Patterns for Overlapping Communication and Computation

Aaron Becker, Ramprasad Venkataraman, and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{abecker3, ramv, kale}@illinois.edu

Abstract

Parallel applications commonly face the problem of sitting idle while waiting for remote data to become available. Even for problems where plenty of parallelism is available and good load balance is achievable, performance may be disappointing if local work cannot be overlapped with communication. We describe three patterns for achieving the overlap of communication with computation: overdecomposition, non-blocking communication, and speculation.

1 Introduction

Nearly every parallel application spends some of its time waiting for data to become available. For any problem that isn't embarrassingly parallel, there are dependencies between work done locally and work done remotely, whether that remote work is done by another local thread, another CPU, specialized local hardware such as a GPU, another node in a cluster, or some remote computing resource in a grid environment. Minimizing these dependencies is an important part of creating high performance parallel applications. However, some dependencies are inherent to the problem decomposition and can never be eliminated. In those cases, we would like to minimize time spent idle while waiting for remote data.

To minimize idle time, we must ensure that the work done by the parallel algorithm is evenly distributed across the system. If some processing elements (PEs) are overloaded with work relative to others, then the underloaded elements will inevitably have to wait for their overloaded counterparts to finish. Note that each part of a parallel system need not have an equal amount of work. Rather, work should be distributed so as to minimize the maximum time taken by any resource. Solutions with this property are referred to as *load balanced*.

However, even a perfectly load-balanced application may spend significant time waiting for remote data. The actual transfer of data in a parallel system always takes time, whether it be a over an internet connection, a fast network interconnect, or direct memory transfer across a bus. Load balancing does nothing to hide the cost of these transfers, it only ensures that no particular hardware resource is an unnecessary bottleneck in our computation.

To address the cost of waiting for the transfer of remote data, we must overlap this communication with local computation. If local work can be scheduled so that data transfers occur while independent local work is performed, a substantial source of idle time can be eliminated. However, finding enough independent local work to bridge the gap between the time when data is needed and when it is available can be a difficult problem to solve. Fortunately, many programs that successfully achieve overlap of communication and computation exhibit a few common patterns. In this paper we discuss three such patterns: overdecomposition, non-blocking communication, and speculation.

2 Overdecomposition

2.1 Intent

Enable the overlap of communication and computation without significant changes to a parallel algorithm, by dividing the computation into more independent pieces than there are PEs and overlapping data waits on one piece with work on other pieces.

2.2 Applicability

- Applications whose data are amenable to decomposition
- Applications whose performance is sensitive to dataflow latencies
- Parallel frameworks and runtimes that aim to enable higher processor utilization

2.3 Motivation

The most common way to develop parallel applications is to structure the computation by dividing the problem into p parts, where p is the number of PEs in the parallel system. Each PE works on its own portion of the problem, sending local results out to the other PEs that need them when they become available and waiting for remote data when that data is necessary to make progress on the local computation. Any idle time in such an application occurs when remote data is needed but not yet available. Sometimes this idle time can be reduced by changing the parallel algorithm to minimize communication and improve the distribution of work throughout the system. However, often this idle time is caused by the inherent structure of communication and computation in the application.

Regardless of the source of idle time, it is desirable to achieve a good overlap of communication and computation with minimal changes to the algorithm. Preserving the general control flow and communication semantics of the unoptimized code during the optimization process achieves several desirable goals:

1. Minimizes the programmer effort needed for optimization
2. Avoids introducing bugs during the optimization process
3. Avoids obfuscating program logic in the name of optimization

Rather than attempting to overhaul the application or the algorithm, it is much easier to find other useful work that a PE can do while waiting for remote data required to proceed with the original computations. The objective of overdecomposition is to recapture this idle time by performing useful work on other portions of data in an attempt to sidestep the limitations imposed by the structure of the application.

2.4 Example

Consider a simple explicit finite element simulation on an unstructured mesh. These simulations are commonly used in science and engineering fields to solve partial differential equations. The mesh consists of n elements, with $n >> p$. The mesh is partitioned, often using a graph partitioner such as ParMETIS [7]. On each partition, there is an interior region whose elements have only local neighbors, and a boundary region whose elements have non-local neighbors.

