback on each processor after a reduction has propagated past that processor's position in the tree. This signals a transition out of the pivot identification work class, and pending rank-1 updates can then be executed (inset of figure 6.4b). Figure 6.6 shows that this yields an increasing performance improvement as we strong-scale. This gulf appears because strong scaling LU leads to a growing proportion of execution time spent in active panel factorizations.

We believe such a notification mechanism can be a general technique for scheduling around asynchronous sender-driven collectives. This directly aids in transitioning between exclusive scheduling classes.

6.2.9 Synchrony Amidst Asynchrony

By partitioning work into exclusive scheduling classes, we demonstrate that ideally highly synchronous workflows can run without interference from large-grain latency-insensitive asynchronous computation. Moreover, by placing an asynchronous collective in a separate scheduling class, fine-grained critical path work runs unaffected by larger grains, which are deferred by the scheduler's transition into a higher scheduling class. For dense LU, we describe an application-specific implementation of such a scheme and show that it substantially improves performance.

Our methodology attempts to increase the efficiency of synchronous operations in an asynchronous programming model. This suggests that for some parallel algorithms, purely asynchronous programming models may have disadvantages. For instance, if highly synchronous work is on the critical path, ensuring that it executes early, uninterrupted by other work, may be essential to obtaining high performance. Hence, it seems that while asynchrony may be required to effectively program on the next generation of supercomputers, methodologies and runtime tools that increase the efficiency of synchronous operations, allowing them to execute without interruption, will also be necessary for the programming models of the future.

Essentially, the techniques described here recognizes and accounts for the fundamentally synchronization-heavy nature of active-panel factorization with columb-by-column partial pivoting. By having the runtime system build a fence around phases of execution involving intense synchronization, we allow it to pro-actively mitigate some of the downsides of that synchronization.

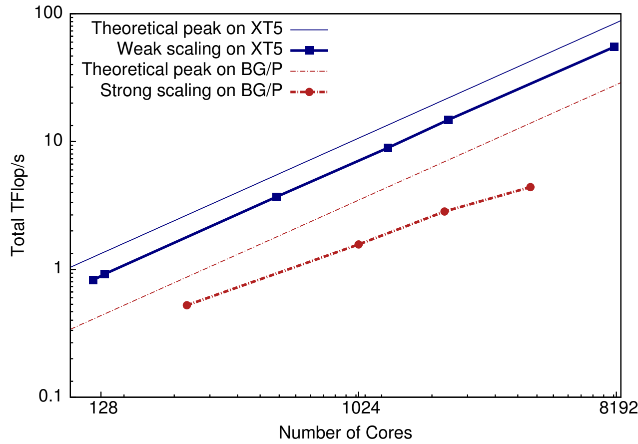


Figure 6.7: Weak scaling (memory usage of matrix is constant around 75%) from 120 to 8064 processors on Jaguar, a Cray XT5 machine with 12 cores per node. Strong scaling ($n = 96,000$) from 256 to 4096 processors on Intrepid, an IBM BG/P machine with 4 cores per node.

If the concern to be managed around that synchronization were not primarily the internal interference of coarser-grained work, other responses would have been necessary. For instance, system noise or inter-processor performance variation [19] would cause some processors to make their reduction contributions later than others. Since the active panel factorization process on each node is memory-bandwidth bound, threads on some cores in each node would not participate, and continue to work on more cache-friendly trailing updates instead [35]. Thus, in a case with extrinsic timing variation, it may be more appropriate for threads not participating in the active panel factorization to sleep for that interval. Sleeping would leave ample power available for the active panel cores to run at top speed, and potentially allow interfering processes to be scheduled there and avoid creating interruptions.

Arch.	Cores	N	b	P	Q	r	s	Peak
XT5	120	126K	504	24	5	6	5	67
XT5	132	132K	500	22	6	11	6	67.1
XT5	528	264K	500	44	12	22	6	67.4
XT5	1296	420K	500	72	18	24	3	66.2
XT5	2112	528K	500	88	24	44	3	67.4
XT5	8064	1048K	500	192	42	64	3	65.7
BG/P	256	96K	300	64	4	8	3	60.2
BG/P	1024	96K	300	128	8	8	3	45
BG/P	2048	96K	150	128	16	16	2	40.7
BG/P	4096	96K	150	256	16	16	2	31.6

Table 6.1: Highest performing runs plotted on Figure 6.7.

Block size	450	500	504	525	560	700
DGEMM (%)	78.2	81.9	82.3	81.8	81.6	83.6
LU (%)	65.5	66.6	67.0	66.5	65.5	65.0

Table 6.2: Percent of peak achieved by DGEMM and LU factorization on Cray XT5 with 120 cores and $n = 126000$.

Library	Peak	Cores	n	Arch.
UPC [49]	76.6	512	229K	XT3
DPLASMA [53]	58.3	3072	454K	XT5
ScalaPack [52]	59	3072	454K	XT5
HPCC [54] HPL	65.8	224220	3936K	XT5
Jaguar top500 [27]	75.5	224162	5474K	XT5
CharmLU	67.4	2112	528K	XT5

Table 6.3: Percent of peak achieved by various linear algebra libraries. CharmLU is the implementation presented in this paper.

6.2.10 Performance

DGEMM Performance

The peak performance obtained by an LU solver is bounded by the performance of the DGEMM implementation that it invokes. The performance of a DGEMM often varies with the size of the matrix on which it operates; a larger DGEMM generally executes more efficiently than a smaller one. The tradeoff between coarse grain sizes that aid in higher DGEMM efficiency and fine grain sizes that allow greater overlap of communication and computation is shown in Table 6.2.

Table 6.3 compares our implementation with other dense LU solvers. Note that the architectures and matrix sizes vary, so it is difficult to provide an exact comparison, but the values imply that our implementation is competitive.

Scaling

To demonstrate the scalability of the dense LU solver described in this chapter, it was weak scaled to 8064 processors on the Jaguar Cray XT5 using approximately 75% of memory. This represents about 530 blocks of 500×500 doubles for each processor. The solver obtains over 67% of peak on XT5 machines. Additionally, we also demonstrate the capabilities of the solver in the strong scaling regime up to 2048 processors on the Intrepid IBM Blue Gene/P, achieving over 50% parallel efficiency. Figure 6.7 and Table 6.1 show both sets of results.

6.3 Related Work

Communication-avoiding algorithms have been developed to reduce the message count and data volumes involved in various computations. In the process of reaching toward communication lower bounds on solutions to the various problems, some of these new algorithms also drastically change the synchronization structure relative to traditional methods.

One clear example of this is seen in the 'tournament pivoting' technique in communication-avoiding LU factorization [55, 56]. Conventional partial

pivoting requires a reduction for each column of the matrix to be factored, and that reduction is dependent on results of the reduction immediately preceding it. As illustrated in Section 6.2.6, this tight synchronization makes it very sensitive to interference. Tournament pivoting uses only a single large-block reduction for each block of k columns, thus synchronizing k times less often. This can be seen as an instance of pattern 3.2, doing more with each collective to make them less frequent.

6.4 Summary

Scalable implementations of dense LU factorization have generally relied heavily on synchronizing collectives to coordinate availability of limited network and memory resources. The work described above shows that it is possible to unload this coordination to other mechanisms. Thus, the synchronizing operations can be replaced with narrower communication that's responsible only for the payload data and not any other implicit state.

Chapter 7

Tree-Structured Adaptive Mesh Refinement

List of Patterns Illustrated:

3.4 Separate communication from coordination (§ 7.2.2)

3.6 Replace synchronizing collectives with p2p messages that achieve the desired effect (§ 7.4)

3.7 Semantic object naming (§ 7.2.4)

Tree-structured adaptive mesh refinement (AMR) is implemented in simulation frameworks such as Flash [57] and Enzo-P/Cello [58]. It offers a highly-regular mesh structure, compared to the relative freedom offered by patch-structured AMR. The difference in structural regularity between the two styles offers a tradeoff. On the one hand, patch-based meshes can offer high efficiency of deploying fine-resolution grid points to regions of the problem domain that truly demand them for an accurate solution, keeping the direct operation counts low. On the other hand, tree-based meshes offer lower execution overhead and much simpler implementation.

This chapter describes work on remeshing algorithms for a tree-structured AMR mini-application in Charm++. This mini-app was originally implemented by Akhil Langer, and optimized, scaled, and benchmarked by me and Jonathan Lifflander [32][1]. It replaces global collectives seen in the remeshing algorithms of other AMR frameworks with point-to-point messages and a pair of termination detection operations to determine when algorithm phases have been completed across all objects. Later work built upon improvements

[1]The text and figures of Sections 7.1–7.3 are adapted from the cited paper with permission. ©2012 IEEE.

in CHARM++ and SDAG to reduce that to one termination detection during remeshing [59]. The present work, discussed starting in Section 7.4, describes how to bring that to a logical conclusion, in which synchronization is entirely localized among nearby objects, and no global termination detection is needed.

This design evolution applies several of the patterns described in Chapter 3. By using bitvector coordinates of the blocks as semantic names, we avoid the need for the mesh construction algorithm to number blocks or communicate their location. This means that mesh refinement only needs to communicate each blocks' local conditions to its neighbors for all blocks to determine the future structure of their neighborhood. The cascading nature of these communications meant that some mechanism was needed to detect convergence, but the synchronization that implies could be separated from the individual messages. Finally, by defining a purely local convergence mechanism, the convergence-signaling synchronization could be eliminated.

7.1 Related Work

We use a block-structured AMR scheme that has similar refinement structure to [60, 61]. PARAMESH [62], Burstedde et al [63] implement a design that requires each process to store the mesh structure, which requires $O(p)$ memory per process and $O(\log p)$ time per lookup of a neighboring leaf block. Burstedde et al. [64] describe a distributed AMR strategy that uses a parallel prioritized ripple propagation algorithm for mesh restructuring, causing the number of communication rounds to grow with the number of refinement levels. Each round involves message exchanges between processors and an equivalent of a system reduction to indicate the beginning of the balancing at the next level of refinement. This approach is also limited because it does not allow coarsening of sibling quadrants that are distributed across separate processors. The SAMRAI framework [65] incurs significant overhead creating a 'communication schedule' during remeshing that also involves multiple collective communication rounds.

Bangerth at el. [66] describe a scalable design for the parts of a parallel AMR calculation *other* than the mesh generation/restructuring effort. They explicitly delegate maintenance and distribution of the mesh structure to an 'oracle' which must be able to answer queries akin to what the algorithms

described in this paper provide, for which they give `p4est` [67] as an example. They describe a hierarchical mesh point identification scheme as a requirement of said oracles that matches the one we describe for mesh blocks. Our mapping scheme explicitly uses these identifiers to generate a roughly balanced mapping of mesh blocks to processors.

Our bitvector indexing and mapping scheme pushes the simplicity benefits of the forest-of-trees decomposition used by `p4est` [67, 68] into each tree, creating a completely uniform representation of element identity. Rather than splitting the elements along a Peano-Hilbert space-filling curve (or Z-order curve) [69] to load balance, which requires expensive collective communication and synchronization and imposes substantial memory usage requirements (at least one entry per rank on every rank), we use these identifiers to generate a mapping that provides a large degree of natural balance. Where that is insufficient, the Charm++ [47] runtime system underlying our implementation provides efficient object migration, hash-based lookup of migrated objects, and application-transparent forwarding and new-location notification when migrations occur. These support a wide variety of dynamic load-balancing and mapping schemes which can be applied in concert with our algorithms.

Load balancing has been studied extensively for AMR, ranging from using graph partitioning techniques [70, 71, 72], to other more AMR-specific methods [73]. We focus on building a locally computable mapping strategy for mesh blocks that localizes work and evenly distributes it without any synchronization or periodic redistribution of work.

7.2 Algorithm Description

Traditional AMR algorithms are designed in terms of processors that contain many blocks. In contrast, blocks in our design are first-class entities that operate as if each resides on its own virtual processor (§ 7.2.1). As the computation proceeds, refinement and coarsening operations expand and contract the collection of blocks. The refinement decisions are made locally by each block and then are propagated to affected neighboring blocks recursively to keep blocks within one refinement level of their neighbors (§ 7.2.2). Because remeshing is constrained to occur at discrete points in simulation time, we use a scalable termination detection mechanism (§ 7.2.3) to globally deter-

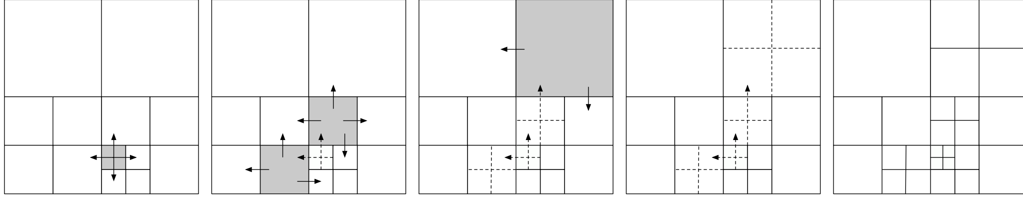


Figure 7.1: Propagation of refinement decision messages, based on local error criteria and near-neighbor communication. Shaded blocks have concluded that they must refine, and send messages (solid arrows) accordingly (a-c). The path and effects of this rippling message chain are shown by dashed lines and arrows (b-d). Eventually, all the blocks reach a consensus state (e).

mine when all refinement decisions have been finalized. Besides this, blocks synchronize with each other only by the communication of boundary cells, and otherwise execute completely asynchronously.

Each block is addressed by its location in the refinement tree. The underlying runtime system provides direct communication between arbitrary blocks. We describe a mapping from block addresses to processors that provides reasonable load balance and locality under the dynamic workload evolution that AMR presents (§ 7.2.4). This avoids the need for explicitly redistributing the load during the computation.

7.2.1 Distributed Parallel Objects

To obtain high performance, AMR implementations typically partition work into k blocks for p processor cores, where $k > p$. Existing algorithms and implementations treat processors as fundamental first-class entities that explicitly manage $\frac{k}{p}$ blocks. However, the computation is local to each block or between neighboring blocks, so processor-centricity obscures the fundamental character. Our design treats each block as the basic element of a medium-grained parallel execution. Each block is expressed as an uniquely addressable object within a parallel collection that encapsulates data and methods. By taking a dynamic collection of blocks as our fundamental entity, we enable straightforward expression of the new algorithms described later in this section.

Each block in our design is a virtual endpoint of communication. Instead of addressing messages to a system rank, each message is addressed to an object that is managed by the runtime. The runtime ensures that

each message is delivered to the appropriate processor where the object currently resides. Directly addressing blocks requires that they have distinct, processor-independent names that can be efficiently mapped (and possibly remapped) to a host processor. This requirement turns out to lead to other algorithmic improvements relative to existing implementations (§ 7.2.4) and it takes only $O(N/P)$ memory per process where N is the total number of blocks and P is the total number of processes.

The block-centric formulation of our design offers several algorithmic advantages: firstly, the updates on a block's zones can begin as soon as it receives the necessary halo data from that block's neighbors. Secondly, the computation of each block's update steps can overlap with communication for all the other blocks on the same processor. Finally, a great deal of implementation complexity is spared in the application code.

Our novel algorithm relies on efficient, asynchronous messages between block objects. Each block can send a message to another block by remotely invoking a method on it with some associated data. The data is sent as a message to the appropriate processor by the runtime and executed in turn on the targeted block. Messages can be sent to currently nonexistent objects: because the block-to-processor mapping is deterministic given the block's unique address, messages can be simply buffered by the runtime on the processor where the block will be dynamically constructed. This behavior allows us to limit the amount of synchronization that is required in our algorithm.

7.2.2 Mesh Restructuring Decision Algorithm

During the course of execution, the simulated domain is expected to evolve such that some zones require finer resolution to obtain accurate results, while other zones can be safely simulated more coarsely. Like other AMR implementations, we currently make these adjustments periodically between steps of the simulation. The defining features of our algorithm are that it uses only point-to-point messages between spatially-neighboring blocks to communicate remeshing decisions, and that it synchronizes through a lightweight termination detection mechanism (§ 7.2.3) only to determine when all blocks have reached consensus on their remeshing decisions.

