Topics in Parallel and Distributed Computing:

Introducing Concurrency in Undergraduate Courses[1,2]

Chapter 6

Networks and MPI for Cluster Computing

Ryan E. Grant, Stephen L. Olivier

Sandia National Laboratories[3]

regrant@sandia.gov, slolivi@sandia.gov

[1]How to cite this book: Prasad, Gupta, Rosenberg, Sussman, and Weems. *Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses*, 1st Edition, Morgan Kaufmann, ISBN : 9780128038994, Pages: 360.

[2]Free preprint version of the CDER book: `http://grid.cs.gsu.edu/~tcpp/curriculum/?q=cedr_book`.

[3]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND Report Number 2013-10652P.

TABLE OF CONTENTS

LIST OF TABLES **184**

LIST OF FIGURES **185**

CHAPTER 6 NETWORKS AND MPI FOR CLUSTER COMPUTING **186**

6.1 Why message passing / MPI? 188

6.1.1 Shared memory cache coherent non-uniform memory access (cc-NUMA) architecture does not extend to extreme scale 191

6.1.2 Message Passing Provides Scalability for Grand Challenge Problems 193

6.1.3 Single Program Multiple Data (SPMD) model enables reasoning about and programming O(million) process programs 196

6.1.4 High Performance Computing (HPC) in the cloud compared to traditional HPC 199

6.2 The message passing concept 200

6.2.1 Point-to-point communication 200

6.2.2 Introduction to collective communication 206

6.2.3 Bulk Synchronous Processing (BSP) model illustrated with a simple example 211

6.2.4 Non-Blocking Communication 213

6.3 High Performance Networks 216

6.3.1 Differences from commodity networks 217

6.3.2 Introduction to Remote Direct Memory Access (RDMA) 219

6.3.3 Main Memory and RDMA 221

6.3.4 Ethernet RDMA 222

6.4 Advanced concepts 222

6.4.1 Hybrid programming with MPI and OpenMP 222

6.4.2 Supercomputers: Enabling High Performance Computing 224

REFERENCES **239**

LIST OF TABLES

Table 6.1 Historical Supercomputer Performance 226

LIST OF FIGURES

Figure 6.1 Traditional HPC compared to the Cloud HPC Model 228

Figure 6.2 Point-to-point Communication Example. 229

Figure 6.3 Point to Point protocol switching from Eager to Rendezvous . . . 230

Figure 6.4 Collective operations for message passing programs. 231

Figure 6.5 A BSP superstep. 232

Figure 6.6 Example division of grid points among processes in the BSP temperature program. 233

Figure 6.7 Computation-Communication Overlap Execution Comparison . . 234

Figure 6.8 Comparison of networking stacks - traditional vs. OS bypass . . . 235

Figure 6.9 Remote Direct Memory Access Overview 236

Figure 6.10 The queue pair model 237

Figure 6.11 Hybrid programming using threads and MPI 238

CHAPTER 6

NETWORKS AND MPI FOR CLUSTER COMPUTING

Ryan E. Grant, Stephen L. Olivier

Sandia National Laboratories[1]

regrant@sandia.gov, slolivi@sandia.gov

ABSTRACT

Cluster computing, message passing, and high performance networks form the foundation for high performance computing and modern supercomputers. This chapter will introduce the reader to the key concepts needed to understand how cluster computing differs from other types of distributed computing and provides a brief introduction to supercomputing. The material presented here is intended to be introduced in core computer science courses, alongside other parallel and distributed computing concepts. This information can be easily integrated into discussions of shared memory parallelism, and it serves as an example of methods for extreme parallelism that are needed for supercomputing.

Relevant core courses: CS2, DS/A, Systems

Relevant PDC topics: Cluster Computing, Message Passing, SPMD, BSP, Communication Algorithms, MIPS/FLOPS, LinPack, Distributed Memory, PGAS Languages, Synchronization

Learning outcomes: Students will become aware of the existence of message passing

[1]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND Report Number 2013-10652P.

paradigms for cluster computing. They will understand the concept of collective communication and the needs for unique identifiers for each computational process. The concept of supercomputing and the unique concerns of operating large scale systems will be introduced.

Context for use: These topics should be introduced along with shared memory parallel computing as a replacement for or complement to shared memory. They can be used to complement any discussion of machine parallelism by representing that individual computing nodes can then be leveraged for further multi-node parallelism.

This chapter offers insights into the practice of cluster computing and supercomputing for instructors, while covering the technical details needed to explain the concepts to CS/CE students. It includes guidance on how and when to introduce these topics to students within the PDC curriculum, as well as some suggested classroom activities that can be used to enhance student classroom participation.

This chapter begins with an introductory essay explaining the key concepts of message passing in cluster computing, along with some common misperceptions. This essay focuses on a real-world application context, relating the theoretical concepts to examples of practice that can be used in a classroom setting.

The material following the essay will detail additional technical information needed to teach cluster computing topics at a core CS level. This material will cover all the required knowledge for an instructor aiming to provide a Bloom classification of K to C for cluster computing and message passing concepts in their core CS courses. We have also provided an advanced concepts section that briefly discusses topics best introduced in advanced level elective courses along with the relevant technical information needed to teach such subjects. Some of this advanced material may also be introduced by the instructor for enrichment proposes in core level courses.

6.1 Why message passing / MPI?

Vocabulary: Node - An individual computing system that is part of a cluster (e.g., a rack server or a workstation)

Non-Uniform Memory Access - A memory architecture configuration in which costs to access different parts of memory are not equal.

(MPI) The Message Passing Interface - A standard way of exchanging data (via messages) between cooperating compute nodes.

Message passing is the current standard programming model for multi-compute node parallel computing systems. The idea of message passing has existed for many years. The idea of message-based networks formed the basis for what we now know as the Internet. The

Internet and other wide area networks work using packetized messages, which are interpreted as either a stream of data (Transmission Control Protocol (TCP)) or as independent messages (User Datagram Protocol (UDP)). The Message Passing Interface (MPI), is a messaging middleware that provides a standard way of communicating via messages with a given group of communication partners. It defines what form the messages passed should take, and allows the target of the messages to determine where the message was sent from, whether it has been delivered in order, what process it was meant to be delivered to, and where it should be placed in memory when it arrives. It provides the features needed to develop very large parallel code using efficient message exchanges that can even be used between compute nodes with different architectures (e.g., big Endian versus little Endian). It provides collective operations, like broadcast, gather, and scatter, which allows for efficient communication of computational results for parallel code. MPI is supported on many different platforms, and by many different vendors, providing a single interface for code development that is portable between many different types of systems.