The application consists of a series of timesteps, and at each timestep the simulated physical properties of each element are updated. Each partition communicates updated information about

its boundaries to its neighboring partitions at each timestep to allow boundary elements to be updated correctly. The need to wait for updated data on the neighbors of boundary elements before continuing is a potential source of idle time.

2.5 Solution

The drawback with dividing a problem into one part per PE is that when work on that part inevitably stalls waiting for remote data, the PE is left idle. Overdecomposition addresses this issue by:

- Decomposing the problem into k partitions, with $k > p$, the number of PEs
- Assigning multiple partitions of the work to each PE
- Recapturing wait time in the work on one partition by switching to work on other partitions on the same PE

The algorithm for solving each individual piece of the problem remains essentially unchanged. Key to the actual implementation of patterns enabling overdecomposition is the ability to work on and switch between different portions of the work on each PE. Such capabilities can be implemented at different layers of the application stack:

- At a layer below the actual application code (within a parallel framework or runtime) leaving the application largely unaffected
- As part of the application in scenarios where such support is not available from existing lower layers

Irrespective of where such support is implemented, there may be some expectations of the application. Chief among these is that the application structure its data to permit the existence of multiple partitions on each PE. It seems reasonable that there be some level of data encapsulation within the application so that multiple partitions can coexist within the same address space. Other requirements can depend on the layer at which support is implemented and the actual implementation specifics.

2.5.1 Overdecomposition with framework support

In some parallel programming environments, support for overdecomposition is built in and trivial to take advantage of. For example, applications built on the Charm++ parallel runtime system [6] will have a large number (k) of independent objects, each performing a portion of the overall computation and communicating with other such objects. Several such objects exist on each PE and their executions are controlled by a common scheduler. Each object is given a chance to perform work only when data meant for that object becomes available. This same functionality is also packaged in an implementation of the MPI standard called AMPI (Adaptive MPI) [5], where the number of MPI processes (k) is decoupled from the number of PEs (p) and it is recommended to overdecompose the computation to achieve high performance.

2.5.2 Implementing overdecomposition support

For parallel programs which do not have support for overdecomposition built in to their programming model, realizing the benefits of overdecomposition involves more effort. For example, MPI

applications have one process per PE, and the MPI programming model does not address the issue of multiple threads of execution on a single PE, so any overdecomposition capabilities in traditional MPI programs must be built up separately on top of MPI.

Typical MPI applications do not differentiate between a partition and the compute resource (PE) that works on it as there are only as many partitions as there are PEs. To enable overdecomposition the application has to rid itself of this mindset and embrace the possibility of multiple partitions on each PE. The application may also have to review how its computations are packaged to clarify the data dependencies. That is, computations that depend on some remote data will need to be identified and separated so that they can be triggered as a response to the availability of this data.

An implementation must deal with issues such as:

- managing the partitions on each PE and performing the appropriate bookkeeping
- polling the underlying communication layers (library) for messages
- handling communication routines like barriers and reductions which still function on a per PE basis
- implementing functionality to switch to another partition when blocked
- implementing scheduling strategies for the pieces on each PE

For these reasons, lower level support for overdecomposition is extremely beneficial. Adding overdecomposition at the application level without any library support may require significant experience with parallel frameworks and represent a distraction from work at the application level.

2.6 Discussion

Overdecomposition is closely related to virtualization, a process in which a single hardware or software resource masquerades as multiple resources. Virtualization can lead naturally to overdecomposition when each virtual resource participates in an application, presenting the appearance of more PEs than actually exist and thereby placing multiple problem pieces on each physical PE. The terms "overdecomposition" and "virtualization" are sometimes used interchangeably, but virtualization more commonly refers to the case where the apparent computing resources do not match the physically available resources, whereas overdecomposition more broadly refers to any situation in which the problem is broken into more pieces than PEs and the PEs can switch between multiple pieces.

In the case of the finite element example application, overdecomposition means switching from one mesh partition per PE to multiple partitions. Then while one partition waits for updated values from its neighbors, another can be computing its own local values. However, this advantage comes with a tradeoff: the amount of communication that needs to take place at each timestep is a function of the length of all inter-partition boundaries. By increasing the number of partitions, we increase this boundary length and therefore also increase total communication. There is a tradeoff to be made: more partitions means more opportunities to overlap idle time on one partition with useful work on another, but it also means more communication.