When a block reaches the point in simulation time at which mesh resolution is to be reconsidered, it must decide whether it will *refine*, *stay* at its current

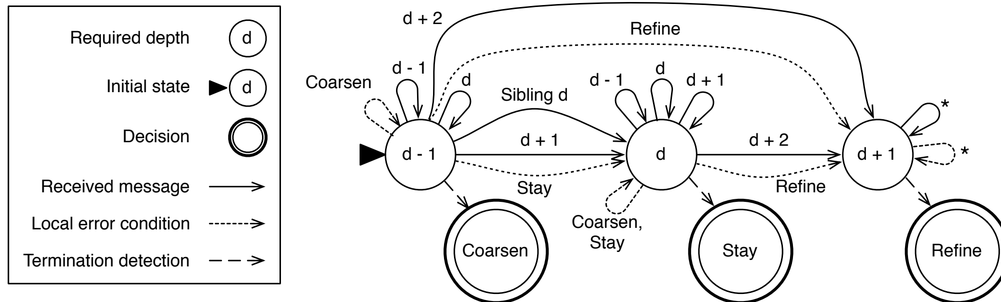


Figure 7.2: The finite state machine describing each block's decision process during the original mesh restructuring algorithm. A block's decision can change as a result of receiving messages from neighbors or siblings and as a result of evaluating its local error condition. When termination is detected all decisions are finalized.

resolution, or *coarsen* before subsequent time steps. Each block can assume as a precondition that all of its neighbors and siblings (i.e. its communication partners) start off at a refinement depth that differs from its own by at most one. To minimize the overall computational load, every block should be coarsened as much as possible. The requirement for accuracy means that any block's decision to refine or maintain its resolution will constrain its neighbors and siblings to maintain or increase their own resolution.

Figure 7.1 illustrates an example of how this process might proceed. Part (a) shows that a single block decides to refine (shown as shaded) based on its local error estimate and all the other blocks locally decide to maintain their current resolution. The shaded block sends messages (drawn as solid arrows) to its communication partners indicating that it intends to increase its refinement depth, and they must adjust accordingly to keep the invariant of at most one level of difference between neighbors. Parts (b) and (c) depict how this decision's effect ripple out to nearby blocks, with affected blocks downstream (those whose resolution changes) shaded, and the path of affected blocks shown by dashed lines and arrows.

The overall algorithm that each block executes can be described by the finite state machine illustrated in Figure 7.2. Each d state represents a possible refinement depth for the block relative to its current depth. All of the blocks move from a d state to a decision state when termination detection indicates that they have reached consensus. The primary transitions from

one state to another are driven by the receipt of messages from neighbors and siblings indicating their intended depth. Each time a block moves from one d state to another, it sends messages to each of its communication partners indicating the state that it has entered, possibly causing them to transition and communicate as well. Although blocks will try to coarsen themselves by default, any stimulus (message or local error condition) indicating a need for higher resolution will take precedence. This can be seen in the state machine's monotonic flow from coarser states toward more refined states.

Each block's machine is initialized to a state that would have it coarsen (indicated by the large triangle) as soon as its execution passes the previous cycle of remeshing decision-making. Because the blocks do not execute in lock step with one another, a block may receive messages that advance its state machine to $d+1$ and thereby constrain its decision even before it has finished timestepping to the remeshing point. This allows for a small optimization in which a block need not evaluate its local error condition if its neighbors' decisions dictate that it must refine. If a block does finish timestepping while in a state other than $d + 1$, it evaluates its local error condition and follows the appropriate transition as indicated by the dotted arrows.

Note that there are no transitions that move into the $d - 1$ state from another state. As a result, no block will ever send a message indicating its own intention to coarsen, and no block will receive a message indicating that a less-refined neighbor wishes to change to level $d - 2$. Thus, there are no $d - 2$ transitions in the state machine.

After all the decisions are finalized, blocks are created or destroyed as a result. A block that has decided to coarsen (in concert with its siblings) sends its downsampled data to its parent block and then destroys itself. A block that has decided to refine constructs new child blocks and sends the corresponding portion of its data to each of them.

7.2.3 Termination Detection

Because refinement decisions are determined and further propagated based on distributed mesh data, detecting the global property of consensus requires termination detection. Termination is the state when no messages are in flight and all processes are idle. Many different varieties of algorithms for detecting termination are well-established in the literature [74].

For this application, we use a wave-based four-counter termination detection algorithm that propagates waves of total send and receive message counts up and down a spanning tree that includes all the processors. When the send and receive message counts for two consecutive waves are identical, termination is detected [24]. Because waves are only propagated when a processor is otherwise idle, two identical consecutive counts indicate that no messages are in flight that could spawn more work. Only propagating waves when a processor is otherwise idle heavily reduces the number of waves that are ever started, because any busy processor will block the progression up the spanning tree. For AMR, the delay time between the last block reaching its decision and termination detection is low (empirical results are in § 7.3.3).

7.2.4 Block-to-processor Mapping and Load Balancing

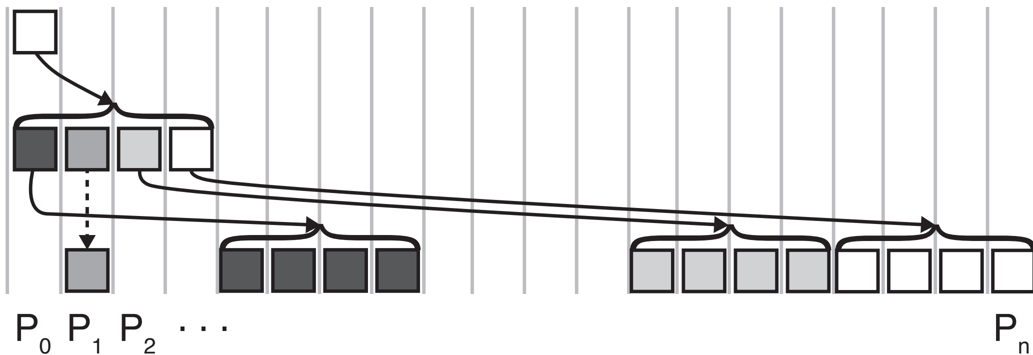
In AMR, the collection of objects expands and contracts unpredictably over time, causing dynamic load imbalances to arise. Synchronized redistribution of blocks is expensive because of the high frequency of growth and shrinkage. Hence, it is important to consider locality and load balance when initially placing new objects. Because we seek to limit synchronization, the seeding function must be locally computable on any processor and deterministic, so that addressing a block is inexpensive and possible before block construction.

Each block's address is a bit-vector b that represents its location in the distributed refinement tree. A block of depth d in a quad-tree (oct-tree, respectively) will require an address of $2d$ ($3d$) bits, in which each pair (triplet) of bits $b_{2d+1}b_{2d}$ maps the block to a sector of the tree at depth d relative to its parent. In other words, it describes the path taken by a recursive traversal through the tree from a nominal root to the block in question. The function we define takes this sequence of bits and maps it to a host processor.

The AMR computation initially starts with a collection of blocks at a constant depth c. Because block refinements tend to be spatially correlated, we initially seed the set of blocks with a uniform random distribution (using an inexpensive hashing function) over the set of processors. The mapping function takes the first c bits, applies the hash, and uses this value as the base processor for this block and all its descendents. The remaining $d-c$ bits are then used as an integer offset from the base processor. The algorithm is detailed in Figure 7.3a.

Input: *blockAddress*, a `bitvector` of length $2d$
nproc, the total number of processors
prime = `0x9e37fffffffc0001`
Output: The processor *proc* it is mapped to
let `bitvector` $base = blockAddress[1:c]$;
let `int` $basePE = (prime * base)$ `>>` $(64 - \lg(nproc))$;
let `bitvector` $remainder = blockAddress[c+1:2d]$;
return $(basePE + remainder)$ % $nproc$

(a) Mapping algorithm



(b) Mapping visualization: in the example shown, the block on P_1 chooses not to refine.

Figure 7.3: The description and visualization of our block-to-processor mapping algorithm, which maps all the descendents from a base block to distinct processors (until they wrap around).

By treating the $d - c$ bits as an integer offset from the base processor, all the descendents of a base block will be mapped to different processors until this offset wraps around. Figure 7.3b visualizes this effect.

7.3 Experimental Results

To empirically test our AMR remeshing algorithm, we benchmark a finite-difference simulation of advection, described by the following hyperbolic partial differential equation:

$$\frac{\partial u}{\partial t} + v\nabla u = 0 \tag{7.1}$$

The advection equation is common in chemistry and describes the advection of a tracer along with the fluid. The density (or concentration) u is the conserved quantity with a bulk motion speed v. In our simulation, v is held constant. We solve the advection equation using a first-order upwind method in two-dimensional space. Although our algorithm is applied to a first-order scheme, our AMR framework can easily be adapted to higher-order multi-dimensional schemes and other hyperbolic problems.

We initialize the simulation with a circular region of density $u = 2$, ambient density $u = 1$, and bulk velocity $v = 1$, with periodic boundary conditions. The error is estimated using the second derivative of the density u, as described by Löhner [75].

7.3.1 Experimental Setup

The experiments were performed on two systems: Cray XK6 'Titan' and IBM Blue Gene/Q 'Vesta'. Each node of Titan consists of one sixteen-core 2.2GHz AMD 'Bulldozer' processor and 32GB DDR3 memory. Only the CPU part of Titan was used for our runs, with no GPU acceleration. Nodes of Titan are connected by the Gemini interconnection network with a 9.8GB/s peak bandwidth per Gemini chip. Our experiments ran with 16 ranks on each node of Titan. Each node of Vesta consists of one 1.6 GHz PowerPC A2 processor with 16 application cores supporting 4-way simultaneous multithreading and 16 GB DDR3 memory. Our experiments ran with 32 ranks on each node of Vesta, using 2-way SMT per core.

Our code uses the Charm++ runtime system [47], which supports dynamic collections of parallel objects in the form of *chare arrays* [34]. We used the

Gemini machine layer in Charm++ for Cray XK6 and the PAMI (Parallel Active Messaging Interface) machine layer for BG/Q. Our code was compiled with the GNU compiler suite version 4.6.2 on Titan and version 4.4.6 for Vesta (Blue Gene/Q release BGQ-V1R1M1-120628).

7.3.2 Overall Performance and Scalability

Our benchmark application performs relatively little calculation in each time step, and remeshes in a cycle of every two timesteps. We chose this configuration in order to both highlight and stress test the remeshing algorithm.

Our benchmark results can be seen in Figure 7.4, which plots the time taken for each cycle over the course of a run. As one would expect, step times scale down with processor count. The upward trend in cycle time seen on the smaller runs can be attributed to slowly-growing load imbalance as the highest-resolution zones shift across the problem domain. The smaller runs are more severely impacted by this because of the effect of blocks descended from different roots being mapped to overlapping processors. At larger scales, where the root blocks are more widely spread over the whole system, this overlap effect diminishes. An explicit dynamic load balancing mechanism could be run periodically to mitigate this effect. However, in the current work, we have found this to not provide sufficient benefits.

An overall view of our code's strong scaling behavior can be seen in Figure 7.4. We depict strong scaling curves, each representing a dynamic range of refinement. A minimum depth of 4 represents a coarsest mesh dimension of 256^2, which quadruples to 512^2 and 1024^2 at depths 5 and 6 respectively. The black lines indicate ideal scaling on each machine relative to the performance of a whole single node. The ideal scaling lines for Blue Gene/Q are drawn through the 32-core point to reflect the higher-performance 2-way symmetric multi-threaded mode in which that and all larger runs were performed.

When scaling from 16 ranks on Cray XK6 with a minimum depth of 5 and maximum depth of 11 (as shown in Figure 7.4c), we are able to achieve 80% parallel efficiency up to 1024 ranks (up to 30 ms/cycle), and 64% parallel efficiency at 2048 ranks (up to 19 ms/cycle). Note that we run with a wide range of refinement levels and increasing dynamic range does not reduce efficiency. Instead, for a given minimum depth, increasing the maximum depth actually increases efficiency: on 2048 ranks our code attains 36% parallel effi-

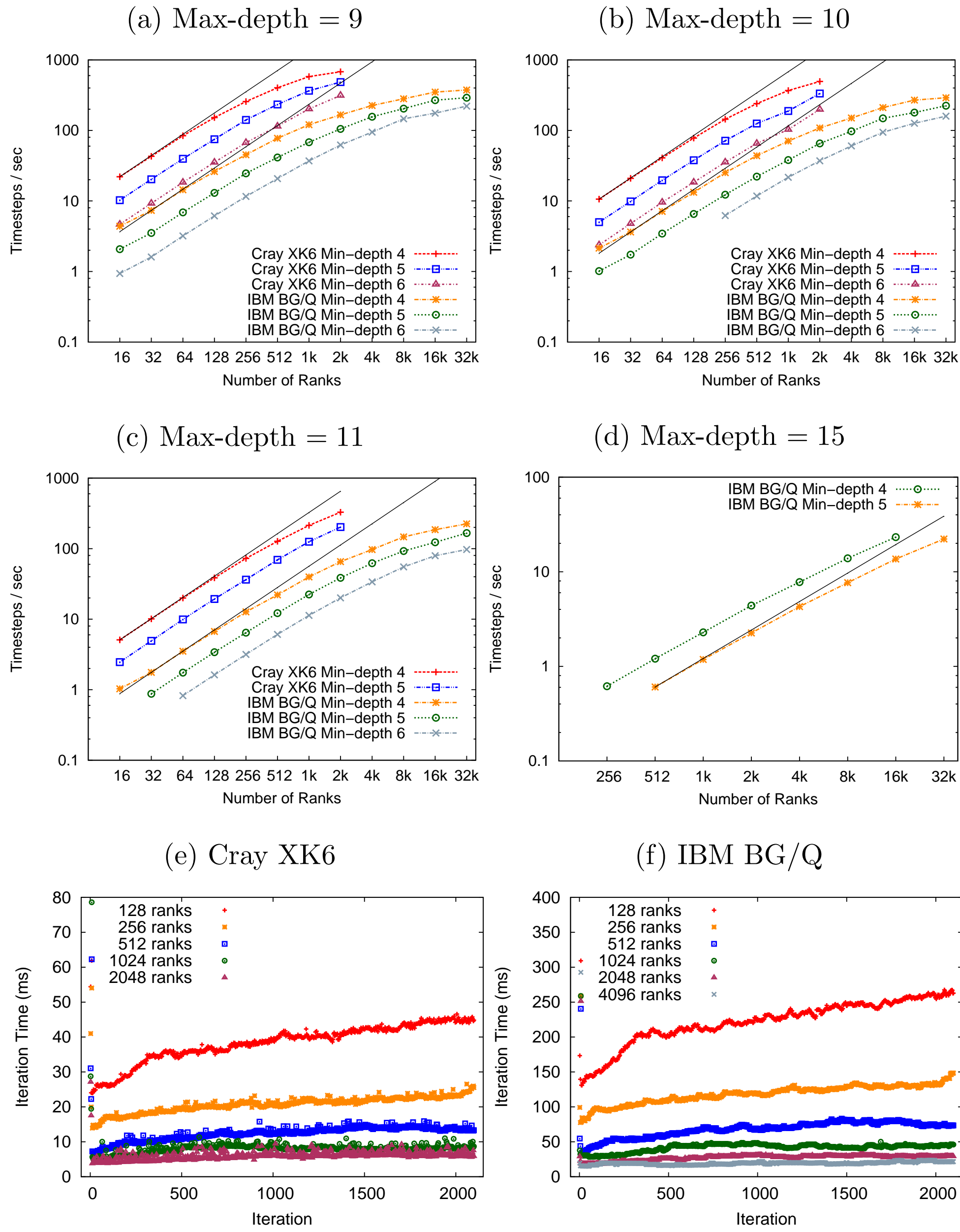


Figure 7.4: Rows 1 & 2: timesteps per second strong scaling on Cray XK6 and IBM BG/Q with various minimum depths; row 3: the duration in milliseconds for each cycle with a max-depth = 10 (each composed of two timesteps and a remeshing operation).

ciency with a depth ranging from 5–9 and increases to 64% parallel efficiency with a depth ranging from 5–11. Although our remeshing scheme requires deeper propagation with a wider depth range, it does not dominate and the increase in work leads to an increase in efficiency.