Message passing techniques prior to MPI were not standardized, and several methods existed. This led to code portability issues, as not all methods were supported on all platforms. For example, some platforms supported Parallel Virtual Machine (PVM) middleware while others supported independently developed message passing libraries like LAM (Local Area Multicomputer). Several other message passing middleware packages also existed, which while incompatible, shared the same goal of providing a message passing interface. Starting in the mid 1990s the developers of these message passing approaches decided to work together to combine the best aspects of their solutions into a standard. Many industry vendors and academic researchers came together to publish the first MPI standard in 1996. MPI-1, as it came to be known, was popular for many years, and was widely adopted. However, the standard was not perfect, and as it aged it failed to address some of the state of the art work being done on high performance networks. The MPI standardization body was rejuvenated with new members and began to meet again to discuss a new MPI version, MPI-2. Continuing work on MPI by many different groups culminated in the release of

MPI-3 in 2013. MPI has been embraced by the high performance computing community as the premier method of inter-node communications, and runs on virtually all of the top supercomputers in the world today.

The platforms that run MPI are typically a cluster of homogeneous systems (nodes) that work cooperatively on large computational tasks. Many of the top supercomputers in the world as of this writing use MPI to communicate between many computational nodes, connected together with very high speed networks (some custom built), while using readily available enterprise class server components. Supercomputers can use a combination of typical x86 general purpose CPUs as well as lighter weight accelerator cores, typically a graphics processing unit (GPU) or light weight general purpose cores.

The bulk synchronous parallel (BSP) programming model is fully supported by MPI. This model has been used for many years as the standard for large-scale scientific computational code. It allows many processes to compute sections of a problem solution with the computation progressing in a series of "supersteps", each ending with a communication amongst all of the participating processes (even if it is just a barrier to keep the processes synchronized at a coarse level).

An alternative architecture to cluster computing with MPI is a shared memory system. Designing large shared memory systems can be very difficult, and designing a shared memory system that contains millions of cores would be impractical. The MPI programming model combined with cluster computing offers a much more economical solution to large scale processing, as it can take advantage of commonly available servers, which can be supplemented with high performance networking hardware. The use of high performance networks is key to many computational problems as low latency, high bandwidth MPI communication is key to good performance for many MPI programs.

MPI is a commonly accepted method of message passing in between nodes of a cluster. However, it may not be the perfect solution for intra-node communication/concurrency. Shared memory optimization for MPI processes is a well researched topic, but MPI is not ideal for dynamically adjusting program concurrency during computation. In order to benefit

from many parallel threads of execution during specific parallel regions of code, other multi-threading techniques can be used. This combination of a local concurrency method and MPI is referred to as MPI+X. At the time of writing the exact method of local concurrency that will be used in future supercomputers is unknown. Some examples of current methods include OpenMP and Cilk.

Given the relationship between MPI and shared memory local concurrency methods, MPI can be introduced alongside discussions on shared memory. It can also be discussed in the context of networks, especially if the students have already been introduced to shared memory and local concurrency. This can be a motivation for why high speed networks are such an important component of large-scale systems. MPI can also be addressed whenever process synchronization is discussed, with the networking delay being accounted for, and the ramifications that this has on the performance of synchronization operations. Finally, a very appropriate time to discuss MPI is in DS/A when discussing collective operations (broadcast, multicast, scatter/gather).

The basic concepts of message passing can be illustrated to students using examples from their own lives (instant messaging, passing paper notes, email when working in a group). These examples typically represent point-to-point communications (one node to another). A proven, effective method of teaching collective operations (in the context of MPI or not), is to have students perform the jobs of actors in the system. For example, to illustrate a broadcast or multicast, the instructor can hand out small pieces of paper to some students. A reduce operation can occur by collecting the pieces and computing their sum. This activity provides a great visual reference and students typically have a good retention of the ideas, particularly if they are involved.

6.1.1 Shared memory cache coherent non-uniform memory access (cc-NUMA) architecture does not extend to extreme scale

Distributed shared memory schemes (e.g., Cluster OpenMP) are difficult to scale to large systems. This is due to the overhead of keeping the many individual processors in the

system observing the same memory space. For example, when accessing shared memory in a distributed scheme, one must make sure that the memory being accessed has not been updated elsewhere. This requires that either all memory updates must be pushed to all of the processors, or that some method be used to determine if the shared memory being read is current (i.e., polling other processors that can access that memory). A distributed memory scheme can also lead to performance inefficiencies due to data placement. For instance, if one CPU is accessing memory at a far away location, it may significantly slow the progress of that CPU. It may be possible to migrate that shared memory to the CPU most in need of it, but this too has overhead. Distributed shared memory systems may not be able to inform an application of when memory is remote, making it difficult for the application to understand when memory accesses will be fast (local) or slow (remote). All of these factors combine to make distributed shared memory schemes difficult to design and implement. As such, many existing approaches, like Cluster OpenMP, are not widely adopted in the cluster computing/supercomputing community.

Message passing helps to avoid some of the data placement performance inefficiencies of distributed shared memory schemes by explicitly exposing the locality of data. While an application may not always know what other MPI processes are running on the same node, the locality of the memory that the MPI process is working with is generally known. Remote memory locations require explicit message passing in order to access them, and therefore the programmer is aware of data locality.

Shared memory processing for local node computing is widely regarded as beneficial for cluster computing. Unlike the distributed shared memory case, the locality of the memory to the CPUs and smaller ranges for possible memory latencies make this approach very practical. When combined with multi-threading techniques, shared memory processing within a cluster node is a powerful parallel processing tool. Approaches like OpenMP allow for the creation of many threads for a parallel portion of application code, with easy methods of indicating the parallelism required and the memory sharing parameters. Because this shared memory multi- threading model is confined to a local compute node, this scales well

as each node is essentially a separate system from the local shared memory point-of-view. Of course, shared memory can also be used by other cluster computing parallel programming interfaces such as MPI. Most MPI implementations are optimized to use shared memory to pass messages between processes on a local node. Current (MPI-3.0) implementations make it difficult to dynamically adjust the amount of parallelism while an application is running. Therefore, MPI is typically combined with other approaches like OpenMP for local concurrency with message passing between the MPI processes.