2.7 Forces

Framework support Framework support for overdecomposition can reduce adoption costs dramatically. Once the framework provides the basic mechanisms of managing a single compute

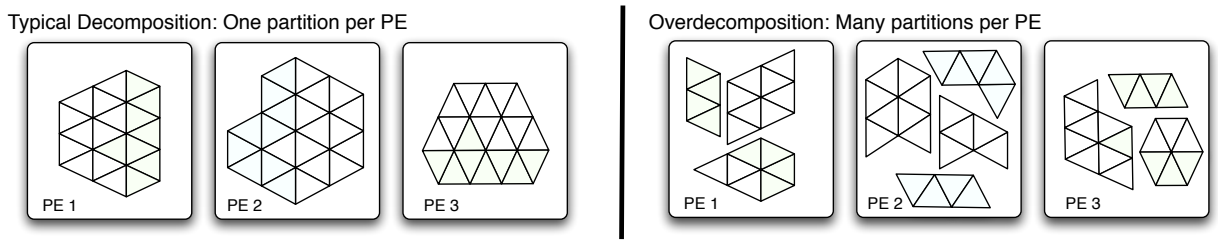

Figure 1: In a finite element application that has not been overdecomposed, each PE is responsible for a single partition. Overdecomposition places multiple partitions on each PE, so that when one partition must wait for remote data, another partition can be worked on.

resource for multiple partitions of the work, the requirements on the application are greatly reduced

Changes to application code Although overdecomposition does not require any fundamental restructuring of the parallel algorithm, it can require application changes. For example, global variables that work perfectly well when there is only one piece per PE may no longer function as intended if multiple pieces share the same address space. Refactoring to fix conflicts between multiple pieces trying to use the same global variables may not be particularly difficult, but it does impose a burden on the programmer.

Available parallelism The fundamental requirement for overdecomposition is the ability to find enough parallelism. Overdecomposition is most useful when the problem can easily be broken down into a large number of small, mostly-independent pieces. Even for problems that can easily be overdecomposed, splitting the problem into a large number of pieces may prove costly. In the case of finite element applications we saw that increasing the number of partitions also increases total communication. Any application in which there is a boundary region that requires communication and an interior region which does not will experience the same effect. In addition, there is context switching overhead associated with moving from one overdecomposed piece to another. The exact cost of this overhead will vary based on implementation, but one must be careful to amortize any context switching overhead by ensuring that each piece is sufficiently expensive computationally.

Data encapsulation Sometimes the assumption of one partition per PE can run very deep in the application. Such an assumption is more difficult to overcome if the application follows certain coding practices like putting all variables in global scope, using procedural approaches to structuring the computation and communication throughout the application etc. Eliminating the assumption that each PE has one and only one partition can take considerable effort depending on the software engineering practices employed by the application.

Flexibility requirements Overdecomposition offers increased flexibility to the application user by decoupling the problem decomposition from the hardware. For example, in situations where the application requires a square grid of processes, overdecomposition allows the utilization of all available hardware when otherwise the application would be required to use a perfect square for the number of PEs. This flexible hardware mapping also has benefits when trying to improve load balance. By moving pieces of work away from overloaded PEs to underloaded PEs, load is more evenly distributed throughout the system. However, for appli-

cations with good static load balance and no natural dependency on the number of partitions, these benefits are moot.

Algorithmic improvements Overdecomposition can reclaim idle time without overhauling the underlying parallel algorithm. However, a poor algorithm that incurs unnecessary costs will not be fixed by overdecomposition. Overdecomposition is best used to tune and improve a good parallel algorithm rather than to fix a poorly designed one.

3 Non-blocking Communication

3.1 Intent

Perform useful computations in a parallel application without waiting for a communication process to complete.

3.2 Applicability

- Algorithms where communication is interspersed amongst the computations
- Applications where some computations are independent of the data to be transferred

3.3 Motivation

A naïve approach to communication will produce suboptimal results in many parallel programming environments. Blocking communication primitives, which do not complete until all data has been successfully transferred, are often the default because of their simplicity.

Consider an application in which work that is dependent on local sends and receives is interspersed with independent computations. How can we reduce the time spent waiting for the sends and receives to complete before performing the subsequent dependent tasks?

The most direct approach to overlapping communication and computation is to explicitly identify places in the application where sends and receives could be initiated prior to doing work that is independent of that data. These places are a natural target for overlapping communication and computation because we have identified independent work in the structure of the application. The difficulty arises in finding the regions where independent work exists.