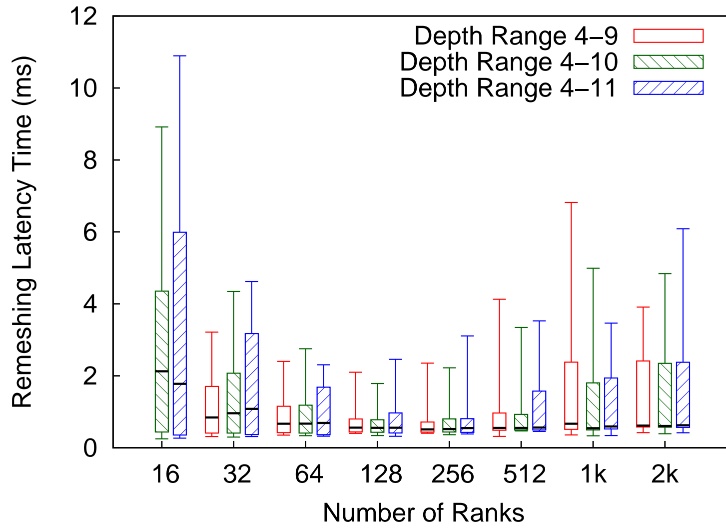
On IBM BG/Q, scaling from 64 ranks to 2048 with a depth range of 6–11, our code achieves 76% parallel efficiency (up to 182 ms/cycle), and when it's pushed to the limit of machine size to 32768 ranks (one rack of BG/Q at SMT 2), it attains 23% parallel efficiency (up to 28 ms/cycle). Upon scaling from 512 to 32k ranks and allowing the depth range to vary from 5–15, we get much higher efficiencies of 99%, 95%, 65%, 55% at 2k, 8k, 16k and 32k ranks, respectively (Figure 7.4d).

7.3.3 Remeshing Performance

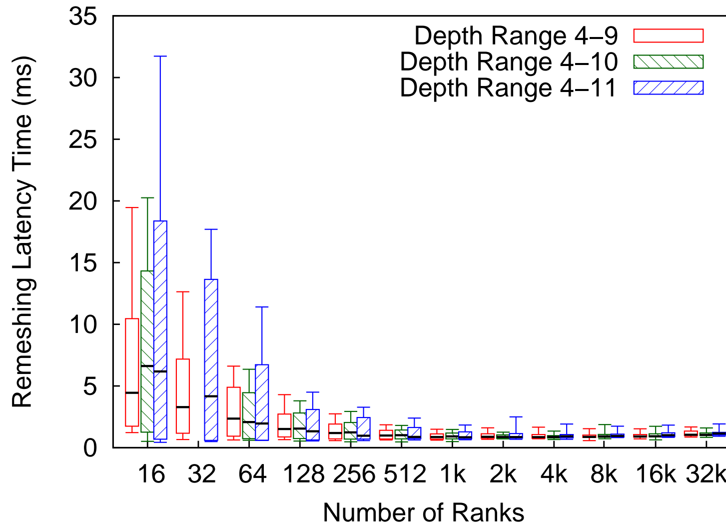
In Figure 7.5, we graph the distribution of remeshing latencies, that is the time interval between the last processor beginning remeshing and the start of the next timestep. This measures the duration spent in remeshing with no overlapping computation. The general trend is that remeshing latency scales down with the number of processors out to a strong scaling limit (about 256 ranks on XK6, and 1024 ranks on BG/Q).

Its performance is actually bounded by two different factors: the communication necessary to make all of the remeshing decisions, and the delay in synchronizing through termination detection after consensus is reached. The first factor dominates at low processor counts, but scales downward as it gets distributed over a larger number of processors. This is easiest to see in Figure 7.5b, where the maximum, 95th percentile, and median times all descend smoothly from 16 ranks to 1024 ranks. To examine the cross-over behavior into the second factor dominance, Figure 7.6 shows the trend in median remeshing times for each set of runs in solid lines, starting from a slightly higher core count to make the slow increase at larger scales apparent.

The wave-based termination detection algorithm as described in § 3.4 has a theoretical upper-bound *delay time* that scales logarithmically with the number of processors, because it uses broadcasts and reductions over a spanning tree. We measure the delay time as the time interval between the last processor processing an application message and it receiving a broadcast indication that consensus has been reached. This duration is graphed in Figure 7.7.

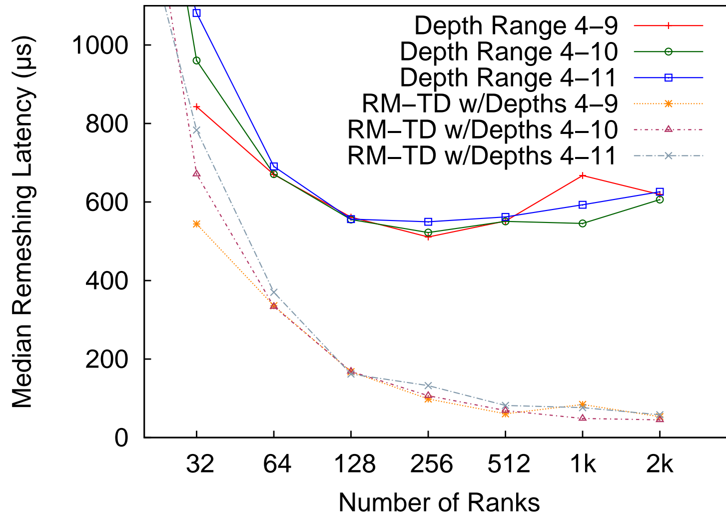


(a) Cray XK6

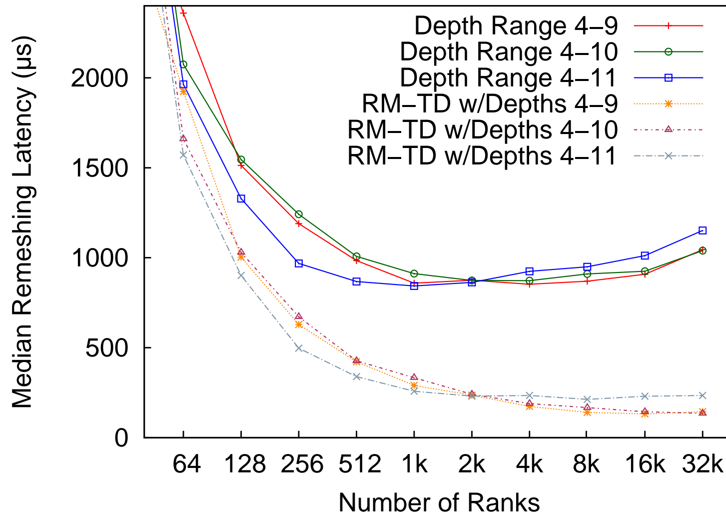


(b) IBM BG/Q

Figure 7.5: The remeshing latency in milliseconds: the non-overlapped delay that remeshing causes in the computation, i.e. the time from the end of non-remeshing work on the last processor to the beginning of the next timestep. The vertical lines stretch between the minimum and maximum values; the box spans between the 5th and 95th percentile; the horizontal line spanning the box indicates the median.

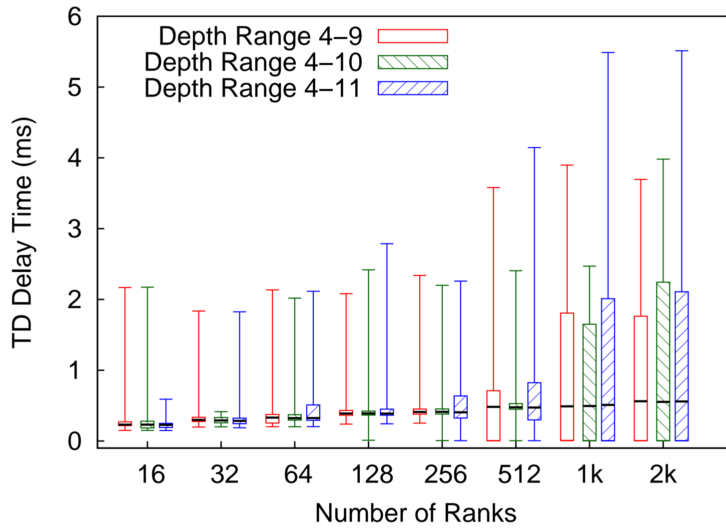


(a) Cray XK6

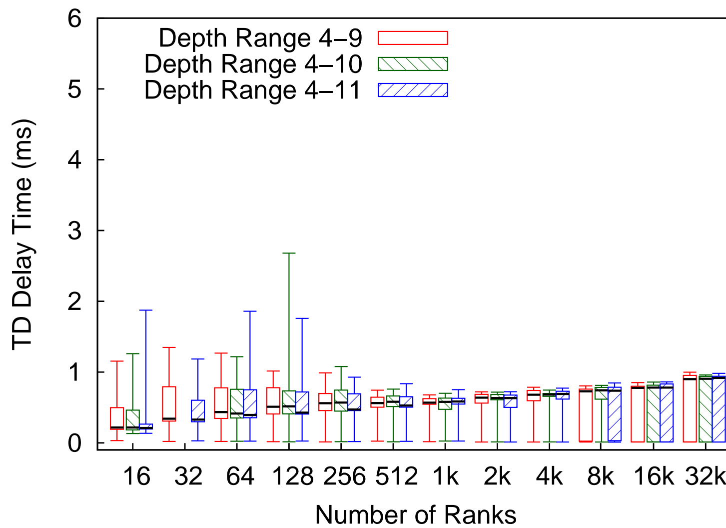


(b) IBM BG/Q

Figure 7.6: The median remeshing latencies (mid-points from Figure 7.5) in microseconds are graphed as the upper three solid lines. The latency scales down with processor count until it becomes synchronization bound by termination detection. The lower three dotted lines represent the difference between the median remeshing latency and median termination detection delay, demonstrating that remeshing time is dominated by termination at larger scales.



(a) Cray XK6



(b) IBM BG/Q

Figure 7.7: The delay time in milliseconds for termination detection. This is measured as the duration between the last work unit executed on any core and the start of the next timestep. The vertical lines stretch between the minimum and maximum values; the box spans between the 5th and 95th percentile; the horizontal line spanning the box indicates the median.

The median remeshing times at larger scale approach a constant offset above the median termination detection delay times as shown by the dotted lines in Figure 7.6. This demonstrates that termination accounts for the slight trend upward in remeshing latency at larger scales.

Overall, these trends show that our remeshing algorithm is not limited by the performance of collective data exchange and has no readily apparent dependence on the depth of the refinement.

7.4 Remeshing With No Global Synchronization

The goal of each member of this collection of algorithms is to incrementally adapt a new or existing mesh to satisfy two constraints. The first constraint is that every point in the problem domain be simulated with at least as much resolution as the application demands for it (e.g. based on accuracy estimates such as bounds on local finite difference truncation error). The second constraint, the 'balance condition,' requires that neighboring units of the mesh not differ by more than one level of refinement. Within those constraints, the algorithms should then not otherwise demand excess resolution at any unit of the mesh structure.

The original algorithm seen in earlier work abstractly kept a finite state machine (shown in figure 7.2) in each tree node representing the adaptation decision that node would take if the decision process ended with no more messages delivered to that object. The possible transitions in that finite state machine are monotonically increasing in resulting mesh resolution – incoming messages from neighbors could keep a node in its current state, or call for more resolution, but never less. Thus, each state could be seen as representing a lower bound on the resolution a node could have when remeshing finished.

My new algorithm (illustrated in figure 7.8) extends the node state and communication to carry upper bounds on necessary resolution as well as lower bounds. These upper bounds are based on each node's local calculation of the resolution needed by points it contains, and knowledge of the state of its neighbors. As in the earlier algorithm, each node sends its state to its neighbors any time that state changes. The intuition of the algorithm is that with suitable logic, we can ensure that each node's lower and upper bounds will eventually converge. When that occurs, the node can conclude that it

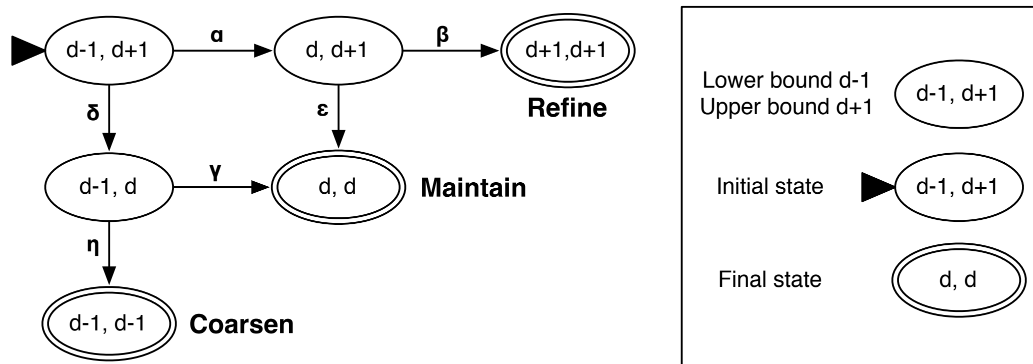


Figure 7.8: The finite state machine describing the new, purely-local converging bounds decision algorithm used by each node in the AMR tree during the remeshing process. The rightward edges increase the node's lower bound based on a local calculation or the increased lower bound of a neighbor (α, β), or increased lower bound of a sibling (α, γ), as in the earlier algorithm. The downward edges decrease the node's upper bound based on a totality of local and surrounding conditions: neighbors and local calculation all releasing a need for greater or current resolution (ε and δ, respectively), and siblings all releasing the need for their current resolution (η). Note that a single event, such as a received message, can spur two successive transitions, analogous to the two-step transitions seen in figure 7.2.

has all the information necessary to decide the level of refinement to provide during the subsequent interval of simulation. Thus, convergence of these bounds takes the place of the global synchronization of termination/quiescence detection used in the the earlier algorithms. This design follows the pattern described in Section 3.6, 'replacing synchronizing collectives with point-to-point messages that achieve the desired effect.'

Ideally, each node would see its bounds converge independently and be able to continue execution immediately afterwards. However, there are two reasons this is not the case. First, even if a node can decide its resolution for the next timestep based on its own bounds, it must still know the outcomes of its neighbors to efficiently communicate ghost cells. The second reason is that a decision for a node to coarsen requires consensus among four or eight 'sibling' nodes to all coarsen and consolidate their data to their less-refined parent in the tree structure. In this case, several nodes will reach a state in which their upper bound says they might hold their current resolution, and their lower bound says nothing has forced them to rule out coarsening. When a full set of siblings see each other in the $(d-1, d)$ state, they can all transition in unison to the state where they conclude that they will coarsen (following the edge labeled η in figure 7.8).

7.4.1 Convergence

To be sure that every node will reach a definitive decision as a result of the remeshing process described, we must show that the algorithm satisfies the following:

> **Theorem:** Every object will eventually reach a converged condition where the lower and upper bounds on its necessary resolution in the subsequent step are equal.

Proving this theorem intuitively rests on the notion that objects that are more refined than any others around them create their own destiny – no other node can drive them to increase their resolution. We define these *maximally refined objects* (MRO) as nodes at level d that have no neighbors at a level of resolution greater than d. We define *maximally refined unconverged objects* (MRUO) as nodes at level d for which all neighbors at level $d+1$ have equal

lower and upper bounds. We will prove that each MRUO converges based on its own error condition and the bounds of its neighbors and siblings, and then inductively prove the theorem by showing that every object eventually becomes an MRUO.

Lemma 1: Every MRUO will eventually have its upper and lower bounds on necessary resolution converge to equality.

If an MRUO at level d has neighbors at level $d+1$, by definition those neighbors have converged. Among those neighbors, suppose at least one of them has decided not to coarsen to level d, but will remain at $d+1$ or refine to $d+2$. That neighbor with maxmimum converged result will drive the MRUO's lower bound to d or $d+1$ to maintain the balance condition. If it is $d+1$, the MRUO has converged to a decision to refine. If it is d, then the MRUO will decide based on its local error condition whether to further increase its lower bound to $d+1$ and refine, or to decrease its upper bound to d and maintain its present resolution.