This can be conceptualized for students as workers (processes or threads) on different machines that need to exchange data at certain points in the job. A useful analogy is that of students working together in the same room that can easily discuss the problem they are working to solve and exchange data (shared memory) and that same group working with other groups that are not in the same location (distributed memory). While message passing is the dominant remote node cluster communication technique, there may be situations in which the performance penalty of current day distributed shared memory versus MPI is acceptable in order to present the whole cluster as one large shared memory computer for ease of programming.

This topic can be introduced when discussing SPMD in CS2 and DS/A, and is also relevant when discussing shared memory. This material is important to include in any discussion of distributed shared memory. An in-depth understanding of message passing is not required, but a good learning goal is to make students aware that message passing exists, and that while distributed shared memory methods exist, message passing is the dominant paradigm. It is also appropriate to re-cap in parallel programming and perhaps in advanced architecture courses.

6.1.2 Message Passing Provides Scalability for Grand Challenge Problems

Answering many of the great scientific questions of the 21st Century will require large interconnected systems that can work together. One of the keys to designing future generation cluster computing systems is extreme scalability. As core counts of large clusters

exceed one million, the overhead incurred to manage millions of simultaneous processes can become onerous. Therefore, it is important to design communication middleware that can scale to millions of processes. MPI has demonstrated that it is capable of scaling to one million cores [1]. However, scalability of communication middleware must be supported by scalable networking hardware. For example, if a typical connection-based networking transport is used, the overhead of maintaining communication state for each process to every other process can consume significant amounts of resources. Connectionless networking transports alleviate this issue, but require additional mechanisms to ensure that messages can reach their target processes, given that no or very little communication state information is maintained.

Assuming that a scalable underlying network is available, message passing allows one to logically approach application design at million process scales. Utilizing MPI ranks, each process can be assigned a portion of work easily. This approach provides scalability by allowing good application code to efficiently manage communication. Good application code minimizes global synchronization and allows for communication/computation overlap by asynchronously processing communication operations while computation is ongoing. Programming such applications is not an easy task and requires expert programming specialists that have detailed knowledge of not only programming but of lower-level software environments and even hardware. Large computing systems also require dedicated teams of technicians to keep them operational. Alternatives to traditional "big iron" systems exist, but they may not be scalable for all problems.

Need More Tightly-Coupled Systems Than Distributed Computing Efforts Distributed computing efforts that utilize the internet and individual consumer devices with "donated" compute cycles are a good fit for a subset of problems. For efforts like SETI@home [2], which uses volunteers' personal computers to look for evidence of extraterrestrial communication, it is an excellent fit. The problem they solve is very large, but the data and the processing of the data is not dependent on the state of the whole system. In

other words, a home computer can work on a piece of the dataset collected by SETI without interacting with other computers working on other pieces. The result of one computation is not dependent on the output from another. Therefore, this model fits quite well with using many home computers (even gaming consoles) connected to the Internet to perform parallel computations in a reasonable amount of time.

Many large scientific problems require that all participating processes combine the results of their calculations before proceeding to the next step of the computation. For example, consider a physics simulation of many molecules. Each molecule exerts a force on all of the others in the system. Consequently, when simulating the whole system, each molecule simulated needs to know the position and velocity of each other molecule in order to calculate its own position and velocity. This simulation proceeds in discrete "time steps". In every time step all of the computing systems need to exchange some information. If a single system falls behind, then all of the others must wait for it to complete its calculation before they can proceed with the next time step. Such a simulation is impractical to perform at great scale and speed in a loosely-coupled distributed computing paradigm.

Time delays introduced by the physical distribution of the compute systems and slow interconnection networks make the coordinated use of many distributed home computers impractical. Widely distributed computations also typically require more effort to secure, and are subject to potentially malicious behavior when they are driven by public volunteers. Different hardware architectures and system performance variations also make coordinated dependency laden computations impractical.

This material is appropriate for CS2, DS/A and systems when discussing cluster computing and clouds/grids. Students may already be aware of projects like Folding@home and SETI@home, and may even be volunteers with said projects. By introducing the pros and cons of distributed computing it can be easily demonstrated to students why large homogeneous private supercomputers are still useful.

When teaching this material, it is important to keep in mind that cluster computing applications are typically latency sensitive. Scalability in a cluster computing context refers

to the maximum speedup that can be obtained. Adding additional resources, like compute cores or entire nodes, may not increase speedup for many scientific applications, as it simply adds more processes waiting for a key resource. This is illustrative of why distributed systems over a wide area may be inherently limited in their scalability, due to the high latency of long distance networks. For example, typical network latencies over the Internet can range from 10s to 100s of milliseconds. In contrast high speed networks for cluster computers can have latencies below 1 microsecond.

6.1.3 Single Program Multiple Data (SPMD) model enables reasoning about and programming O(million) process programs

Vocabulary: SPMD - Single Program Multiple Data - A parallelism technique where many processes of a single program work on different input datasets in order to complete processing quickly.

Dividing the work on a very large problem and executing it in the same way on many individual nodes is a reasonable way to approach programming at very high process counts. This approach, called Single Program Multiple Data (SPMD), allows for reasoning about thousands, hundreds of thousands, and even millions of simultaneous processes. However, programming applications for very large process counts is not trivial. There are many additional concerns that must be taken into account when using extreme numbers of processes as functions that were previously reasonable (e.g., collective operations) become increasingly expensive as the process count rises. Synchronization between processes becomes a performance bottleneck as the percentage of execution time spent in communication between processes both for collectives and synchronization collectives like barriers increases. Shared resource contention is also a concern, in that there are many requesters to limited numbers of resources, and the amount of time waiting to use shared resources increases on an individual process basis. A good analogy to explain this concept to students is thinking about the lines at an amusement park. When there are a few thousand people in the park, everything runs smoothly, but if the population booms to a million people, everything comes to a halt quickly.

Unlike the obvious items like the capacity of the rides, other concerns come into play, like the width of the walkways. In the given analogy, the crowded walkways are equivalent to an overloaded network and the massive lines at the rides would be equivalent to the contention for shared resources such as file systems or shared memory (especially if the shared memory is distributed and needed by multiple processes at once). This is very much like the extra concerns that must be taken into account with large numbers of simultaneous processes on a computing system.

The key advantage to the SPMD model is that it allows a structured approach to performing a massive job. Although other concerns must be considered, the fundamental idea behind the approach is that a single code base can run from one process up to a million processes. This provides easy scalability from a programmatic point of view, and simplifies maintenance of the code. While a naïvely written MPI program *might* be able to be run with many simultaneous processes, its performance would be disappointing. The SPMD model provides the tools for expressing a massively parallel program but does not guarantee that said program is useful/performant.