3.4 Example

Suppose we are writing a parallel conjugate gradient solver for Poisson equations in a two dimensional space. Each process is responsible for one block of a large matrix, computing the local part of a matrix-vector product. The solver performs many iterations, and in each iteration there are four steps:

1. Update the interior of the local matrix block by multiplication with local vector elements.
2. Compute partial results for elements on the local block border.
3. Sum partial results with partial results from neighboring blocks.
4. Evaluate the local matrix block, using the updated inner region and the summed results for the outer region.

Step 1, which requires only local data, is independent of 2 and 3, but step 3 depends on step 2, and step 4 depends on 1-3. All four steps must be completed before the iteration can finish.

MPI pseudocode for this algorithm is given in listing 1. The inner and outer matrix regions are two distinct arrays: `inner` and `outer`. The exchange of block boundary regions is done through a blocking call to `MPI_Allreduce`, using a communicator containing the neighbors of our matrix block.

Listing 1: Blocking Poisson Solver

```
1 void update(double* inner, double* outer, double* summed_outer,
2         MPI_Comm neighbors_comm) {
3     compute_local_inner(inner);
4     compute_local_outer(outer)
5     MPI_Allreduce(outer, summed_outer, SIZE,
6         MPI_DOUBLE, MPI_SUM, neighbors_comm);
7     evaluate_block(inner, summed_outer)
8 }
```

3.5 Solution

The problem with blocking communication semantics is that the time spent waiting for the communication to complete could have been spent doing useful work. Fortunately, many parallel programming models provide non-blocking alternatives. These replace the single blocking call with a two step process. First the communication is initiated in a call that immediately returns control to the programmer without waiting for the data to be transferred. The programmer can later test to see if the communication has completed or block until all transfers are complete once all independent local computations are complete. In MPI, for example, rather than using the blocking `MPI_Recv` call to receive incoming data, you would use `MPI_IRecv`, followed by `MPI_Test` to check for completion or `MPI_Wait` to block until the communication has completed. The same principles apply to collective operations, but the advantages can be even more dramatic due to the fact that a blocking collective call must wait on all other processes, rather than just one. Here are some techniques to exploit non-blocking communication in parallel applications:

Posting receives early A common technique for exploiting non-blocking communication is to post a receive for any data that is expected as early as possible during the computations. This allows the communication library underneath the application to start working as soon as the remote data becomes available. This increases the chances that data will be ready when it is required, eliminating idle time.

Double Buffering Often codes can be restructured to create independent work from computations with dependencies. For example, unnecessary data dependencies can be eliminated through the use of a technique called double buffering. To use this technique, instead of performing local computations on a memory buffer and then updating that buffer with remote data, instead maintain two copies of the buffer, one communication copy and one computation copy. Data is transferred using the communication copy while local calculations proceed on the computation copy. When both are finished, the buffers are swapped. The application of techniques like double buffering results in an application with the appropriate structure to take advantage of non-blocking communication.

Multiple send/receives Data sends can occur only when the data to be sent has been processed and is ready. Sometimes, when the data to be sent is large enough, the send/receives can be split into two or more separate transactions. Thus sending the first portion of the data

can be overlapped with the computations for preparing the second portion of the data and so forth. However, this may be beneficial only if the extra overhead of more communication transactions is offset by the overlap that can be achieved.

3.6 Discussion

The benefits of non-blocking collectives increase with the number of participating processes but decrease with the speed of the transport mechanism, due to the reduced time spent waiting for data transfer. For example, consider two common architectures for high performance computing clusters: IBM Blue Gene and Cray XT4. Non-blocking collectives would be relatively less beneficial on Blue Gene machines because they have a specialized communication network to accelerate collectives [9], whereas on XT4 collectives do not receive special treatment. Non-blocking communication would be less beneficial on either of these machines than on a cluster connected by gigabit ethernet due to the increased bandwidth and reduced latency of the interconnects on XT4 and Blue Gene.

However, non-blocking communication can be an effective optimization regardless of how well the network performs. Because some PEs will arrive at any collective operation before others, in a blocking collective they will needlessly spend time waiting for the last PE to reach the collective. Thus, while a high-speed interconnect may attenuate the value of non-blocking communication, improved transfer rates can never eliminate the advantages of non-blocking communication.