Consider an MRUO m at level d that has no more-refined neighbors at level $d+1$, or whose more-refined neighbors have all converged on a decision to coarsen down to level d. In the next step, m would be an MRO. Thus, call m a *nascent MRO*. The neighbors of nascent MRO m as such have no influence on m's bounds. Thus, if the local error condition for m says it must increase or maintain its current resolution, that result is definitive, and both bounds can be set accordingly.

If the local error condition of a nascent MRO m will allow it to coarsen, its upper bound can be decreased to d. At this point, m must compare its bounds and state with those of its siblings. If any sibling indicates that it must maintain or refine and thus raises its lower bound to d, then the nascent MRO m raises its own lower bound to d and has converged on a decision to maintain its present resolution. Otherwise, all of the siblings will become apparent as nascent MROs in state $(d-1,\ d)$ whose local error condition will allow coarsening. When this is the case, then the nascent MRO m can decrease its upper bound to $d-1$ in concert with its siblings, and m has thus converged on a decision to coarsen.

Lemma 2: Every object eventually becomes an MRUO.

This can be proven by induction on each object's level of refinement. We can take MROs at global maximum level d_{max} as a base case. MROs have no neighbors at a greater resolution than their own, and so they trivially have no such neighbors with unconverged bounds. Therefore, MROs are MRUOs at the beginning of the remeshing process.

An object m at level d can have neighbors at levels $d-1$, d, or $d+1$. The state of neighbors at levels $d-1$ and d do not affect whether m is an MRUO. Only the neighbors at level $d+1$ are relevant. By induction, we assume that all neighbors at level $d+1$ already became MRUOs. By Lemma 1, all MRUOs eventually converge. Thus, all neighbors of m at level $d+1$ converge, and thus m must eventually become an MRUO.

Proof: By Lemma 2, every object eventually becomes an MRUO, and by Lemma 1, every MRUO eventually has its lower and upper bounds converge to equality. Thus, every object has its lower and upper bounds converge to equality.

7.4.2 Bounded Message Count

We wish to show that this new algorithm does not send excessive messages beyond those that would have been sent in the earlier algorithms. The original algorithm using termination detection guaranteed a bound of $\mathcal{O}(N)$ messages sent as part of each remeshing step. We can see this by noting that each object has at most a constant number of neighbors, that an object sends a message to each of its neighbors when its remeshing state changes, and that each object can experience at most two of these state changes due to the monotonicity of the algorithm's state machine. Essentially the same argument applies to this new algorithm.

In the new algorithm, the lower and upper bounds are initially separated by two steps, and move toward each other but cannot cross. This gives three cases, one for each outcome, that can experience up to two transitions and consequently send at most two sets of messages:

1. Coarsen: Upper bound steps from refine to maintain, and from maintain to coarsen;
2. Maintain: Upper bound steps from refine to maintain, and lower bound steps from coarsen to maintain (or vice versa);
3. Refine: Lower bound steps from coarsen to maintain, and from maintain to refine

There is a degenerate variation on the Refine case, where a level-d object has a more refined level $d+1$ neighbor declare that it is refining. When this happens, the first intermediate step is skipped, and the object's lower bound jumps from coarsen directly to refine. In this case, because the object experienced fewer state transitions, it sends only one set of messages, but with the same cumulative effect on its neighbors and siblings.

7.5 Future Work

Implementation Validation We can validate the newly described global-synchronization-free algorithm by comparison to the results of its predecessor. Specifically, we can extend the code to run both the quiescence-based algorithm and the convergence-based algorithm concurrently. When the quiescence-based algorithm reaches completion, we can then assert that each object has lower and upper bounds that converged to equal both each other and the conclusion of the original algorithm.

Performance Evaluation By benchmarking on systems with and without substantial noise or other variation (e.g. a commodity cluster and a Blue Gene), we can separate some of the hypothetical effects of removing the synchronization imposed by quiescence detection. Thus, we can take any performance difference measured on the quiet system to be indicative purely of the saved cost of the synchronizing operation itself and any rapidly-varying load imbalance. We would then expect to find an additional advantage of the new algorithm over the old on a noisier system to reflect its mitigation of noise as well.

Chapter 8

Desynchronizing and Optimizing the Chombo AMR Framework

List of Patterns Illustrated:

3.3 Send and consume data expected from a collective incrementally (§ 8.1.6)

3.7 Semantic object naming (§ 8.1.3)

3.1 Batch (blocking on / waiting for) collectives (§ 8.2)

3.6 Replace synchronizing collectives with p2p messages that achieve the desired effect (§ 8.2.3)

Chombo is a framework for the construction of structured mesh AMR algorithms and applications. It presents a single-program, multiple-data (SPMD) programming model to developers using it. Each process executes a common control flow, which can call on the framework to provide iterators over boxes of the overall mesh that reside on that process. The base version of the framework provides serial and MPI implementations. In the MPI implementation, execution follows the BSP model, in that each process repeatedly does all necessary computations on all owned boxes during a phase, and then all communication relating to those boxes. The strict synchronization of every increment of computation or communication imposed substantial costs of load imbalance, network latency, and noise.

Like many AMR frameworks, Chombo decomposes its work so that there are typically many units on each processor. Thus, one would hope that this *overdecomposition* could be exploited to let units that have their necessary inputs make progress while others wait for incoming messages. However, its

SPMD/BSP implementation paradigm ran counter to any mode of opportunistic execution. Instead, it synchronized progress of all units on a processor to each other, even when they could have made independent progress.

The initial goal of the present work on the Chombo framework was to introduce asynchronous execution on the level of each box. After implementation using the Charm++ programming model and runtime system, it was found that this shift alone was insufficient to substantially mitigate the scalability concerns mentioned above. These problems remained because even common hyperbolic PDE demonstration codes in Chombo retained frequent synchronization in the form of global reductions to compute maximum wave speeds and associated safe timestep lengths. By restructuring when and how these reductions were performed, the global synchronization became much less frequent, and the pattern of execution became much more favorable.

An additional goal of this effort was to re-engineer Chombo in such a way that applications would require only minimal modifications to benefit from the revised execution model. This goal was largely met. Meeting this goal required the development of new techniques for encapsulating the execution of existing SPMD 'legacy' code to run asynchronously, and associated improvements in the Charm++ runtime system infrastructure used in place of MPI in the revised implementation.

8.1 Transparent Asynchronous Execution of Existing SPMD Application Code

One of the desires in approaching this project was that existing applications built on the Chombo framework benefit from this work with little or no modification to code outside the framework. In principle, the sharply delineated interface between driver code in the framework and application logic and user provided classes was expected to facilitate this goal. By and large that expectation was met, but with caveats coming from the 'layer cake' call graph structure of the Chombo code – framework and client code alternates several times, as shown in figure 8.1.

Unsurprisingly, accomplishing this task required modifications at several points in this stack. The individual pieces of code affected are enumerated in figure 8.2. Nodes in this graph represent logical units of work on the code.

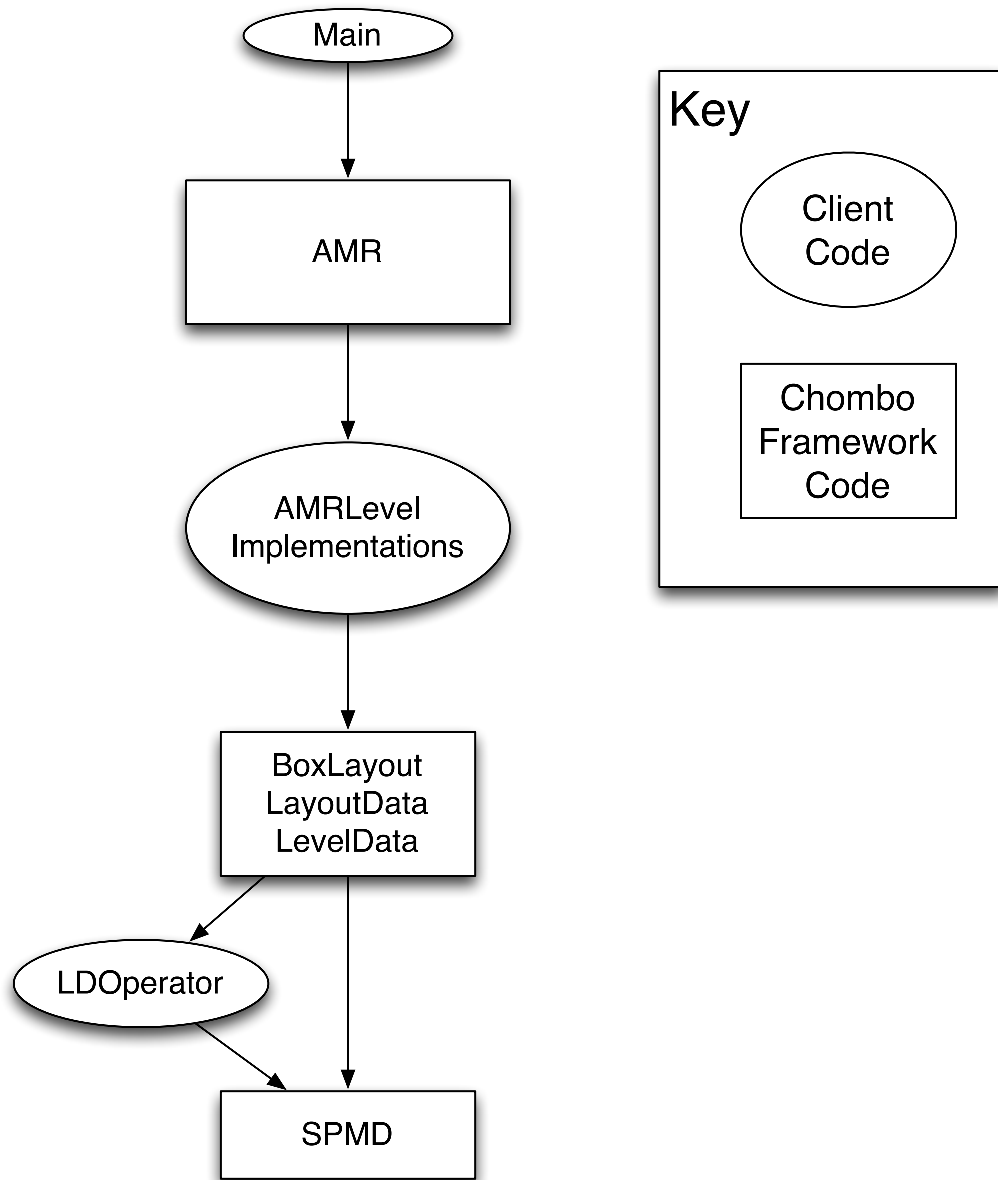


Figure 8.1: CHOMBO's 'layer cake' structure of alternating layers of client and framework code. The work described in this chapter took great care to isolate changes inside the framework code, to minimize the impact of porting on application code.

Edges in the graph depict conceptual dependences in developing and understanding the revised implementation. Changes to the individual components are described in the remainder of this section.

8.1.1 Application Initialization

The modifications to top-level application code to enable asynchronous execution were truly minimal. The entire patch necessary for a typical example is shown in Listing 8.1. The common header for Charm++-MPI interoperation gets included. Since Chombo applications can be configured entirely through an input file, command-line arguments get adapted so they can primarily be passed through to the Charm++ runtime system. Calls to additional initialization and finalization routines are added. All of these changes are essentially the same as in any code that wishes to deploy Charm++-MPI interoperation. Compiling the application code depends only on the added header, and not on any part of the Charm++ toolchain or any generated files.

8.1.2 Encapsulating Per-Box Independent Execution

The control flow involves communication operations that express true data dependencies. In the original MPI implementation of Chombo, these are thus receives that have to be waited on before the operation returns. Because multiple boxes assigned to a processor share a single flow of control, waiting to satisfy all of these dependencies effectively synchronizes execution across all of the boxes on each processor. To break down this synchronization, we provide an independent flow of control for each box, in the form of user-level threads. Every processor will host several of these threads rather than the single master thread it had previously.

To preserve existing logic, each of these threads needs to execute essentially the *entire* control flow. Thus, the present design replicates that control flow for each box, as if it were running alone on a dedicated processor. Each of those threads must be able to identify the box it serves, have its own instance of notionally “global”, or “per process” state, especially state describing simulation progress, and carry meta-data sufficient to control when

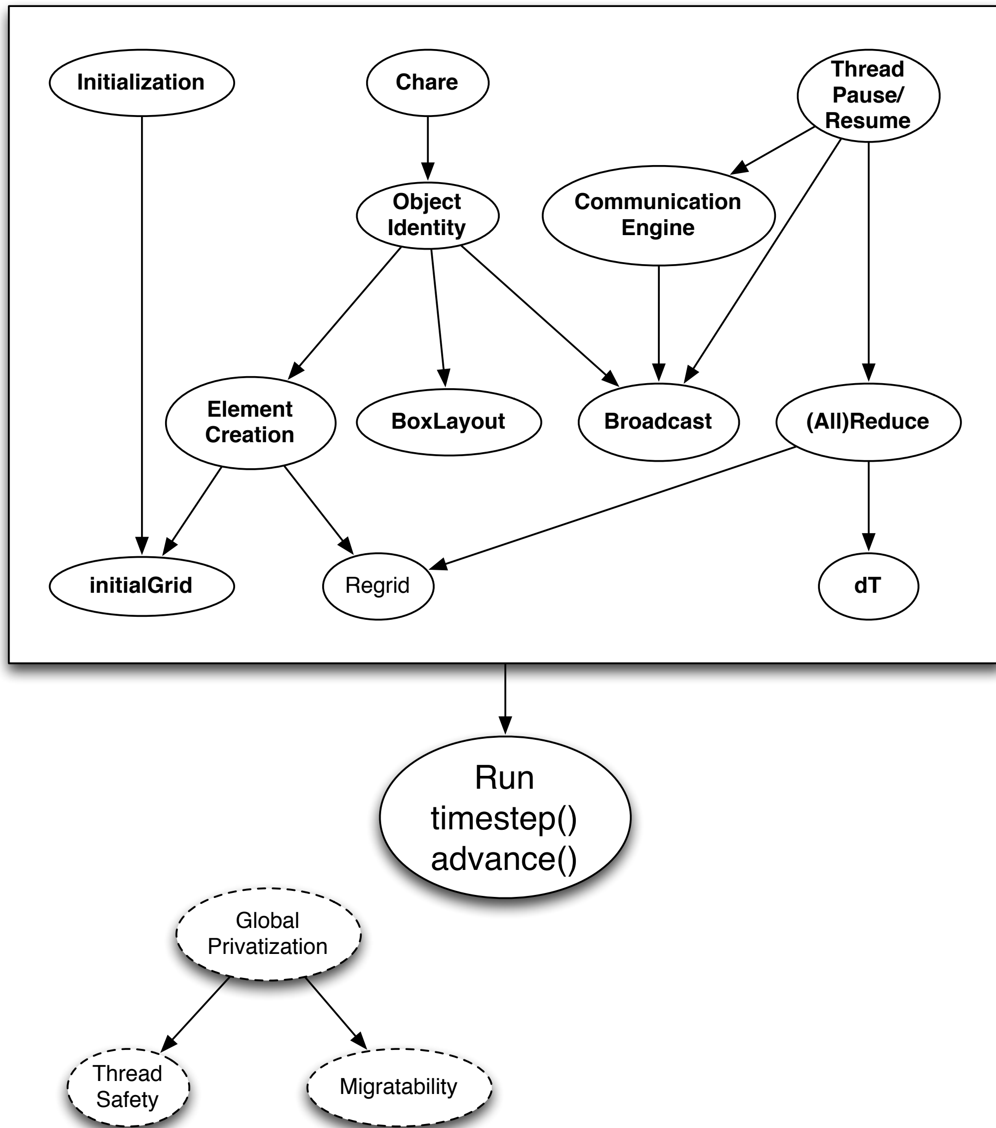


Figure 8.2: The portions of code changed in the course of adapting Chombo from BSP execution on MPI to asynchronous execution on Charm++. Arrows indicate dependencies in understanding the changes in each component.

```
releasedExamples/AMRGodunov/execPolytropic/amrGodunov.cpp
@@ -37,7 +37,9 @@

 #include <mpi.h>
+#include <mpi-interoperate.h>
+
@@ -106,18 +109,25 @@ int main(int a_argc, char* a_argv[])

   // Parse the command line and the input file (if any)
-  ParmParse pp(a_argc-2,a_argv+2,NULL,inFile);
+  ParmParse pp(0, NULL, NULL, inFile);
+
+  a_argv[1] = a_argv[0];
+  a_argv++;
+  a_argc--;
+  AMR::charmInit(a_argc, a_argv);
+

   // Run amrGodunov, i.e., do the computation
   amrGodunov();

   // Exit MPI
   CH_TIMER_REPORT();
   dumpmemoryatexit();
+  CharmLibExit();
   MPI_Finalize();
 }
```