This material is intended to introduce the students to thinking about the challenges of scaling these approaches to extreme scales. Although the answers to the questions posed to students by this material may be outside of the scope of the curriculum, they are meant to stimulate independent thought on the topic. This is an excellent topic to broach to students nearing the end of a lecture to spur student thought outside of the classroom on what they have just learned and the depth of the topic of parallelism.

Suggested questions for students to consider are:

> How do you debug a program with one million processes?
>
> *Answer: This is only possible with lightweight distributed debugging tools like the Stack Trace Analysis Tool (STAT). Such tools use lightweight tracing and distributed processing of the data. Using a 1-to-N approach is not feasible at these scales given the amount of data being processed.*
>
> What kind of applications can take advantage of such a system?

Answer: Not all applications can take advantage of massive parallelism. The applications that can best leverage massive parallelism are weak scaling applications. Weak scaling applications scale their problem size to the number of processes available, while strong scaling applications keep given problem sizes but attempt to distribute that problem over more resources. Obviously, strong scaling applications have upper limits on the amount of parallelism that is practical for a given problem. Once the problem has been sufficiently divided into small enough chunks, further dividing it is either impossible, or leads to performance issues as the amount of communication vs the amount of computation increases.

Given the size of the system, failures of individual nodes will happen, how will this affect performance?

Answer: Failures, whether they be hardware or software based, can greatly impact performance. Methods to avoid having catastrophic loss, like checkpoint/restart (which saves and restores application state), are effective at preventing complete job loss, but incur overheads. With very large systems failures can occur regularly (once or multiple times per day) and therefore checkpointing is required for large long running jobs to ensure that they can run to completion. Stopping the program execution and saving all required data and state to a persistent storage apparatus requires heavy use of the system interconnect and pauses the job during the entire operation (assuming a co-ordinated checkpointing scheme). Individual system performance impact will depend on how quickly such checkpoints can be taken (network performance and persistent storage capabilities) along with how often they need to be performed (related to how reliable the system is).

This material can be introduced when introducing SPMD in CS2, alternatively it can also be introduced when covering communication algorithms in DS/A.

6.1.4 High Performance Computing (HPC) in the cloud compared to traditional HPC

HPC in the cloud uses the commercial cloud model as a basis for the hardware used for high performance computing code. The virtualization model used in cloud computing introduces additional overhead over the traditional private cluster model. An illustration of this is provided in Figure 6.1

In addition, the minimal service level guarantees provided by the cloud model may be insufficient for an HPC application's needs. Another major performance concern for HPC in the cloud is the presence of other jobs running in the cloud that may interfere with the resource hungry HPC applications. This can also affect the performance of other jobs in the cloud as the HPC applications consume large amounts of resources, particularly network resources. Current HPC cloud implementations are stand alone systems much like a traditional cluster. HPC in the cloud requires huge amounts of resources and typically needs very high performance networking hardware. As such, creating an HPC capable cloud computer requires significant investment in high performance interconnects, and this infrastructure may not be practical for use throughout a general use cloud that spends most of its time on non-HPC jobs.

The cloud may be an excellent solution for consumers that only require intermittent or marginal amounts of HPC capability. An example of this would be small to medium sized companies that can make use of HPC for simulations, but cannot justify the large investment required for dedicated HPC hardware. The cloud also provides the benefit of offering up to date hardware, and paying only for resources when they are being utilized. These benefits and drawbacks to HPC in the cloud versus traditional HPC show that while HPC in the cloud will broaden access to HPC capabilities, some historical users of supercomputers (e.g., large corporations, government, and the military) are unlikely to move to a cloud computing platform, both for performance requirements and the sensitivity of the applications and data being used.

Cluster computing is well suited to HPC. The local nature of the hardware and the exclusivity of the system to a limited number of users is ideal for peak performance. The

main drawback to this approach is mostly economic. Cluster-based supercomputers are very expensive both to procure and operate. Supercomputers routinely consume millions of dollars of energy every year, regardless of their utilization, and require teams of highly skilled technicians to keep them operational. In many cases, traditional cluster computing remains the method of choice for the largest consumers of HPC. Smaller clusters are also accessible to many organizations, and while cloud and grid computing may be attractive in some scenarios, many groups choose to operate private cluster computers.

In this section we have described the strengths and weaknesses of several different models for high performance computing. It is suggested this material be introduced as a pros/cons discussion in CS2 and DS/A, when the approaches are first mentioned. By contrasting the benefits of each approach, the differences between them can also be discussed.

6.2 The message passing concept

The message-passing paradigm is a powerful concept for problem solving from small clusters to extreme-scale supercomputers. Two main communication mechanisms are provided: point-to-point and collective communications. Point-to-point communication allows pair-wise data exchanges, while a collective communication coordinates data exchange and synchronization across many processes in a single operation.

6.2.1 Point-to-point communication

Basic idea of Communicating Processes Vocabulary: communicating sequential processes (CSP) - a model of concurrent computation in which independent processes exchange messages to communicate with each other

A fundamental need in shared-memory programming for all but the most trivial of problems is to properly protect or order reads and writes to shared variables. The mechanisms provided by multithreading languages and libraries, e.g., locks, critical sections, and flush operations, can be difficult to use without introducing data races. The contrasting conceptual foundation for message passing is the idea of

communicating sequential processes (CSP). [2] A collection of independent processes is created, each with a distinct memory space. Two processes may run on two cores of the same multiprocessor chip (e.g., on a student's laptop), different chips on a common motherboard, or on different racks connected by cable links. Data is exchanged between processors by the sending and receiving of messages. The distinction between programming models for multithreading and message passing is of primary importance and can be discussed in the context of operating system processes and threads, or in the context of different concurrent algorithms and data structures.

Unique addresses for each process Vocabulary: Message Passing Interface (MPI) - a standard API for message passing programs, rank - a unique integer process identifier, communicator - a group of processes

To pass messages between processes, each process must have a unique identifier. Unique identifiers are established and distributed among the processes, allowing any process to communicate with any other process. In the ubiquitous message passing interface (MPI) standard, the identifier is known as the process rank. The rank space is an ordered set of non-negative integers, enabling algorithms based on communicating in a particular order or breaking symmetry by rank.