Non-blocking communication operations are available across a wide variety of programming environments. In MPI, the most common programming model for clusters, `MPI_ISend` and `MPI_IRecv` provide non-blocking semantics for point-to-point communication, and non-blocking collectives are available as an extension to MPI-2 [9]. Unified Parallel C offers the split-phase barrier, a non-blocking global synchronization operation [2]. Non-blocking memory transfers are a common and highly useful technique for interfacing with floating point accelerator architectures such as GPGPUs and Cell, allowing data to be streamed on and off of the accelerator hardware without halting local execution [10, 1].

Returning to our example, we can restructure our solver to overlap the independent work done in step 1 with the communication in step 3. Step 4, which depends on the results of step 3, follows the completion of the non-blocking communication. Listing 2 shows the updated algorithm. The blocking `MPI_Allreduce` call is replaced by the non-blocking alternative `MPI_Iallreduce`. After initiating the communication, the independent local work is completed, and then we wait until the communication to finish using `MPI_Wait`.

Listing 2: Non-Blocking Poisson Solver

```
1  void update(double* inner, double* outer, double* summed_outer,
2          MPI_Comm neighbors_comm) {
3      MPI_Request req;
4      MPI_Status status;
5      compute_local_outer(outer)
6      MPI_Iallreduce(outer, summed_outer, SIZE,
7          MPI_DOUBLE, MPI_SUM, neighbors_comm, &req);
8      compute_local_inner(inner);
9      MPI_Wait(&req, &status);
10     evaluate_block(inner, summed_outer);
11 }
```

A detailed examination of the effects of non-blocking collectives on a conjugate gradient solver like this one was performed by Hoefler et al. [4]. They find that switching to non-blocking calls improves scalability and gives a speedup of up to 30% over the blocking equivalent.

3.7 Forces

Simplicity The advantage of blocking communication over non-blocking is its simplicity. A communication call that returns only when that communication is complete has similar semantics to a function call which returns only when its value is computed. Non-blocking calls split a send or receive, which is conceptually a single operation, into two stages which may be separated by large amounts of intervening code. In fact, optimization will tend to push the two stages farther apart in an effort to completely overlap the communication with computation. This results in a non-trivial decrease in the readability of non-blocking code.

Availability In some cases, there is simply no opportunity to use non-blocking communication. In the case of MPI collectives, there is no non-blocking option for MPI-1, and for MPI-2 non-blocking collectives are available only as a third-party library. Even where non-blocking calls are available, their actual effectiveness can vary based on implementation quality. Again using the example of MPI, network progress is not guaranteed on each send and receive when multiple non-blocking communication operations are outstanding. The effectiveness of non-blocking routines is highly dependent on hardware capabilities and the quality of the MPI implementation used.

4 Speculation

4.1 Intent

Avoid waiting for communication to verify that a pending operation is safe to execute.

4.2 Applicability

Applications which perform computations that:

- have an irregular parallel structure
- have dynamic dependencies that are not guaranteed to be satisfied
- have dependencies that fail unforeseeably, but at a mangeably low rate

4.3 Motivation

Some parallel applications are characterized by irregular patterns of communication. Typically these are applications that are difficult to decompose, with unpredictable patterns of dependencies. A common theme in these applications is the need to verify that a pending operation is safe to perform. In the case of parallel discrete event simulation (PDES), that means bounding the minimum timestamp among outstanding events to verify that the next event to process will not violate causality. In an unstructured meshing code it may mean acquiring locks on a region of the mesh to be modified.

In practice, for many applications these checks almost always indicate that it is safe to proceed, and the operation can then be completed. However, if the operations are not computationally expensive, the cost of the checks may rival or even exceed the cost of the actual operations. Thus, the rare occurrence of unsafe operations is imposing a high cost on the execution of every operation.