Listing 8.1: Minimal application code changes necessary to run on the framework adapted to Charm++

```
module ChomboCharm {
  include "REAL.H";

  namespace ChomboCharm {
    initproc registerCharePointer();

    message AMRData {
      char buffer[];
    };

    // Indexed by generation, level, index in box vector
    array [3D] AMRChare {
      entry AMRChare();

      entry [threaded] void initialGrid(int e_generation,
          int e_level_limit, bool e_provisional);
      entry [threaded] void initialDt();

      entry [threaded] void run(Real e_max_time,
                                int e_max_step);

      entry AMRChare(int e_finest_level,
            vector<int> e_definingGenerations,
            Vector< Vector<Box> >& e_oldBoxes,
            vector<int> e_steps_since_regrid);
      entry [threaded] void continueFromRegrid(
          int e_regridBaseLevel,
          Vector< Vector<Box> >& e_new_grids,
          Real e_max_time, Real e_cur_time, Real e_dt_base,
          int e_max_step, int e_cur_step);
      entry void postInsertion(CkReductionMsg *m);

      entry void moveData(AMRData* e_data);

      entry void allReduce(CkReductionMsg *m);
      entry void broadcast(AMRData* e_data);
    };
  }
};
```

Listing 8.2: Charm++ interface definition file for ported Chombo

```
namespace ChomboCharm {

CtvDeclare(AMRChare *, runningChare);

class AMRChare : public CBase_AMRChare {
  // Other members elided

  enum SleepReason {NONE, COMM, ALLREDUCE, BROADCAST,
     INSERTION};
  SleepReason m_sleepReason = NONE;
  CthThread m_thread = NULL;

  void pause(const char *a_cause, const SleepReason
     a_reason) {
    CkAssert(m_sleepReason == NONE);
    CkAssert(m_thread == NULL);
    m_sleepReason = a_reason;

    m_thread = CthSelf();
    CthSuspend();

    CkAssert(m_thread == NULL);
    CkAssert(m_sleepReason == NONE);
  }

  void resume(const char *a_cause, const SleepReason
     a_reason) {
    if (a_reason != m_sleepReason) {
      chout() << "Mismatched pause/resume reasons: "
              << "expecting " << m_sleepReason
              << ", got " << a_reason << " / " << a_cause
              << endl;
      CkAbort("bad resume from pause");
    }

    CthAwaken(m_thread);
    m_thread = NULL;

    m_sleepReason = NONE;
  }
};
}
```

Listing 8.3: Definitions for the user-level thread constructs used to encapsulate control flow of existing CHOMBO code

its execution should block or resume. We encapsulate all of this state and stop/start control logic in a CHARM++ chare array element identified with each box. These distinct elements also serve as the endpoints for communication between boxes, since threads in CHARM++ are not directly addressable.

The chare definition code is shown in listing 8.2, and the associated user-level thread logic can be found in listing 8.3. In the chare definition, the `[threaded]` attribute on some entry methods indicates that the runtime system should spawn a user-level thread in which to run the call to that method from start to finish. Methods so marked contain direct transplants of previous CHOMBO code, which contain calls into client application code, which may call for blocking communication. The thread control code provides the bi-directional binding between a running thread and the chare within which it runs. Code lower in the stack can reference the thread-local variable `runningChare` to get a pointer to the encapsulating chare. Using this pointer, it can then record the thread's identity in the chare before pausing. Subsequently, code in the chare that detects that it has satisfied the reason for the thread to pause can use the stored reference to awaken it.

Contrast with other dynamic tasking parallel runtime systems

This design present a strong contrast with that of the Uintah AMR framework and other dynamic tasking parallel runtime systems. Per-box threaded CHOMBO represents its entire dynamic task DAG in the compiled control flow of the framework and application code. Uintah constructs a run-time representation of its entire task DAG, and explicitly numbers and sequences the nodes and data dependence edges within that graph. This scheduling step has been observed to represent a major bottleneck in Uintah development and usage, since its performance does not scale as well as the computation that it orchestrates. Other dynamic tasking parallel runtime systems range between these extremes.

KAAPI [76] uses an explicit representation of the data dependence graph generated by application code. The representation is possibly made succinct by parameterization. It does no global scheduling, and so avoids such a bottleneck. It exploits this explicit representation for unique rapid-recovery fault tolerance support [11]. In contrast, message-logging fault tolerance in CHARM++ must record more information by observing runtime communi-

cation traffic and replay more of execution during recovery [77].

OmpSs uses an explicit in-memory representation of task dependencies. These dependencies are implicitly derived from dependences on units of data described by the code. The derivation of data dependencies in OmpSs can come either from programmer-provided annotations or compiler analysis.

8.1.3 Uniquely Identifying Chare Array Elements

Each chare array element must have a distinct *index*, for the RTS to manage it, to identify itself, and for other objects to interact with it. For many Charm++ applications, this index represents each element's logical "place" or "coordinate" in the computational structure. For instance, applications using spatial decomposition naturally map spatial coordinates to and from array indices [78]. Applications using structured data decomposition often index by the corresponding unit of data; e.g. linear algebra computations indexing by matrix block [50]. Other applications use their array elements as undistinguished members of a set, for which indices carry no semantic content. One example of this structure is ChaNGa's array of 'bucket' objects, each of which owns an arbitrary set of pieces of the Barnes-Hut tree [79]. Another example is EpiSimdemics' partitioning of people and places among collections of `PeopleManager` and `LocationManager` objects, respectively [14].

The present Charm++ implementation of Chombo uses a hybrid of the approaches described above, and introduces an additional new approach of its own. Each element index is a 3-vector of integers. One integer represents the `generation` of the chare and encapsulated box, meaning the number of regridding cycles that have been performed up to the point where the present set of boxes for the designated level was formed. This introduces a temporal aspect of object identification, corresponding to simulation progress, that is novel among Charm++ applications. Another integer represents the `level` of refinement of the box for which the chare is responsible, following the structure-representation style. The third integer represents the the `index` of the box within that level of refinement in Chombo's structural metadata, following the set indexing style. The code defining this relation is shown in Listing 8.4.

The process of globally sorting and numbering all of the boxes making up a given level has been shown to be a scalability bottleneck in several

```
class AMRChare : public CBase_AMRChare {
  // Other members elided

  int getGeneration() { return thisIndex.x; }
  int getLevel() { return thisIndex.y; }
  int getIndex() { return thisIndex.z; }

  int m_definingGeneration;
  int m_definingLevel;
  std::vector<int> m_definingGens;

  // During regrid operations, the defining generation of any
  // BoxLayout instances defined prior to the actual
     regrid() call is
  // at least one less than the generation of any new objects
  int definingGen()
  {
    int g = m_definingGens[m_definingLevel];
    if (g == getGeneration())
      return g-1;
    else
      return g;
  }

  Vector<AMRLevel *> m_amrlevels;
};
```

Listing 8.4: Declarations for basic members of the chare objects to identify themselves, and the levels of the mesh being defined during execution

Chombo applications. In line with the 'semantic naming' pattern described in Section 3.7, it would be desirable to replace the `index` component with something that does not require these steps. Within a level, boxes at a given position within the domain are unique. Additionally, any reference one box makes to another is based on knowing that other box's position. Thus, that position could itself be used as the final part of a generated unique identifier.

8.1.4 Aligning `BoxLayout` with Chares for Allocation and Computation

Chombo uses the `BoxLayout` abstraction to represent a collection of boxes at a common level of refinement. This abstraction lets each process allocate space and construct the objects for which it's responsible. It also provides iterators over local boxes used to guide the actual computation work to be applied to each box.

Adaptation of Chombo to Charm++ required modifying `BoxLayout` to map boxes to objects rather than processors. Each instance identifies with a generation and level being formed at the time of its construction or definition, based on the chare-level control variables set in the scaffolding built around calls to user code. This usage is shown in Listing 8.5. The control variables are set in constructors and other methods supporting `initialGrid` and `regrid` operations.

If the defined `generation` and `level` variables match the chare's identity, then the `BoxLayout` instance allocates data for a single box whose position and dimensions are determined by indexing into the layout description with the chare's `index` identity element. If these variables do not match the chare's identity, then the `BoxLayout` instance can conclude that it is not responsible for any underlying data or computation. This is shown in Listing 8.6.

8.1.5 Initial Grid Construction

The logic of Chombo's original implementation of `AMR::initialGrid()` has ported somewhat directly into the Charm++ implementation. The defining difference is the introduced separation between the top-level logic and the per-object logic. The top-level logic runs in the SPMD mode, and

```
diff a/lib/src/BoxTools/BoxLayout.H
@@ -651,6 +662,8 @@ protected:
   RefCountedPtr<DataIterator>           m_dataIterator;
   RefCountedPtr<Vector<LayoutIndex> >   m_indicies;
+  RefCountedPtr<int>                    m_level;
+  RefCountedPtr<int>                    m_generation;

diff a/lib/src/BoxTools/BoxLayout.cpp
@@ -68,8 +70,11 @@ BoxLayout::BoxLayout()
    m_sorted(new bool(false)),
    m_dataIterator(RefCountedPtr<DataIterator>()),
-   m_indicies(new Vector<LayoutIndex>())
+   m_indicies(new Vector<LayoutIndex>()),
+   m_level(new int(CtvAccess(runningChare) ?
   CtvAccess(runningChare)->m_definingLevel : -1)),
+   m_generation(new int(CtvAccess(runningChare) ?
   CtvAccess(runningChare)->m_definingGeneration : -1))
 { }
@@ -639,6 +639,17 @@ class BoxLayout {
+
+  int getLevel()       const { return *m_level; }
+  int getGeneration() const { return *m_generation; }
+
@@ -237,6 +229,8 @@
 void BoxLayout::define(const Vector<Box>& a_boxes, const
    Vector<int>& a_procIDs)
 {
+  m_level = RefCountedPtr<int>( new
   int(CtvAccess(runningChare) ?
   CtvAccess(runningChare)->m_definingLevel : -1));
+  m_generation = RefCountedPtr<int>( new
   int(CtvAccess(runningChare) ?
   CtvAccess(runningChare)->m_definingGeneration : -1));
   checkDefine(a_boxes, a_procIDs);
   const int num_boxes = a_boxes.size();
```

Listing 8.5: Changes to CHOMBO's `BoxLayout` class to identify itself with the generation and level of the layout being defined. This listing elides copy operations and other duplicative changes that provide no additional insight.

```
diff a/lib/src/BoxTools/BoxLayout.cpp
@@ -121,32 +128,17 @@
 void BoxLayout::buildDataIndex()
 {
-  std::list<DataIndex> dlist;
-  unsigned int index = 0;
-  unsigned int datIn = 0;
-  unsigned int p = CHprocID();
-  int count=0;
-  const Entry* box;
-
-  while (index < size()) {
-      box = &(*(m_boxes))[index];
-      if (box->m_procID == p) {
-          DataIndex current(index, datIn, m_layout);
-          dlist.push_back(current);
-          count++;
-          datIn++;
-      }
-      ++index;
-  }
-
-  m_dataIndex = RefCountedPtr<Vector<DataIndex> >(new
   Vector<DataIndex>(count));
-  std::list<DataIndex>::iterator b=dlist.begin();
-  for (int i=0; i<count; ++i, ++b) {
-      m_dataIndex->operator[](i) = *b;
-  }
+  AMRChare* chare = CtvAccess(runningChare);
+  if (chare &&
+      chare->getGeneration() == *m_generation &&
+      chare->getLevel()      == *m_level)
+  {
+    m_dataIndex = RefCountedPtr<Vector<DataIndex> >(new
   Vector<DataIndex>(1));
+    (*m_dataIndex)[0] = DataIndex(chare->getIndex(), 0,
   m_layout);
+  } else {
+    m_dataIndex = RefCountedPtr<Vector<DataIndex> >(new
   Vector<DataIndex>(0));
+  }
 }
```

Listing 8.6: Changes to CHOMBO's `BoxLayout` class to allocate data and assign computation by containing object instead of host processor

is now responsible for iteratively inserting chare objects for each box, and running the Charm++ scheduler to ask those objects to do the prior per-box work. The individual objects assume responsibility for constructing and initializing the client-defined `AMRLevel` objects. The ported code inside each chare is show in listing 8.7. A key point to note is that it sets the variables `m_definingGeneration` and `m_definingLevel` which are used to communicate across framework layers of the stack to any `BoxLayout` objects defined by the client code run within each virtual method call on the `AMRLevel` objects.

The code, run in SPMD mode, that constructs the chare objects and calls the `initialGrid()` method on them can be seen in listing 8.8. It takes the vector of boxes in each level being formed, and inserts an element corresponding to each of them. This work is distributed in a round-robin fashion over the processors in the job, to avoid a sequential bottleneck in either the iteration over a large number of boxes, or the message injection to create each object. This bottleneck was apparent in earlier versions of the adaptation.

8.1.6 Communication Engine

The bidirectional mapping between box ownership and object identity serves to simplify the communication engine in Chombo. When determining which boxes to communicate with, iteration over 'all the boxes on this processor' becomes the trivial 'my box' if the layout involved in communication matches the containing chare, or else none at all. The subsequent transformation of box-to-box interactions to messages between their assigned processors also becomes trivial. Messages are directed or waited upon in literal terms of the partner object's identity. The concern of locating that object on a particular processor and directing messages here is delegated to existing components of the Charm++ RTS infrastructure [80].

The code for initiating the communication engine can be seen in listing 8.9. It begins by passing all necessary parameters through to the running chare object. The object is then responsible for generating messages to send and setting up the list of messages it expects to receive before execution can continue. The send and receive enumeration can be seen in listing 8.10.