Communication groups, called communicators, can be defined in order to partition the possible destinations that processes can send to. Only processes that belong to a communicator can send to other processes in that communicator. This is a very useful concept when multiple MPI instances need to be used simultaneously. An example use case would be a library that uses MPI operating at the same time as an MPI application. In order for the library and application not to receive each others MPI messages, one can separate them out into separate communicators, ensuring that messages sent by the library will never be received by the application, and vice

[2] Students may benefit from reading the seminal paper by C.A.R. Hoare [3].

versa. Each of these communicator groups described are intracommunicators, meaning that they only communicate amongst their group. It is also possible to create an intercommunicator, one that allows communication between intracommunicators. This approach is useful to provide both private message passing between processes of a library/application and also communication between the communicator groups. The hierarchical structure of this arrangement controls how the two communicators pass messages between themselves, which allows for comprehensible communication between MPI jobs.

In some cases the need for a defined communicator is unnecessary. For example, an application may not expect to be interoperable with other message passing applications/libraries. In this case, a communicator spanning all active processes may be appropriate. MPI defines this communicator as MPI_COMM_WORLD, and in practice, many applications use MPI_COMM_WORLD as their communicator.

Send-receive pairs illustrated by example A Point-to-point communication in the message passing model requires both the initiator and target of the message to explicitly participate in the communication of messages. Because a single executable from the same source code is launched on each process, the role of each process is typically specified by indexing statements to the rank. The following is a basic example.[3]

The example code below and corresponding Figure 6.2 illustrate the classic game of telephone. The process with rank 0 comes up with a number, which is then passed to the next rank, and from then on to the next rank, and so on. Note the importance of the ordering of the receive and send calls. A given rank must first receive the number before sending it onward. There is no ambiguity in this ordering. This program requires *O(P)* time, where P is the number of processes. Challenge

[3]In this example and those that follow, a simplified API is used rather than the full MPI calls to simplify the examples and emphasize the core ideas. A later section of this chapter gives the complete forms of `MPI_Send()` and `MPI_Recv()` calls. For more information and other complete API calls, consult the MPI standard [4].

students to think about how they might write a message passing program to perform the same task in *(O(log P))* time. How long would the program take if rank 0 sent a message to each of the other ranks? While it may seem that such an approach could require only constant time, in practice this is not the case. In the section on collective communications, this communication pattern, broadcast, is shown to have a collective routine in the MPI with an efficient underlying implementation.

```
int my_rank;
int num_ranks;
int magic_number;
comm my_communicator;

init(&my_rank, &num_ranks, my_communicator);

if (my_rank == 0)
   magic_number = rand();
else
   receive(my_rank-1, my_rank, &magic_number, my_communicator);

if (my_rank < num_ranks - 1)
   send(my_rank+1, my_rank, &magic_number, my_communicator);

printf("Rank %d received magic number %d.\n", my_rank, magic_number);

finalize();
```

Using buffers for storage In contrast to shared memory multithreading programming models, message passing programs do not have shared variables. Instead, all data transfer is accomplished through the sending and receiving of mes-

sages. The data passed between processors is stored in buffers. When using buffers to receive messages, the order in which they are written into the target machine's memory must match the order in which the messages were sent. For the case of a one million or more process application, keeping buffers for each possible communicating partner is impractical. Some algorithms are designed to limit the number of communication partners. Another way to mitigate the impact of space limitations is to combine message passing with multithreading, such that several threads executing on adjacent cores communicate through only one MPI process. This hybrid parallel programming model is described further in the advanced topics section.

Point-to-point Communications and Message Size When sending point-to-point messages buffers are not kept for each communicating partner, and those buffers that are reserved cannot handle very large message sizes. Consequently, for performance reasons a number of smaller buffers will typically be maintained by a message passing implementation in order to quickly service incoming messages with pre-allocated buffers. There are specific reasons why registering memory upon message arrival is impractical for high performance networks, which are explored in more detail in Section 6.3.3.

In order to exploit these pre-registered buffers, message passing implementations can use an "eager" protocol to send messages immediately to the target without first ensuring that buffer space exists on the remote side. It is possible that no buffer space (or at least one suitable for the message) exists, and the message passing implementation would have to drop the message and negotiate with the source to allocate sufficient buffer space for the message. Alternatively, above a given message threshold, it may be advantageous not to attempt an "eager" message transmission but instead to negotiate via handshake messages with the target to determine that appropriate buffer space is allocated to the message before its transmission. This scheme is typically called a "rendezvous" protocol. Eager protocols are best suited

to small messages while rendezvous is best suited to large messages. Consequently, a reasonable message passing library implementation can use both protocols by switching from eager to rendezvous at a message size threshold determined through appropriate tuning.

Figure 6.3 illustrates a bandwidth curve that represents the protocol switching that can take place inside a message passing library. For small messages the eager curve is superior, and therefore it is used. When the rendezvous curve eventually overtakes the eager curve for larger message sizes, the implementation can switch to take advantage of the superior rendezvous performance.

Complete forms of `MPI_Send()` and `MPI_Recv()` and message tags The preceding examples have used simplified syntax for the message passing calls to focus on the basics of the paradigm. The following are the complete forms of the `MPI_Send()` and `MPI_Recv()` calls:

```
MPI_Send ( void*         data,
           int           count,
           MPI_Datatype  datatype,
           int           destination,
           int           tag,
           MPI_Comm      communicator )

MPI_Recv ( void*         data,
           int           count,
           MPI_Datatype  datatype,
           int           source,
           int           tag,
           MPI_Comm      communicator,
           MPI_Status*   status )
```

The first two arguments of the `MPI_Send()` call are straightforward: a pointer to the data buffer and the amount of data being sent. The third specifies the data type. The MPI standard defines basic data types corresponding to C primitive types such as `MPI_INT` and `MPI_DOUBLE`, but user-defined data types may alse be specified. The next arguments are the destination rank and an integer tag. The tag value may be used by the programmer to distinguish the data being sent, especially since MPI does not guarantee the ordering of messages. For example, the user may choose to use the tag value 32 to designate the result of a particular calculation. The last argument is the communicator.

The arguments of the `MPI_Recv()` call mirror those of `MPI_Send()`. The first four are the data buffer, the amount of data being received, the data type, and the source rank. The fifth is the tag, which on the receive side directs the MPI implementation to match the receive call to an incoming message sent with the specified tag value. Following the communicator is an `MPI_Status` pointer, which we will not discuss here.