Speculation is a technique that can overlap the execution of code that is not guaranteed to be safe with communication that will determine the safety of the code. If the code was safe, then we

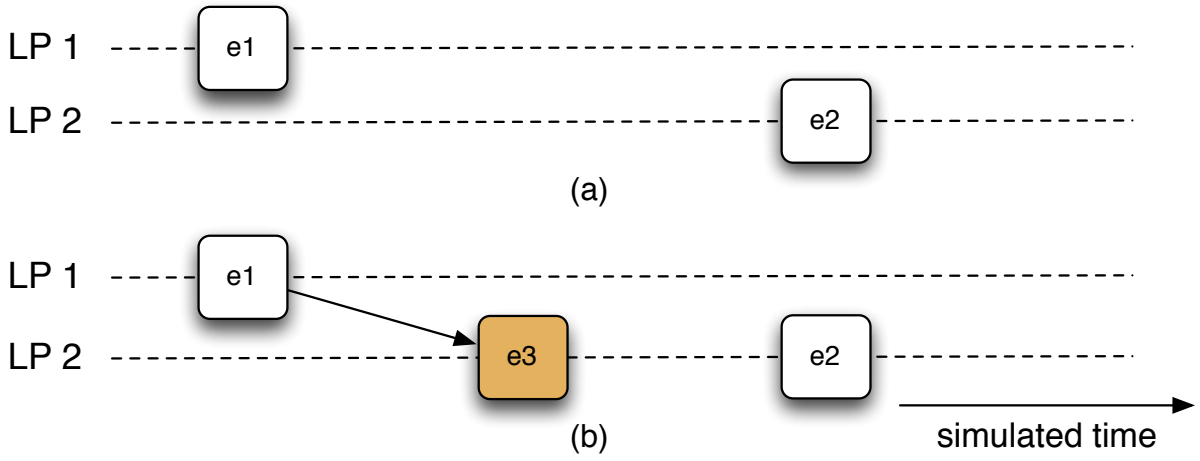

Figure 2: In (a), it is not safe for LP2 to process its earliest available event, E2. LP1 can still create events that must be processed before E2, as shown in (b). To guarantee that E2 is safe to process, communication with LP1 is needed.

avoid the cost of waiting for the communication. If not, then the effects of the improperly executed code are rolled back.

4.4 Example

Parallel discrete event simulation (PDES) is a technique that applies to a wide variety of simulations. A discrete event simulation models some real system that is composed of distinct interacting processes. Each of these physical processes is modeled by a *logical process* (LP), with interactions between physical processes modeled by sending messages between the LPs. The work done by each LP consists of a series of events, each of which occurs at a particular simulated time, indicated by a timestamp. These events update local state variables and may create new dependent events either locally or on another LP. Each LP must process all of its events in timestamp order to prevent causality violations.

This application seems naturally suited for a parallel implementation. Each PE can be responsible for one or more LPs, processing local events in order and sending the messages generated by local LPs to remote LPs when needed. However, processing the earliest available local event is not necessarily safe. The processing of an event on a remote LP can generate an event with an earlier timestamp, as shown in figure 2. This violates causality, and cannot be allowed.

Because an LP cannot determine whether or not its earliest available event is safe to execute based on purely local information, some synchronization must take place between LPs to ensure correctness. In *conservative* synchronization, events which are not guaranteed to be safe must block until communication with other LPs that determines a lower bound on timestamp of messages that can be completed safely. Depending on the structure of the simulated system, the cost of synchronization may be high relative to the amount of work done in an event, leading to large amounts of time spent in communication when no local work is being done. This synchronization operation cannot be easily overlapped with local computation, because only events that were already proven safe to execute in the previous synchronization phase can be safely processed.

4.5 Solution

Since waiting for verification that each operation is safe to execute is too costly, we apply the philosophy of "it is easier to ask for forgiveness than permission". For each operation we begin by initiating all communication needed to verify safety. We then perform the operation without any guarantee of safety, overlapping the computation of the operation with the communication needed for the safety guarantee. We then wait for the outcome of the safety check. If the operation was safe to perform, then we can simply continue on. Otherwise, we should not have performed the operation and must undo it.

There are two major approaches to undoing operations: *checkpointing* and *anti-code*. Checkpointing is a general technique for reverting a program to a known good state, and is used widely outside the context of speculation. It works by recording all relevant local state information prior to performing the operation. Rolling back an operation is simply a matter of restoring all state from the checkpoint. If the operation does not need to be rolled back, the checkpoint information can simply be discarded. Checkpointing is generally quite simple to implement, but it imposes memory overhead equal to the size of the program state that is being modified.