Code for the receive path can be seen in listing 8.11. This illustrates how messages are matched between sends and corresponding receives. The code counts how many communication operations have been performed, and

```
void AMRChare::initialGrid(int e_generation,
    int e_level_limit, bool e_provisional) {
  Running running(this);
  // Refinement replaced this generation of objects, so
      destroy this object rather than doing work.
  if (getGeneration() < e_generation) {
    thisProxy[thisIndex].ckDestroy();
    return;
  }
  m_definingGeneration = e_generation;
  m_finest_level = e_level_limit;
  AMR *amr = CsvAccess(g_local_amr);

  int &defLevel = m_definingLevel;

  for (defLevel = 0; defLevel <= e_level_limit; ++defLevel)
    m_amrlevels[defLevel]
      ->initialGrid((*amr->c_old_grids)[defLevel]);
  for (defLevel = e_level_limit; defLevel >= 0; --defLevel)
    m_amrlevels[defLevel]->postInitialGrid(false);
  for (defLevel = 0; defLevel <= e_level_limit; ++defLevel)
    m_amrlevels[defLevel]->initialData();

  if (e_provisional) {
    // Initialization isn't done yet, so tag for refinement
    for (defLevel = 0; defLevel <= e_level_limit;
        ++defLevel) {
      IntVectSet ivs;
      m_amrlevels[defLevel]->tagCellsInit(ivs);
      (*amr->c_tags)[defLevel] |= ivs;
    }
  } else {
    // Initialization is done, so finish setting up the
        extra levels
    for (defLevel = e_level_limit + 1;
          defLevel <= amr->m_max_level; ++defLevel) {
      m_amrlevels[defLevel]->initialGrid(Vector<Box>());
      m_amrlevels[defLevel]->initialData();
    }
    // call post-initialize once all the levels have been
        defined
    for (defLevel = e_level_limit; defLevel >= 0; --defLevel)
      m_amrlevels[defLevel]->postInitialize();
  }
  m_communicationOperations = 0;
  m_finest_level_old = m_finest_level;

  if (getLevel() == 0 && getIndex() == 0)
    CkStartQD(CkCallback(CkCallback::libCkExit));
}
```

Listing 8.7: Adaptation of the code from `AMR::initiaGrid()` to run encapsulated in each object, setting the `definingLevel` variable to inform `BoxLayout` definition as it goes

```
for (int level = 0; level <= top_level; ++level) {
  m_amrlevels[level]->initialData();
  // Round-robin insertions across processors to avoid
     bottleneck
  for (int i = procID();
       i < old_grids[level].size();
       i += numProc()) {
    // Generation is given by the 'top_level' loop counter
    chares(top_level, level, i).insert();
  }
}

if (procID() == 0) {
  // Broadcast to all chares from just one processor
  chares.initialGrid(top_level, top_level, true);
  chares.doneInserting();
}

// Switch from SPMD MPI execution into Charm++
CsvAccess(isCharmRunning) = true;
StartCharmScheduler();
CsvAccess(isCharmRunning) = false;
```

Listing 8.8: Code to insert new chare array element objects and call `initialGrid()` on them

during each operation, how many messages have been sent between each pair of senders and recipients. These counters take the place of MPI's message matching logic.

Chombo's original MPI implementation of its communication engine effectively modeled the MPI-3 neighborhood variable-length all-to-all collective communication operation. Thus, the adaptation to per-box asynchronous execution in Charm++ represents an instance of the pattern of incrementally producing, communicating, and consuming data that would have been passed through a synchronizing collective (described in Section 3.3).

It's possible to apply an additional optimization to short-circuit data transfer between objects assigned to the same processor, rather than allocating a message buffer and copying the data into and back out of that buffer. This would also avoid an extraneous trip through the Charm++ scheduler. Current benchmarks seemed unlikely to benefit substantially from this optimization. Thus, this optimization is not implemented in the adaptation presented here.

```
template<class T>
void BoxLayoutData<T>::makeItSoBegin(Args... args) const {
  CtvAccess(runningChare)->beginCommunication(
    auto_ptr<CommOp> (new CommOpT<T>(args...)));
}
void AMRChare::beginCommunication(
    std::auto_ptr<CommOp> a_commOp) {
  m_commOp = a_commOp;
  m_communicationOperations++;
  const int srcGeneration  = m_commOp->getSrcGeneration();
  const int srcLevel       = m_commOp->getSrcLevel();
  const int destGeneration = m_commOp->getDestGeneration();
  const int destLevel      = m_commOp->getDestLevel();
  // Enumerate receives
  if (destGeneration == getGeneration() && destLevel ==
     getLevel()) {
    m_recvMessageCounts.clear(); // Sequence number counters
    CopyIterator it{m_commOp->getCopier(), CopyIterator::TO};
    enumerateReceives(it, srcGeneration, srcLevel);
    it = CopyIterator(m_commOp->getCopier(),
        CopyIterator::LOCAL);
    enumerateReceives(it, srcGeneration, srcLevel);
  }
  // Send stuff
  if (srcGeneration == getGeneration() && srcLevel ==
     getLevel()) {
    m_sendMessageCounts.clear(); // Sequence number counters
    CopyIterator it{m_commOp->getCopier(),
        CopyIterator::FROM};
    enumerateSends(it, destGeneration, destLevel);
    it = CopyIterator(m_commOp->getCopier(),
        CopyIterator::LOCAL);
    enumerateSends(it, destGeneration, destLevel);
  }
}
template<class T>
void BoxLayoutData<T>::makeItSoEnd(Args...) const {
  AMRChare* chare = CtvAccess(runningChare);
  chare->processMessages();
  if (chare->m_expectedMessages.size() != 0) {
    chare->pause("comm", AMRChare::COMM);
  }
}
```

Listing 8.9: Logic to set up a communication operation

```
void AMRChare::enumerateReceives(CopyIterator& it,
    const int srcGeneration, const int srcLevel) {
  for (; it.ok(); ++it) {
      const MotionItem& item = it();

      CkIndex3D sender = {srcGeneration,
                          srcLevel,
                          item.fromIndex.intCode()};

      m_expectedMessages.push_back(
          ExpectedMessage(sender,
                          m_communicationOperations,
                          m_recvMessageCounts[sender]++,
                          item.toIndex,
                          item.toRegion));
  }
}

void AMRChare::enumerateSends(CopyIterator& it,
    const int destGeneration, const int destLevel) {
  for (; it.ok(); ++it) {
      const MotionItem& item = it();

      int size = m_commOp->getSize(item);
      AMRData* message = new (size)
        AMRData(thisIndex,
                m_communicationOperations,
                m_sendMessageCounts[item.toIndex.intCode()],
                size);
      m_commOp->linearOut(item, message->buffer);
      thisProxy(destGeneration,
                destLevel,
                item.toIndex.intCode()
                ).moveData(message);
      m_sendMessageCounts[item.toIndex.intCode()]++;
    }
}
```

Listing 8.10: Code to enumerate messages to send and messages expected to be received

```
void AMRChare::moveData(AMRData* e_message) {
  m_messages.push_back(e_message);
  processMessages();
  // No more expected messages, and . . .
  if (m_expectedMessages.size() == 0
      // . . . the chare had a thread blocked, waiting for
         messages
      && m_thread != NULL && m_sleepReason == COMM) {
    resume("comm", COMM);
  }
}

void AMRChare::processMessages() {
  for (ExpectedMessage expectation : m_expectedMessages) {
    auto messageIter = find_if(m_messages.begin(),
                               m_messages.end(),
                               messageMatches(*expectation));
    if (messageIter != m_messages.end()) {
      AMRData* message = *messageIter;
      m_commOp->linearIn(expectation->m_toIndex,
                         expectation->m_toRegion,
                         message->buffer);
      // XXX Does not handle dynamic allocation case
      expectation = m_expectedMessages.erase(expectation);
      m_messages.erase(messageIter);
      delete message;
    } else {
      expectation++;
    }
  }
}

struct messageMatches {
  AMRChare::ExpectedMessage expectation;
  messageMatches(AMRChare::ExpectedMessage e) :
     expectation(e) {}
  bool operator() (AMRData* message) {
    return
      expectation.m_sender            == message->sender &&
      expectation.m_communicationStep ==
         message->communicationStep &&
      expectation.m_sequence          == message->sequence;
  }
};
```

Listing 8.11: The message receive path, which runs outside thread control. Hence, when all messages have been received, it must awaken the blocked thread

8.1.7 Collective Implementation

Due to the implementation's use of MPI/Charm++ interoperability and mode-switching execution [81], the implementations of some collective operations have to be adapted to be aware of which mode is active, and call the underlying system accordingly. Since Chombo presents a fairly narrow API to applications, the only operations that needed adaptation were broadcasts and all-reduce.

Broadcast

The modifications to `broadcast()` are shown in Listing 8.12. The corresponding Charm++ code that the common function calls when execution is running in Charm++ mode is shown in Listing 8.13.

The Charm++ code begins by looking up the operating chare as described in Section 8.1.3, and passes the necessary broadcast arguments to its `AMRChare::beginBroadcast()` method. As with other communication operations described in Section 8.1.6, the method begins by incrementing the chare's communication operation sequence number to be used for matching and storing a buffer pointer to indicate where the received data should be made available. If the chare is the designated 'root' object, based on owning box 0 of level 0 in the current generation, then it constructs and sends the broadcast message to all chares in the array. All chares check whether the broadcast message has arrived before their own control flow was ready to receive it; if it hasn't, they block their individual execution until the message is delivered, as described in Section 8.1.2.

The receive path for broadcasts mimics that of other box-to-box communication operations described in Section 8.1.6. The `AMRChare::broadcast()` entry method saves the delivered message in its chare object's matching buffer. Then it checks if the chare was blocked waiting for a broadcast, and if so, whether this broadcast was the one expected. If so, its execution through the primary control flow is resumed, as described in Section 8.1.2. Message matching and processing follows a similar control flow to normal box-to-box messages. It looks for a message from the root with the right sequence number. If that message is found, its payload is copied out to the receive buffer, the message is discarded, and the method returns success. If not, it returns failure.

```
diff --git a/lib/src/BaseTools/SPMDI.H
   b/lib/src/BaseTools/SPMDI.H
index 9d31d3b..5586502 100644
--- a/lib/src/BaseTools/SPMDI.H
+++ b/lib/src/BaseTools/SPMDI.H
@@ -123,45 +113,57 @@ gather(Vector<T>& a_outVec, const T&
   a_input, int a_dest)
 template <class T>
 inline void
 broadcast(T& a_inAndOut,  int a_src)
 {
+  bool isCharm = CsvAccess(isCharmRunning);
+  bool isRoot = isCharm ?
+    ChomboCharm::isBcastRoot() :
+    procID() == a_src;
   int isize;
-  if (procID() == a_src)
+  if (isRoot)
     isize = linearSize(a_inAndOut);

-  MPI_Bcast(&isize, 1, MPI_INT, a_src, Chombo_MPI::comm);
+  if (isCharm)
+    ChomboCharm::broadcast((char*)&isize, sizeof(isize));
+  else
+    MPI_Bcast(&isize, 1, MPI_INT, a_src, Chombo_MPI::comm);

   void* broadBuf = mallocMT(isize);

   //take inAndOut from src and put it into broadBuf
-  if (procID() == a_src)
+  if (isRoot)
     linearOut(broadBuf, a_inAndOut);

-  //broadcast broadBuf to all procs
-  MPI_Bcast(broadBuf, isize, MPI_BYTE, a_src,
   Chombo_MPI::comm);
-
+  if (isCharm)
+    ChomboCharm::broadcast(broadBuf, isize);
   else
+    MPI_Bcast(broadBuf, isize, MPI_BYTE, a_src,
   Chombo_MPI::comm);

   //take broadBuf and put back into inAndOut if not src
-  if (procID() != a_src)
+  if (!isRoot)
     linearIn(a_inAndOut, broadBuf);

   //delete memory for buffer
   freeMT(broadBuf);
 }
```

Listing 8.12: Modifications to the broadcast implementation to switch between MPI and CHARM++ execution modes

```
void broadcast(char *a_buf, int a_size) {
  AMRChare *chare = CtvAccess(runningChare);
  CkAssert(chare);
  chare->beginBroadcast(a_buf, a_size);
}
bool isBcastRoot() {
  AMRChare *chare = CtvAccess(runningChare);
  return chare->getLevel() == 0 && chare->getIndex() == 0;
}
void AMRChare::beginBroadcast(char *a_buf, int a_size) {
  m_communicationOperations++;
  m_broadcastBuf = a_buf;

  if (isBcastRoot()) {
    AMRData *msg = new (a_size) AMRData(thisIndex,
       m_communicationOperations, 0, a_size);
    memcpy(msg->buffer, a_buf, a_size);
    // Entry method invocation broadcast to all elements of
       the chare array
    thisProxy.broadcast(msg);
  }

  if (!processBcastMsg())
    pause("broadcast", AMRChare::BROADCAST, m_cur_step);
}
// Entry method called through thisProxy, above
void AMRChare::broadcast(AMRData *e_msg) {
  m_messages.push_back(e_msg);
  if (m_thread != NULL && m_sleepReason == BROADCAST &&
     processBcastMsg())
    resume("broadcast", BROADCAST);
}
bool AMRChare::processBcastMsg() {
  CkIndex3D root = {m_definingGens[0], 0, 0};
  vector<AMRData*>::iterator recvMsgIt =
    find_if(m_messages.begin(), m_messages.end(),
            messageMatches(ExpectedMessage(root,
               m_communicationOperations, 0)));
  if (recvMsgIt != m_messages.end()) {
    AMRData *recvMsg = *recvMsgIt;
    memcpy(m_broadcastBuf, recvMsg->buffer, recvMsg->size);
    m_messages.erase(recvMsgIt);
    delete recvMsg;
    m_broadcastBuf = NULL;
    return true;
  }
  return false;
}
```

Listing 8.13: Charm++ implementation of broadcast to distribute a value from the designated 'root' box to all other box chare objects.

Allreduce

The other commonly used collective operation in CHOMBO framework and application code was `MPI_Allreduce`. The framework did not provide a wrapper function for this, and so many places in the code called it directly. Adaptation to mixed MPI/CHARM++ execution required an ability to choose which engine's implementation to call. Thus, the first step of adaptation was to implement a wrapper `allReduce()` routine for the types required: `float, double, int, IntVectSet`. The implementation of this wrapper and its backing CHARM++ code can be seen in Listing 8.14. All call sites of `MPI_Allreduce` in relevant portions of the CHOMBO repository were replaced with calls to the wrapper. Some instances of a pattern implementing the same effect through a sequence of gather/reduce/broadcast were also replaced.

Because an all-reduce operation inherently depends on contributions from every element before any element can receive the results and continue execution, its control flow is simplified relative to broadcasts. Specifically, no sequence number or other matching construct is necessary, because no more than one can be in flight at a time. Thus, each object makes its contribution, and then pauses its execution until the result is available. The underlying implementation for all types is effectively a reduce-broadcast algorithm using the supporting CHARM++ primitives.

The primary use of this operation was in determining allowable timestep lengths, possibly through the intermediate calculation of maximum wave speeds. As described in Section 8.2, this operation has been lifted from application code into the adapted framework, allowing consolidated use of the non-blocking implementation shown in Listing 8.15.

8.1.8 Collective Coordination

As noted above, each box object is also an independent participant in collective operations. As in MPI, CHARM++ requires contribution to collectives to follow the same sequence across all participants within a given group (MPI communicator or chare array). In my modifications to CHOMBO, this posed some trouble because new elements are created dynamically and expected to consistently participate in collectives following their creation.

```
void AMRChare::beginAllReduce(double& a_inAndOut, MPI_Op
    a_op) {
  // Start a global reduction, performing the specified
     operation
  CkCallback cb(CkIndex_AMRChare::allReduce(NULL),
     thisProxy);
  contribute(sizeof(double), &a_inAndOut,
             a_op == MPI_MAX ? CkReduction::max_double :
                CkReduction::sum_double,
             cb, m_cur_step);

  // Wait for it to complete
  pause("allreduce", ALLREDUCE, m_cur_step);

  // Extract the resulting value
  a_inAndOut = *(double *)m_reductionMsg->getData();

  // Discard the message
  delete m_reductionMsg;
  m_reductionMsg = NULL;
}
void AMRChare::beginAllReduce(IntVectSet& a_inAndOut) {
  // Start a global gather
  vector<Box> boxes = a_inAndOut.boxes().stdVector();
  contribute(sizeof(Box)*boxes.size(), &boxes[0],
     CkReduction::concat,
             CkCallback(CkIndex_AMRChare::allReduce(NULL),
                thisProxy));

  pause("allreduce", ALLREDUCE, m_cur_step);

  // Consolidate the results. NOTE: This should be a reducer
  //  running on the input once, not per recipient!
  Box *b = (Box *)m_reductionMsg->getData(),
      *bend = b + m_reductionMsg->getSize()/sizeof(Box);
  for (; b < bend; ++b)
    a_inAndOut |= *b;

  delete m_reductionMsg;
  m_reductionMsg = NULL;
}
// Entry method target of the above contribute() calls
void AMRChare::allReduce(CkReductionMsg *m) {
  // Store the message received
  m_reductionMsg = m;
  // Resume execution if it was blocked waiting for this
  if (ALLREDUCE == m_sleepReason)
    resume("allreduce", ALLREDUCE);
}
```

Listing 8.14: Wrapper and Charm++ implementation of `allReduce`, using a reduce-broadcast algorithm based on lower-level Charm++ primitives

```
void AMRChare::beginIAllReduce(const double a_in) {
  CkCallback cb(CkIndex_AMRChare::allReduce(NULL),
     thisProxy);
  contribute(sizeof(double), &a_in, CkReduction::max_double,
     cb);
}

double AMRChare::endIAllReduce() {
  if (NULL == m_reductionMsg)
    pause("allreduce", ALLREDUCE, m_wake_prio);

  Real out = *(double *)m_reductionMsg->getData();

  delete m_reductionMsg;
  m_reductionMsg = NULL;
  return out;
}
```

Listing 8.15: Implementation of a non-blocking `allReduce`, used specifically for global stable timestep calculation

The protocols for these operations in Charm++ are slightly deficient for this purpose without additional synchronization [82]. The sequence numbers used to match messages to operations are recorded by each object at the time its insertion call is made from processor level state. That processor level state reflects the furthest progress in the reduction sequence of *any* object in that array residing on that processor. Thus, objects calling for new insertions and then proceeding to make subsequent collective calls lead to later object insertion calls on that processor to have their stamped sequence numbers offset from the intended value. Without modifying this fundamental piece of Charm++ infrastructure, I was forced to insert blocking synchronization after existing objects' calls to insert new objects for the next generation, to enforce waiting until all insertion calls have been made, with the counters in the same state. Since regridding itself remains both synchronous and rare, and this additional synchronization comes as part of regridding, fixing the infrastructure to avoid this was not a priority for gaining additional performance or scalability.