6.2.2 Introduction to collective communication

Barrier synchronization Vocabulary: barrier - a synchronization operation that ensures all processes reach a common synchronization point before proceeding

One of the most important operations for a distributed process computation is the ability to synchronize all of the processes at a given point during execution. Similar to shared memory threading models, a barrier operation forces all of the processes to report that they have arrived at the barrier point in the code. After all of the processes have arrived at the barrier, the processes are then permitted to continue execution. In message passing, the efficiency of barrier implementations is critical. Students should be asked to recall the different barrier implementations from the shared memory programming chapters, e.g., centralized, tournament, dissemination,

tree barriers, and consider their implementation in the message passing model. Even with a very efficient implementation, barrier operations often limit the performance of applications that use them frequently on large scale machines.

Other types of Collectives Vocabulary: collective operation - communication routines than involve all processes

A barrier is called a collective operation because all processes, rather than a single send-receive pair, participate. Many other types of collective operations exist as well, Broadcast (one-to-all), Gather (get data from all other processes), Reduce (combines values from all processes into a single value, e.g., an addition), and All-to-all (every process broadcasts). Figure 6.4 shows examples of collective operations.

Consider the earlier example of the "telephone game", in which a value originating from one process is transmitted to all the other processes. The Broadcast collective can be used to do this transmission through a single call, as shown below.

```
int my_rank;
int num_ranks;
int magic_number;
comm my_communicator;

init(&my_rank, &num_ranks, my_communicator);

if (my_rank == 0)
   magic_number = rand();

broadcast(&magic_number, 0, my_communicator);

printf("Rank %d received magic number %d.\n", my_rank, magic_number);
```

```
finalize();
```

The arguments to the call specify the data and the process originating the value, process 0 in this case. In contrast to the earlier example using explicit send and receive calls, this program's use of the broadcast collective is more concise and places the burden of orchestrating the message passing to accomplish the broadcast on the message passing implementation rather than the programmer. A common implementation of the broadcast collective is a tree-based transmission of the broadcast data, and the implementation may be optimized and tuned for the underlying communication system architecture and the size of the data.

Now consider the problem of summing a large set of numbers stored in several files. Suppose each process sums the numbers contained in one of the files. Finally, the total sum can be calculated by combining the local sums from all processes. The following example shows an implementation of this strategy using the Gather collective:

```
int my_rank, num_ranks;
long n;
long i;
double *data;
long global_sum = 0, my_sum = 0;
long *local_sums;
comm my_communicator;

init(&my_rank, &num_ranks, my_communicator);

get_data_from_file(data, n, my_rank);

for (i = 0; i < n; i++)
```

```
   my_sum += data[i];

if (my_rank == 0)
   local_sums = (long *) (sizeof(int)*num_ranks);

Gather(&my_sum, local_sums, 0, my_communicator);

if (my_rank == 0)
{
   for (i = 0; i < num_ranks; i++)
      global_sum += local_sum[i];

   printf("The sum of the numbers is %d.\n", global_sum);

   free(local_sums);
}

finalize();
```

The example assumes that the function `get_data_from_file` allocates space for the data and populates it with values read from a file corresponding to its rank, e.g., `data7.txt`. The Gather collective places all the local sums into an space allocated on process 0, which then combines the local sums. The Reduce collective can be used to eliminate the need for the last step, as shown in the code below.

```
int my_rank, num_ranks;
long n;
long i;
double *data;
```

```
long global_sum = 0, my_sum = 0;
comm my_communicator;

init(&my_rank, &num_ranks, my_communicator);

get_data_from_file(data, n, my_rank);

for (i = 0; i < n; i++)
   my_sum += data[i];

Reduce(&my_sum, &global_sum, SUM, 0, my_communicator);

if (my_rank == 0)
   printf("The sum of the numbers is %d.\n", global_sum);

finalize();
```

Upon the completion of the Reduce collective, the global sum resides in the `global_sum` variable on process 0.

Note that some collectives are composed from simpler collectives. For example, All-Gather gets data from all processes as in a Gather operation, then sends the collected data to all processors as in a broadcast. Students can consider how collectives could be used to solve problems such as sorting or creating a histogram from a set of values.

Suggested Classroom Activity Collectives can be more easily understood by conducting a classroom exercise where students serve as proxies for processes. For example, the instructor can write numbers on sticky notes and distribute them to the students in the front row of the lecture hall. This demonstrates a broadcast.

The students can then easily illustrate an all-gather or all-reduce by having one student collect the values from the other students (and perhaps sum them).

Collective communication can be discussed in DS/A when discussing broadcast/multicast and scatter/gather.

6.2.3 Bulk Synchronous Processing (BSP) model illustrated with a simple example

Vocabulary: Bulk synchronous processing (BSP) - a model of parallel computation in which at each step all processes compute independently, then communicate, then synchronize

Many parallel applications follow a Bulk Synchronous Parallel model: they compute and exchange data during a "superstep" and then synchronize through a barrier. Figure 6.5 illustrates a BSP superstep. The need for synchronization can be illustrated to students through the following example. Imagine that the interactions of many molecules are simulated, and such a simulation is done in time steps. At the end of each time step, the forces on each molecule determine its direction and velocity. The direction and velocity of each molecule influence the forces acting on all of the other molecules. To proceed to the next time step in the simulation, all of the data about each molecule's velocity and direction need to be exchanged amongst the processes, and only then do they have the data required to proceed to the force calculations in the next time step.

The example code below represents an extremely simplified BSP program. It computes temperatures at a given point in a grid from the neighboring temperatures. The grid points are divided among the processes, as shown in Figure 6.6. The code computes the values for all of the points in the process's assigned portion of the array and then exchanges information about temperatures at its array boundaries with other processes. At the end of each iteration the processes synchronize through a barrier call to ensure that all of the processes have successfully exchanged boundary temperature data before proceeding to the next time step. It is worth noting that

this example is extremely simplified and the complexity in the compute_temp() and exchange_boundary_temps() functions should not be overlooked.

```
int my_rank;
int num_ranks;
comm my_communicator;
int temperature[][];
int num_temps_x, num_temps_y;
int time_steps;
int requested_time_steps;

init(&my_rank, &num_ranks, my_communicator);
temps_per_rank = num_temps_x/num_ranks;

for (time_steps=0; time_steps < requested_time_steps; time_steps++) {
  /* compute temperatures for each point using surrounding points */
  for (x=temps_per_rank*my_rank; x < temps_per_rank*my_rank + temps_per_rank; x++) {
    for (y=0; y < num_temps_y; y++) {
      compute_temp(temperature[x,y]);
    }
  }
  /* exchange the results that might be needed by other processes */
  exchange_boundary_temps(&temperatures,my_communicator);
  /* synchronize all of the processes before continuing to the next iteration */
  barrier(&my_rank, my_communicator);
}

printf("done temperature simulation");
finalize();
```

This topic should be covered in DS/A as per curriculum.