Anti-code is a less general technique than checkpointing, but it avoids the memory overhead of storing pre-operation state. To implement an anti-code approach, the programmer creates an anti-operation counterpart for each operation function that can be rolled back. The anti-operation undoes all of the operations performed by the original operation, leaving the program state as it was before the operation was initiated. Because the initial state of the program does not need to be recorded, anti-code techniques avoid the memory overhead of checkpointing. However, this technique forces the programmer to write an additional function for each operation to undo the actions of that operation. In general, it may not be possible to create an anti-operation for each operation, or the anti-operation may be prohibitively costly. In these cases checkpointing is the preferred approach. However, even checkpointing is not a general purpose solution. The operation may perform actions that cannot be rolled back, such as controlling the physical movements of an attached robotic arm. In these cases, rollback is impossible regardless of the technique used, and so speculation cannot be applied.

In the case of PDES, the alternative to conservative synchronization is called the *Time Warp* algorithm, which uses speculation to avoid some of the costs of conservative synchronization. In Time Warp, each LP optimistically assumes that processing its earliest available event will not produce any causality errors. Therefore, each PE remains busy as long as there is some local LP with unprocessed events. Messages to spawn events on remote LPs are overlapped with local execution because the safety checks that ensure that causality constraints are satisfied have been eliminated.

To ensure correctness, Time Warp allows for rollback when causality violations occur. Each LP keeps its events in a queue ordered by timestamp. Rather than discarding events after they are processed, the state of the local LP is saved, and any messages the event generated are recorded. Then the next event in the queue is processed. Whenever the LP receives a message from another LP, the event is placed into the local queue. If the new event's timestamp comes after all locally processed events, then execution proceeds normally. Otherwise, a causality violation has occurred, and the LP needs to roll back. The LP rolls back its causality-violating events in reverse order, restoring its local state and sending a rollback message for each remote event that the causality-violating event generated. Once all of these events have been rolled back, execution can start up again. For a detailed description of the mechanics of both conservative synchronization and Time Warp PDES, see Fujimoto [3].

Speculation can be applied to any problem domain where rollback is possible and where opera-

tions must be guaranteed safe before they can proceed. The Galois system [8] provides a software framework for speculative execution and rollback for such systems, and has proven effective for problems with amorphous parallelism such as Delaunay mesh refinement and BK image segmentation.

4.6 Forces

The critical question when evaluating speculative algorithms like Time Warp is whether the benefits of avoiding safety checks outweigh the costs of rolling back incorrect speculation.

Cost of correctness Speculation is only worthwhile when there are large penalties associated with ensuring correctness. In cases where there is ample parallelism available and plenty of work that is known to be safe, then the benefits of speculation are small. Typically the gains offered by speculation must be very large so that they overwhelm the cost of performing rollbacks and provide a net benefit to the application.

Frequency of failures Speculation is only beneficial when the benefits of optimistic execution outweigh the costs of rolling back on failures. The failure rate of speculative operations must be low enough that the rollback cost does not overwhelm any gains from speculation.

Cost of rollback Together with the frequency of failures, this defines the cost of speculation. This cost is felt both in time spent in rollback and the space required to store checkpointed state for possible rollbacks. The larger the state of the application, the more expensive speculation will be. This cost is not offset by a low failure rate, as the amount application state that must be stored does not depend on whether or not the stored state will ever be used.

5 Conclusion

Once an application has been effectively parallelized, identifying opportunities to overlap communication with local computation is an important part of the optimization process. Without this overlap, even applications with ample parallelism and good load balance may perform poorly due to excessive time spent idle while waiting for the results of communication. We have presented three approaches to allow local work to proceed while communication completes: overdecomposition, non-blocking communication, and speculation. These approaches are used in a broad range of high-performance parallel applications, and can reduce the costs associated with waiting for remote data in a variety of contexts.

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, and J. Sheaffer. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, Jan 2008.

[2] W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained upc applications. *Parallel Architectures and Compilation Techniques*, Jan 2005.

[3] R. M. Fujimoto. Parallel and distributed simulation systems. page 300, Jan 2000.

[4] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Computing*, Jan 2007.

[5] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[6] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[7] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM*, Nov 1996.

[8] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Feb 2009.

[9] J. Moreira, G. Almasi, C. Archer, and R. Bellofatto. Blue gene/l programming and operating environment. *IBM Journal of Research and Development*, Jan 2005.

[10] J. Sancho and D. Kerbyson. Analysis of double buffering on two different multicore architectures: Quad-core opteron and the .... *IEEE International Symposium on Parallel and Distributed ...*, Jan 2008.