8.2 Reducing Synchronization by Optimizing Global Reductions for Stable Timestep Computation

Ghost cell exchanges and inter-level interpolation/averaging initially presented the most frequent synchronization in the original bulk-synchronous MPI-based Chombo design. Those operations occur many times per step. After that, the next most frequent source of synchronization is the global computation of a stable timestep. In the baseline design of Chombo, this involved application code performing a blocking all-reduce operation for each level of refinement in every step. The framework logic would then compute and apply the minimum value across all levels locally on each process.

Several steps of synchronization-weakening and -eliminating transformations are possible from this baseline. The first general transformation involves lifting the multiple reduction operations into a single operation per step. That single operation can then be made non-blocking, to provide overlap with end-of-step computations that do not depend on its output. Finally, we can examine the potential impact of eliminating this reduction entirely.

Experiments to benchmark these modifications were performed on Edison, using one PE per physical core. For each node count, all variations were run in a common job allocation, to mitigate effects of network topology variation across jobs. The overall performance effect of these steps can be seen in Figure 8.3.

8.2.1 Consolidating Per-Level Reductions

A preliminary change in the AMRGodunov mini-app involved moving the all-reduce calls so that they would occur in back-to-back calls from the framework into application code, after all levels' computations were complete, rather than at the end of each level's computations[1]. Moving the reductions in this way is an application of the 'batching collectives' pattern. This change drastically reduces the excessive dependencies of the original implementation – the beginning of each level's computation on each process was made to wait for the end of the previous level's computation on all processes! This change allows overlap of work among objects at different levels

[1] Specifically, from `AMRLevel::advance()` to `AMRLevel::computeDt()`.

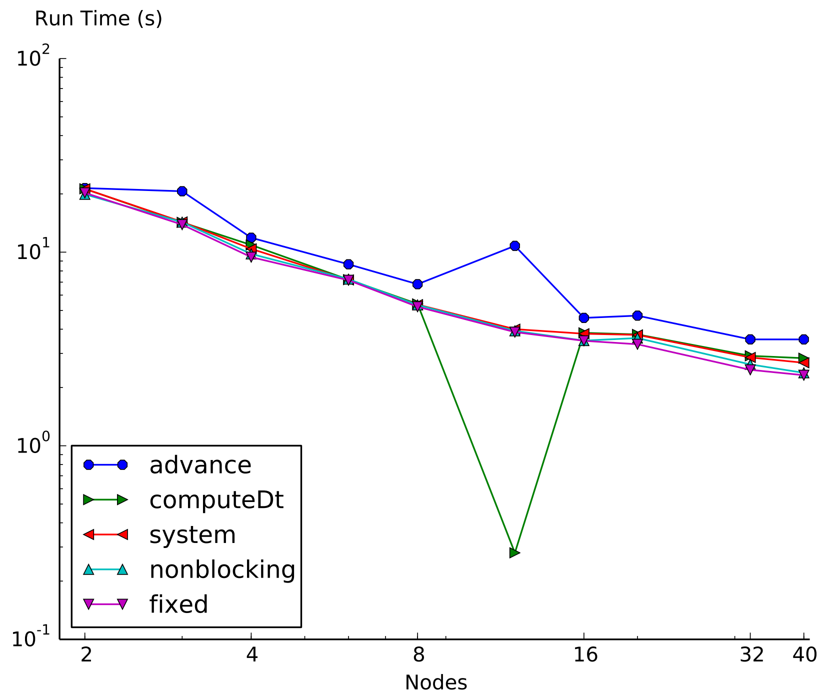


Figure 8.3: The effect of various optimizations to the process of computing a globally stable timestep in the AMRGodunov benchmark. The original code did a global `ALL_REDUCE` at the end of each level's `advance()` method. Successive transformations moved those calls to the `computeDt()` method after all `advance()` calls were complete; lifted them from a per-level call in `computeDt()` to a single call by the system; made that single call nonblocking; and elided that call when a fixed timestep value was set.

of refinement, including sending messages whose contents don't depend on data from earlier levels.

Given independent per-box asynchronous execution, the multi-level overlap and early sends help mitigate per-level load imbalance. As soon as some boxes at a level l have finished their computation and sent whatever messages they will, the dependencies for overlapping boxes at level $l + 1$ can be satisfied, letting them begin execution of that step.

This change widens the main synchronization window to encompass computation at all levels. At the same time, the multiple reductions executed in close succession remain more vulnerable to interference than would be ideal. Overall, this change shows performance improvements across a range of scales.

Surveying existing Chombo applications

The transformation described in this section was first identified and applied the the AMRGodunov example provided with the Chombo framework. This example is taken as a common starting point for writing other Chombo application code, as suggested by the developers of Chombo. Furthermore, the interface specification between the Chombo framework and its application codes suggests a requirement for this mis-design, by demanding that a new `dT` be returned from the `AMRLevel::advance()` method ("Return an estimate of the new time step at this level." [83]), even though the framework discards this value! Thus, I surveyed other example code contained in the Chombo distribution to determine the prevalence of this performance bug.

In the Chombo distribution, the following other codes exhibit this suboptimal structure:

1. `lib/src/EBAMRTimeDependent/EBAMRGodunov`
2. `lib/src/MOLAMRTimeDependent/AMRLevelCons`
3. `releasedExamples/AMRGodunov/srcPolytropic`
4. `releasedExamples/AMRGodunov/srcAdvectDiffuse`
5. `releasedExamples/AMRINS/NavierStokes`
6. `releasedExamples/EBAMRCNS/src/EBAMRCNS`

7. `old/MoveToReleasedExamples/AMRClaw/src/AMRLevelClaw`[2]

8. `old/MoveToReleasedExamples/AMRSelfGravity/charm/AMRLevelSelfGravity`

9. `old/fourthOrderHyperbolic/srcMapped/AMRLevelMappedCons`[3]

10. `old/fourthOrderHyperbolic/srcMappedAMR/AMRLevelMappedCons`

On the other hand, `old/fourthOrderHyperbolic/srcMappedAdvection/AMRLevelAdvect` uniquely does not.

8.2.2 Lifting Reduction Responsibility from Client Application to Framework Code

We note that every application using Chombo's `AMRTimeDependent` driver must follow a similar structure of stable timestep calculation as described above. Based on this observation, we can refactor to shift this responsibility from the application to the framework. Originally, application code would reduce over the locally-computed bounds and return the reduced value to the framework from `AMRLevel::computeDt()`. Instead, the application code can return the result of its purely local calculation at each level to the framework, and let the framework perform the global reduction[4]. Since the framework uses the minimum value obtained across all levels, it can do a single global reduction rather than one per level. This provides only marginal performance benefits, but it sets up opportunities for further optimizations.

The next optimization, now available to make in the framework code, is to use a non-blocking reduction. Chombo applications offer a clear opportunity to overlap this operation with applications' end-of-step work in `AMRLevel::postTimestep()`. This work would typically comprise opera-

[2]This instance does even worse still - it gathers each process's local value to a single rank, does a local MIN-reduction, and then broadcasts the result. Thus, it adds an additional serial bottleneck on top of the excess synchronization.

[3]This shares the noted additional deficiency with AMRClaw.

[4]For the sake of maintaining compatibility of the modified framework with application code before and after this change, we add a new virtual method '`bool AMRLevelFactory::amrlevel_compute_dt_is_local()`' to indicate whether the application's implementation of `AMRLevel` uses the old or new style. The base implementation of this new method returns '`false`', to let existing code continue working without modification.

tions like averaging field values down from fine levels to coarser levels, to ensure consistent results. This provides further performance benefits.

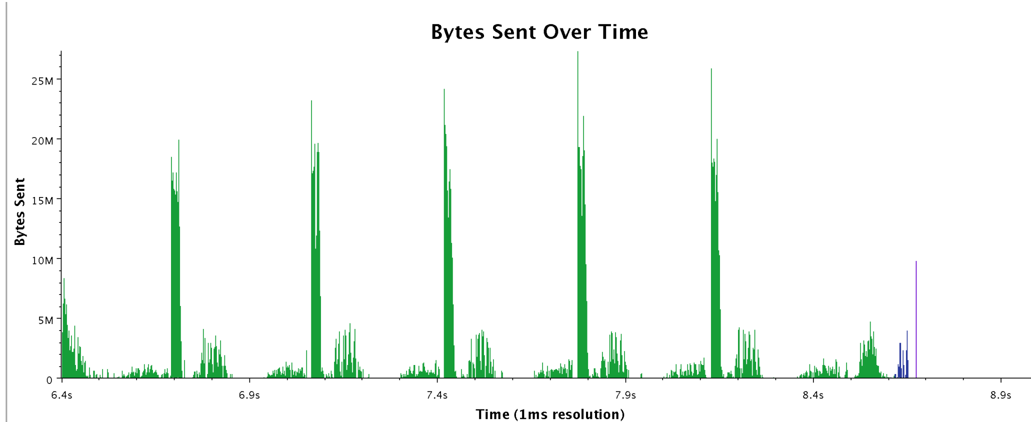
8.2.3 Effect of Eliminating Timestep Reductions

Ideally, no global collective at all would be necessary to compute a timestep that would produce a stable simulation. The inputs that determine it can be computed locally within a 'neighborhood' of the simulation domain, and each neighborhood's value can then be used within that neighborhood, with appropriate coordination at the boundaries. This would be an example of the pattern 'replace synchronizing collectives with p2p messages that achieve the desired effect' (§ 3.6). An implementation of this approach has been independently proposed for development in Enzo-P.

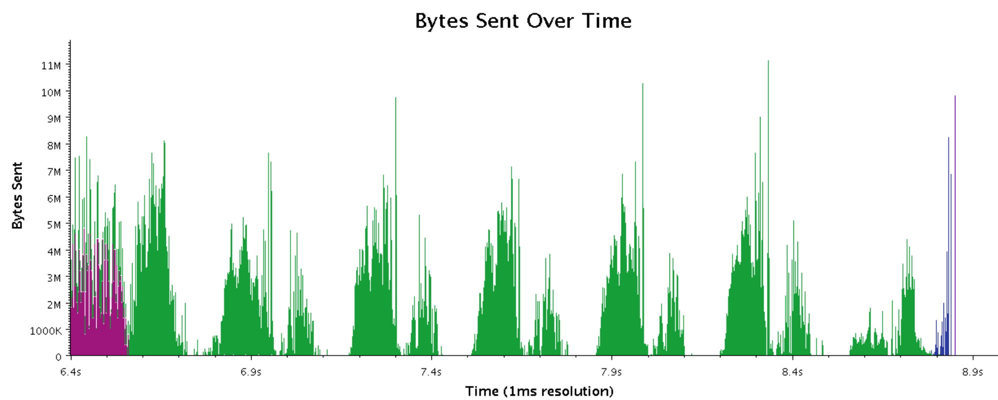
We can model the performance impact of this situation by setting a fixed timestep value for the entire simulation. Since the framework code is now responsible for computing the timestep reduction, a simple modification can be made to let it skip that reduction when a fixed timestep is set. This provides a best-case bound on the potential improvement of such a development.

Comparative execution performance in Figure 8.3 shows that this provides a slight improvement in our microbenchmark, that becomes more marked at larger scales. We can see in Figure 8.5 that this change sharply changes the execution structure of this code. With a reduction at each level, substantial per-level load imbalance and small amounts of communication delay are observed even at small scales. With no global reductions synchronizing execution, load imbalance is less marked and becomes easier to address with less finely-tailored mechanisms.

We can also see that this sharply changes the communication behavior of the code. With global synchronization at the end of each step, most messages between boxes were being sent during the narrow time window shortly after the reduction completes and the next step starts. Without this reduction, those same messages are generated incrementally over the course of the entire step, as shown in Figure 8.4. This reduces the peak network injection rate from 26 GB/s to 11 GB/s. By reducing peak network traffic, this transformation broadly applied can reduce the bandwidth requirements in the design of future machines. More practically, it can reduce network contention and hence network latency on recent supercomputer networks [84, 85, 86].



(a) With an allreduce in each step

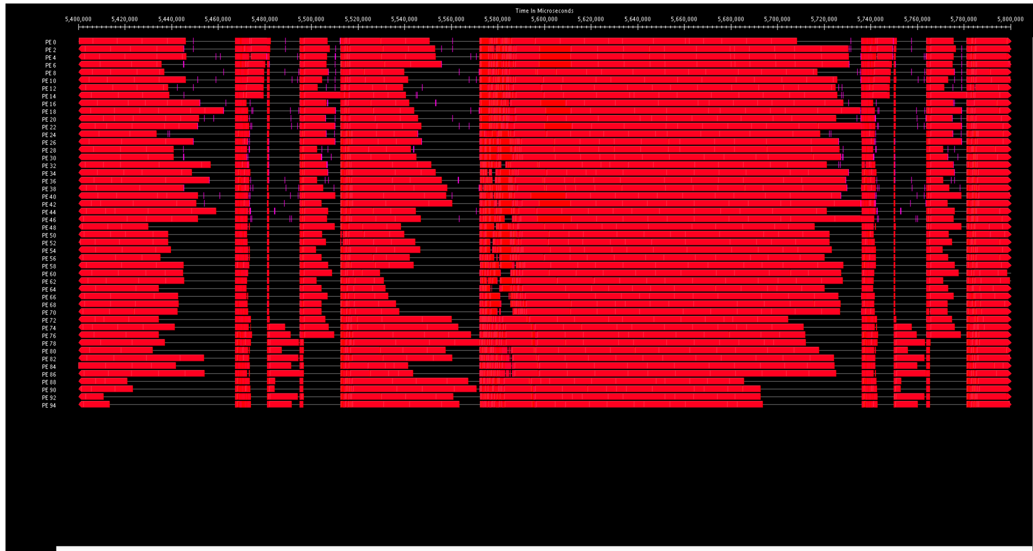


(b) Without any global collective between steps

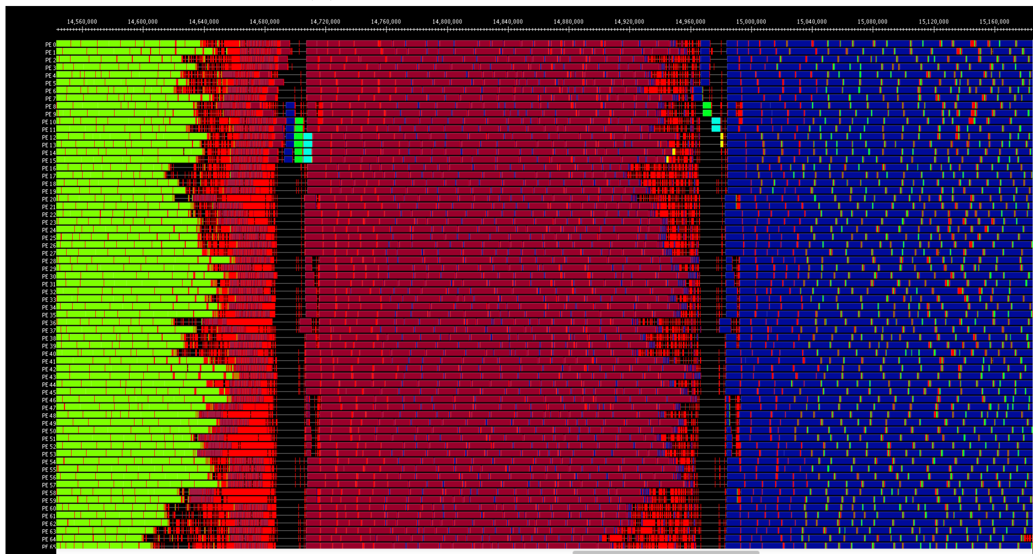
Figure 8.4: The effect of eliminating global synchronization on the distribution of network traffic over time. Eliminating each object's wait for a reduction to complete allows them to begin sending their first round of data for the subsequent step as soon as they are able to compute it. This spreads the traffic over time, causing peak message rates to fall from 26 GB/s to 11 GB/s.