6.2.4 Non-Blocking Communication

This section has thus far presented only blocking/synchronous communication. Blocking means that when a receive call is made, the API does not return until the request is fulfilled. Therefore, when the message passing library returns a result to the application either the target has received the message, or a message has arrived, depending on whether the call is a send or a receive. An alternative type of communication uses non-blocking or asynchronous communication. Such calls are typically denoted in MPI as Isend and Irecv, as opposed to the usual send and recv calls. Asynchronous communication requires that there is some method of achieving independent network progress.

Independent communication progress can be implemented in a variety of ways. First, the communication can be completely offloaded to hardware. In this case, the program can initiate communication through API calls which immediately return. The program can then inquire about the completion of outstanding communication operations with other progress checking API calls. This same basic concept can also be used for a software solution to communication progress. In the software case, the progression is handled by a separate execution thread. There can be a progression thread for each main process, or the progression thread can be shared by multiple main processes.

Asynchronous communication is important for performance, as one can avoid the idling of processes as they wait for communication to complete, doing no useful work while waiting. By returning from communication calls immediately, before the actual communication has completed, the CPU can continue to do useful work during the time period that the communication is taking place. This requires some extra care when programming applications that can use non-blocking communication. At some point the application will need to check if outstanding communication

operations are complete, and, should the operations not be complete, may still have to wait for the communication to complete before proceeding. For example, consider the case illustrated in the code below. In this code, the objective is to compute the sum of all of the elements in an array, and then add randomized noise to a different local array. In order to parallelize this operation, several different processes work on their own sections of the arrays. Once the summation is complete, the results are sent to the root (rank 0) process for final summation. During this final summation, the other non-root nodes would sit idle during communication. However, using asynchronous communication they could be doing useful work on the other array. Once the second array work is done, the processes need to use the result of the first array summation. Therefore they need to wait until the asynchronous receive of the results is complete. Should the computation have exceeded the time needed for the communication to complete, the wait function will return immediately. It is also possible to use test functions, which are similar to wait except that they return immediately. This may be used to compute only while a communication is outstanding.

```
int my_rank, num_ranks;
comm my_communicator;
long n;
long i;
double *data;
double *data2;
long global_sum = 0, local_sum = 0;

init(&my_rank, &num_ranks, my_communicator);

get_data_from_file(data, n, my_rank);
```

```
for (i = 0; i < n; i++)
   local_sum += data[i];

send(root, my_rank, &local_sum, my_communicator);

irecv(root, my_rank, &global_sum, &request, my_communicator);

/* Add a small random number to our second array elements */
for (i=0; i < n; i++)
   data2[i] += rand() % 100000;

wait(&request);

/* Now work with the returned global sum */

for (i=0; i < n; i++)
   data2[i] += global_sum;

finalize();
```

The example for computation-communication overlap above results in the execution time behavior illustrated in Figure 6.7. By overlapping computation and communication the time saved is equivalent to T_{comp2} for this example.

The details of asynchronous/non-blocking communication are appropriate for more advanced courses, but a good learning goal for this material in core courses is that the students are made aware of the possibility of independent communication progress. This can be mentioned along with the BSP model in DS/A (with the goal of a bloom K classification).

6.3 High Performance Networks

Vocabulary:

RDMA - Remote Direct Memory Access, a method of writing to another system's memory directly over a network.

InfiniBand - A popular type of high performance network [5]

iWARP - RDMA capable Ethernet [6]

Cray Networks (Gemini and Aires) - Custom high performance networks developed for Cray supercomputers [7]

Portals - An RDMA network specification from Sandia National Laboratories [8] that has been implemented in the Cray SeaStar line of networks (predecessor to Gemini and Aries)

Myrinet (Myri 10G) - An RDMA capable network produced by Myricom [9]

Several different high performance networks are available today. The most popular, based on their use in the total number of the top 500 supercomputers in the world, are InfiniBand and Ethernet (both Gigabit and 10-Gigabit). However, the highest ranked supercomputers typically use other high performance networks. In some cases the networks are custom designed for a single supercomputer or a supercomputer model line. For example, the Cray supercomputing networks (Aries and Gemini) power some of the world's fastest supercomputers as of 2014.

The performance of networking is critically important to large supercomputers. That some supercomputer designers have gone to the expense and time to build custom network hardware attests to this fact. In the cluster computing world, the latency and bandwidth of the high performance network connecting the nodes are factors that greatly impact system performance. The overall message rate achievable by the networking hardware is also important, as is the overhead placed on a node's CPUs in order to deliver networking services. With the exception of Ethernet, all the other major high performance networks share some common features that help

them to achieve superior performance. These differences are discussed in more detail in section 6.3.1.

6.3.1 Differences from commodity networks

The main differences between high performance network technologies and typical Ethernet hardware are the inclusion of new high performance features and their base signaling rate. The key features of all high performance networks are operating system bypass (because calling the kernel is expensive), zero-copy functionality (whereby the network card can perform memory operations directly into the user-level application's buffers), and, for most networks, offloading (whereby the network card is capable of performing some processing on incoming/outgoing communications).

Operating system bypass simply means that the communication software/driver is not controlled through the kernel. Thus, a context switch to the kernel can be avoided when initiating communication calls. In addition, this mechanism avoids using kernel buffers that require a copy from the networking hardware into a kernel buffer, then an additional copy into the application's buffer. Copies should be avoided when possible as they incur overhead that can slow down communication. Zero-copy functionality for user level communication libraries means that the networking hardware can transfer the incoming data into the appropriate application buffer. While OS bypass avoids copies into the kernel buffers, a user level communication library could still use intermediate buffers, and hence incur the copy overhead. Zero-copy functionality indicates that a user level library is able to avoid any intermediary copies. Offloading is the final typical difference between high speed networks and commodity networks. Offloading allows for some communication processing to occur on the networking hardware itself. For example, an offloaded Ethernet network interface card could provide hardware processing of the TCP/IP stack. Ethernet is not typically used in the top high performance clusters. Figure

6.8 illustrates why this is the case. The non-Ethernet high performance network stack completely bypasses the OS, avoiding a context switch from the application to the kernel. The number of memory copies performed for the traditional networking stack is shown to point out that several memory copies need to occur that are not needed for the high performance networking stack.