8.3 Design Alternatives

Rather than constructing a fresh object/thread association, it may have been easier to use the existing TCharm framework within CHARM++ or the full Adaptive MPI implementation. Using Adaptive MPI would have additionally allowed use of its message matching and sequencing logic, rather than requiring bespoke redevelopment of the communication engine. The downside of this approach is that it would require maintaining the logic in CHOMBO to generate a sequential numbering of the boxes to map them to ranks, and hence prevent subsequent application of the Semantic Nam-



(a) With an allreduce in each level



(b) Without any global collective between steps

Figure 8.5: The effect of eliminating global synchronization on execution structure

ing pattern. An extension of AMPI to allow communicators with sparse or arbitrarily-numbered ranks would resolve that concern.

A much less invasive design would be to make Chombo's `exchange()` and `copyTo()` communication operations non-blocking, have `DataIterator` opportunistically return boxes in the order their dependences from preceding communication are satisfied, and use iteration to drive the MPI progress engine. This approach would allow asynchronous execution within each phase or level of computation, but not beyond. Without capturing the code run in the body of each iterator loop and detecting dependencies from the loop body to code following it, each such loop essentially requires a barrier at its end (as in OpenMP `parallel for` without the `nowait` directive) to ensure correctness. The compiler-based analysis and transformation in OmpSs to support dynamic task dependence inference and tracking would overcome this impediment.

8.4 Future Work

The development and results described here do not fully exploit all features of the Charm++ environment. In particular, it makes no use of object migration for dynamic load balancing nor of dynamic redistribution of tasks within shared-memory processes. The code necessary to enable use of these features has been implemented in the modified version of Chombo. However, their ramifications for the framework and application code have not been fully worked out, and hence performance results are not yet available.

Chapter 9

Conclusion

In this thesis, we have seen that the broad synchronization behavior of distributed-memory parallel programs can be a first-order concern for performance and scalability, alongside the amounts and arrangement of computation and communication. Many challenging problems in parallel computing, the subjects of years or decades of focused research, can be ameliorated or eliminated by reducing synchronization. These include load imbalance, system noise, and communication latencies.

The fundamental relationship between synchronization and the above challenges has been illustrated through a theoretical modeling effort. Relevant effects described by the model are characterized by two traits. First, they cause different processors in a parallel system to arrive at a synchronization point at disparate times. Second, which processors are delayed relative to others at those points varies between adjacent synchronization periods. In the presence of these two traits, the model shows that removing synchronization between adjacent windows allows the variable arrival times to partly or wholly cancel each other out over the course of execution. Thus, the impact of the underlying problem causing the variable timing is reduced.

Several applications presenting opportunities to restructure, reduce, or eliminate synchronization were explored. The impacts on their performance ranged from modest to momentous. In the course of this application work, several patterns for improving synchronization structure emerged. These patterns have been enumerated and characterized.

9.1 Future Work

It would be desirable to examine more applications in light of the ideas presented in this thesis. This will potentially help solve existing scalability

problems faced by those applications with less work than tackling their apparent source head on. It can also support expansion and refinement of the set of patterns described so far.

In support of further application work, it would be useful to have a software tool to indicate when synchronization is exposing other problems. This could be an online detection mechanism or library, or an offline analysis. The primary trait for such a tool to seek would essentially be a generalization of 'imbalance time' [87], in which different processes incur different amounts of idle time while waiting for a synchronizing operation to complete.

Separately, the presented performance model could be reformulated in terms of turning execution-time variations into 'noise' of effectively 'high frequency' relative to the synchronization points, such that it explicitly can be expected to balance out. This would eliminate the analytical artifact in the current algebraic formulation, that produces unrealistic patterns in the expected benefit of reducing synchronization.

References

[1] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.47 pp. 1–11.

[2] J. C. Linford, M.-A. é Hermanns, M. Geimer, D. Böhme, and F. Wolf, "Detecting load imbalance in massively parallel applications," FZ Juelich, Tech. Rep. FZJ-JSC-IB-2008-09, 2009.

[3] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[4] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.

[5] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.12 pp. 1–11.

[6] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization," in *Proceedings of SC'03*, November 2003.

[7] S. Chakravorty and L. V. Kale, "A fault tolerance protocol with fast fault recovery," in *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[8] E. Meneses, X. Ni, and L. V. Kale, “A Message-Logging Protocol for Multicore Systems,” in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[9] E. Meneses and L. V. Kale, “CAMEL: Collective-aware message logging,” 2015.

[10] J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. Kale, “Scalable Replay with Partial-Order Dependencies for Message-Logging Fault Tolerance,” in *Proceedings of IEEE Cluster 2014*, Madrid, Spain, September 2014.

[11] X. Besseron, S. Jafar, T. Gautier, and J. L. Roch, “CCK: An improved coordinated checkpoint/rollback protocol for dataflow applications in KAAPI,” in *2006 2nd International Conference on Information Communication Technologies*, vol. 2, 2006, pp. 3353–3358.

[12] L. Wesolowski, “Software topological message aggregation techniques for large-scale parallel systems,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2014.

[13] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, “TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages,” in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP ’14, Minneapolis, MN, September 2014.

[14] J.-S. Yeom, A. Bhatele, K. R. Bisset, E. Bohm, A. Gupta, L. V. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski, “Overcoming the scalability challenges of epidemic simulations on Blue Waters,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS ’14. IEEE Computer Society, May 2014.

[15] L. Valiant, “A Bridging Model for Parallel Computation,” *Communications of the ACM*, vol. 33, no. 8, August 1990.

[16] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: towns, buildings, construction.* Oxford University Press, 1977, vol. 2.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Abstraction and reuse of object-oriented design.* Springer, 1993.

[18] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for parallel programming.* Pearson Education, 2004.

[19] B. Acun, P. Miller, and L. V. Kalé, “Variation among processors under turbo boost in hpc systems,” in *International Conference on Supercomputing (ICS).* ACM, 2016.

[20] P. N. Swarztrauber, "Multiprocessor FFTs," *Parallel Computing*, vol. 5, no. 1, pp. 197–210, 1987.

[21] P. Miller, S. Li, and C. Mei, "Asynchronous collective output with non-dedicated cores," in *Workshop on Interfaces and Architectures for Scientific Data Storage*, September 2011.

[22] E. Solomonik and L. V. Kale, "Highly Scalable Parallel Sorting," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[23] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 159–168, 2010.

[24] A. B. Sinha, L. V. Kale, and B. Ramkumar, "A dynamic and adaptive quiescence detection algorithm," Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, Tech. Rep. 93-11, 1993.

[25] P. Miller, "Distributed completion detection," Charm++ Git Repository, Feb. 2011. [Online]. Available: https://charm.cs.illinois.edu/gerrit/gitweb?p=charm.git;a=commit;h=3be1a5462d4e515ae5773cdc61327dd5afe137c1

[26] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Inf. Proc. Letters*, vol. 11, no. 1, pp. 1–4, 1980.

[27] "Top500 supercomputing sites," http://top500.org, 2013.

[28] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, T. Jones, and L. Kale, "Exploring partial synchrony in an asynchronous environment using dense LU," Parallel Programming Laboratory, Tech. Rep. 11-34, August 2011.

[29] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[30] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[31] R. Alpert and J. Philbin, "cBSP: Zero-cost synchronization in a modified BSP model," in *Tech. Rept.* NEC Research Institute, 1997.

[32] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. V. Kale, and P. Ricker, "Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement," in *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012)*, New York, USA, October 2012.

[33] J. Bordner and M. L. Norman, "Using and developing Enzo-P/Cello," in *Enzo Workshop*, 2015. [Online]. Available: http://client64-249.sdsc.edu/cello/using-enzo-p.pdf

[34] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[35] J. Lifflander, P. Miller, R. Venkataraman, A. Arya, T. Jones, and L. Kale, "Mapping dense LU factorization on multicore supercomputer nodes," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2012*, May 2012.

[36] N. Bock, M. Challacombe, and L. V. Kalé, "Solvers for $\mathcal{O}(n)$ electronic structure in the strong scaling limit," *SIAM Journal on Scientific Computing*, vol. 38, no. 1, pp. C1–C21, 2016.

[37] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to parallel matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575–582, 1995.

[38] B. N. Gunney, *Scalable Mesh Management for Patch-based AMR*, Jan 2013. [Online]. Available: http://www.osti.gov/scitech/servlets/purl/1068317

[39] P. Miller, M. Robson, B. El-Masri, R. Barman, G. Zheng, A. Jain, and L. Kalé, "Scaling the ISAM land surface model through parallelization of inter-component data transfer," in *Parallel Processing (ICPP), 2014 43rd International Conference on*, Sept 2014, pp. 422–431.

[40] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *Computer Graphics and Applications, IEEE*, vol. 10, no. 4, pp. 76–82, 1990.

[41] S. A. Brown, M. Folk, G. Goucher, R. Rew, P. F. Dubois et al., "Software for portable scientific data management," *Computers in Physics*, vol. 7, no. 3, pp. 304–308, 1993.

[42] O. Sarood, "Benfits of parallelizing IO of large data-intensive applications with a case study of NAMD," M.S. thesis, Computer Science, University of Illinois at Urbana-Champaign, 2009.

[43] "I/O tips – Lustre striping and parallel I/O," http://www.nics.tennessee.edu/io-tips, retrieved 2011-03-30.

[44] R. Buch and S. White, "Parallel I/O in Charm++ and AMPI," Jan. 2016, Class project report for CS598WG.

[45] Y. Saad, "Communication complexity of the Gaussian elimination algorithm on multiprocessors," *Linear Algebra and its Applications*, vol. 77, pp. 315–340, May 1986.

[46] L. V. Kale and M. Bhandarkar, "Structured Dagger: A Coordination Language for Message-Driven Programming," in *Proceedings of Second International Euro-Par Conference*, ser. Lecture Notes in Computer Science, vol. 1123-1124, September 1996, pp. 646–653.

[47] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[48] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid MPI/SMPSs approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810091 pp. 5–16.

[49] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–10.

[50] I. Dooley, C. Mei, J. Lifflander, and L. Kale, "A study of memory-aware scheduling in message driven parallel programs," in *Proceedings of 17th Annual International Conference on High Performance Computing*, 2010.

[51] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl/.

[52] J. Choi, J. Dongarra, and D. Walker, "The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers," in *Proc. Eighth International Parallel Processing Symposium*, H. Siegel, Ed. IEEE Computer Society Press, April 1994.

[53] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. Dongarra, "Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA," University of Tennessee, Tech. Rep. UT-CS-10-660, September 2010.

[54] B. Bland, "HPC challenge class I award G-HPL winning submission," 2010. [Online]. Available: http://icl.cs.utk.edu/hpcc/hpcc_record.cgi?id=380

[55] L. Grigori, J. W. Demmel, and H. Xiang, "CALU: a communication optimal LU factorization algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1317–1350, 2011.

[56] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* IEEE Press, 2008, p. 29.

[57] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon, "Validating the FLASH code: vortex-dominated flows," in *Astrophysics and Space Science.* Springer, 2005, vol. 298, pp. 341–346.

[58] M. Norman, "Cello: An extreme scale AMR framework, and Enzo-P, an application for astrophysics and cosmology built on Cello," in *17th SIAM Conference on Parallel Processing for Scientific Computing*, 2016.

[59] A. Langer and L. Kalé, "Scalable and asynchronous algorithms for block structured adaptive mesh refinement," 2013, poster presented at HiPC 2013.

[60] D. DeZeeuw and K. G. Powell, "An adaptively refined cartesian mesh solver for the euler equations," *JCP*, vol. 104, p. 56, 1993. [Online]. Available: http://hdl.handle.net/2027.42/31037

[61] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, "Paramesh: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, pp. 330–354, 2000.

[62] M. Parashar, X. Li, and S. Chandra, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Application.* Wiley-Interscience, 2009.

[63] T. Tu, D. O'Hallaron, and O. Ghattas, "Scalable parallel octree meshing for terascale applications," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference.* IEEE, 2005, pp. 4–4.

[64] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. Wilcox, and S. Zhong, "Scalable adaptive mantle convection simulation on petascale supercomputers," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for.* IEEE, 2008, pp. 1–15.

[65] A. Wissink, R. Hornung, S. Kohn, S. Smith, and N. Elliott, "Large scale parallel structured AMR calculations using the SAMRAI framework," in *Supercomputing, ACM/IEEE 2001 Conference*, November 2001, p. 22.

[66] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, "Algorithms and data structures for massively parallel generic adaptive finite element codes," *ACM Trans. Math. Softw.*, vol. 38, no. 2, pp. 14:1–14:28, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2049673.2049678

[67] C. Burstedde, L. C. Wilcox, and O. Ghattas, "`p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[68] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, "Extreme-scale AMR," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.25 pp. 1–12.

[69] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," in *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM, 1989, pp. 247–252.

[70] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, vol. 52, no. 2, pp. 150 – 177, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731598914691

[71] Ü. Çatalyürek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1 –11.

[72] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing, ACM/IEEE 2000 Conference*, November 2000, p. 59.

[73] H. deCougny, K. Devine, J. Flaherty, R. Loy, C. Özturan, and M. Shephard, "Load balancing for the parallel adaptive solution of partial differential equations," *Applied Numerical Mathematics*, vol. 16, no. 12, pp. 157 – 182, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0168927494000395

[74] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207 – 221, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121298100341

[75] R. Löhner, "Finite elements in CFD: What lies ahead," *International Journal for Numerical Methods in Engineering*, vol. 24, no. 9, pp. 1741–1756, 1987. [Online]. Available: http://dx.doi.org/10.1002/nme.1620240910

[76] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007, pp. 15–23.

[77] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," in *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[78] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," *Journal of Computational Physics*, vol. 151, pp. 283–312, 1999.

[79] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[80] O. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," in *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, Stanford, CA, Jun 2001, pp. 21–29.

[81] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale, "Charm++ & MPI: Combining the best of both worlds," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, ser. IPDPS '15. IEEE Computer Society, May 2015, lLNL-CONF-663041.

[82] Charm++ Issue Tracker, "Define and document semantics of dynamic insertion in chare arrays wrt broadcasts and reductions." [Online]. Available: https://charm.cs.illinois.edu/redmine/issues/49

[83] "Chombo Software Package for AMR Applications," http://seesar.lbl.gov/anag/chombo.

[84] A. Bhatele, "Automating Topology Aware Mapping for Supercomputers," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, http://hdl.handle.net/2142/16578.

[85] M. Besta and T. Hoefler, "Slim Fly: A Cost Effective Low-Diameter Network Topology," Nov. 2014, proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC14).

[86] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, pp. 77–88, June 2008.

[87] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Boug, and T. Priol, Eds. Springer Berlin Heidelberg, 2007, vol. 4641, pp. 150–159. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74466-5_17