Compared to high performance networks, Ethernet exhibits significant differences in performance. For example, for the InfiniBand high performance interconnect, data rates circa 2013 are 54.5 Gigabits/second and sub microsecond latencies are possible. The fastest widely-available Ethernet technology has a data rate of 10 Gigabits/second and latencies of approximately 5 microseconds. More typical 1-gigabit Ethernet has latencies in tens of microseconds (20-50).

It is helpful for students to understand the difference in the performance requirements of HPC versus traditional networking. A good in-class activity is to discuss the ping times from students' laptops to a departmental server and notice the multiple order of magnitude difference between those communication latencies and the requirements for a high performance network. Students can also use the latest high performance networking hardware specifications (typically publicly available on the internet), to estimate the amount of time that it would take to transfer a large file over a high performance network. For example, in order to transmit a DVD (2.7GB) worth of data over a 56Gb/s interconnect, the students must first convert the interconnect speed to GigaBytes (56/8 = 7GB/s), so a full DVD transfer would take 0.386 seconds to occur. This gives a real-world example that students can relate to their everyday lives, and the experiences they have with existing networks and will give them an appreciation for the power of high performance networks. The discussion of high performance networks is intended to augment the introduction of cluster computing and explain why high-performance interconnects are needed.

The goal of this material is to briefly outline some of the differences between high performance networking hardware and typical commercial hardware. This material

can be used to enhance in classroom discussion of networking or in elective courses. For core courses, a good learning goal is to make students aware that non-Ethernet networks exist for high performance/cluster computing and that they have advanced features that help them achieve lower latency and higher bandwidth.

6.3.2 Introduction to Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a mechanism to allow a compute node to access directly the memory of another compute node over the network. Typically RDMA is used to prevent interruption of the CPU while it is busy in order to handle incoming network data. There are a variety of methods of implementing RDMA. RDMA works in a manner very similar to traditional DMA requests. The main difference is that the initiator of the DMA request is another system, and the request must be sent over a network to the remote DMA controller. An example of this scenario is illustrated in Figure 6.9

RDMA is typically accomplished via the queue-pair model. The queue pair model uses connected pairs of queues at both the origin and the target. Each process has a send queue (SQ) as well as a receive queue (RQ). The SQ and RQ can be populated by work queue elements (WQEs, typically pronounced "wookies"). Each one of these elements corresponds to a communication task. The results of these operations can be placed in a Completion Queue (CQ), also called an Event Queue (EQ) in some RDMA designs. The host processes that use the RDMA device can both post WQEs to the send/recv queue as well as read/remove items from the CQ. This design allows for the processes/threads using the network to post elements describing messages to be sent or received to the local networking hardware and then return to working on other tasks. The network hardware attempts to make independent progress without the need for the application to call into a library in order to progress communication. The application can check on the progress that has been made at any time by examining the CQ/EQ to determine what has completed.

The basic design of the queue pairs and associated CQ/EQ is shown in Figure 6.10

There are two generally accepted models for enabling RDMA, one is called send-recv and it requires that the target node post information about where to place incoming messages that match a WQE in the receive queue. The initiator posts a send request to its SQ and the RDMA hardware sees the send WQE and processes it. This sends a message to the target. The target hardware receives the message and matches it to a corresponding RQ entry. The target then uses the information in the RQ WQE to place the message in the desired location.

The other RDMA model is called RDMA Write/Read, where the source node has information that has previously been exchanged with the target node. This information contains details about the available memory on the target node that can be read (RDMA Read) or written to (RDMA Write). Obviously there are security concerns with allowing RDMA, and while these can addressed using a variety of different security schemes, RDMA is typically only used on private high performance LANs. RDMA is typically paired with user-level communication libraries (providing direct access to network hardware without requiring intervention from the kernel). These techniques are essential to low latency (context switching into an OS kernel takes some time), and throughput. Many high performance computing applications send huge amounts of data and require very low latency communication in order to provide high levels of performance. In this domain, RDMA networks can achieve sub microsecond latencies, so advanced techniques like RDMA and OS bypass are necessary. These two methods can be introduced very briefly along with a discussion of cluster computing and the differences between it and other types of parallel computing. A natural place to introduce this material is when discussing memory accesses and DMA in particular. The exact details of the RDMA methods do not have to be introduced, but students should be made aware that methods of performing remote direct memory accesses exist, even if they do not understand all the implementation details, which are significant and most likely only appropriate for a

graduate level course.

6.3.3 Main Memory and RDMA

The DMA engines on RDMA capable hardware may encounter issues on some machines when they are not capable of determining if a given virtual-to-physical address is still resident in physical memory. Memory pinning can resolve this issue by preventing the swapping of memory in pinned regions. This mechanism guarantees that whenever the DMA engine accesses the pinned memory region, it is modifying the intended memory locations. The cost of memory registration can be significant and impacts performance when done dynamically in code. Consequently, many high performance messaging libraries (e.g., MPI) try to pre-register memory, so that a pool of buffers can be used for low latency communication. For large bulk data transfers, memory registration costs may be warranted. For very large transfers such costs may be unavoidable, as pre-registered buffers of suitable size would require large memory overheads for applications that do not perform very large transfers. Memory registration/pinning can also be pipelined with the writing of the data, reducing registration costs.

Not all high performance networks need to pin memory for RDMA operations. For networks working over lightweight high performance OS kernels, virtual memory may not be supported, and therefore pinning is unnecessary. Alternatively, past network designs have been able to avoid memory pinning at the OS level. The now defunct Quadrics QSNet network hardware [10] was able to manage the pinning of memory at the NIC hardware level. This approach required kernel modifications for Quadrics networks, as well as a NIC hardware TLB. Quadrics networks hardware could track page tables and manage application memory page swapping such that memory pinning was unnecessary.

6.3.4 Ethernet RDMA

A reasonable question that students may raise is why Ethernet has not been extended to support RDMA. TCP/IP offload engines with RDMA support are available, and RDMA capable Ethernet has been standardized as iWARP. There are a number of issues with providing RDMA over Ethernet that needed to be addressed in the iWARP standard. One of the major issues is the translation of a stream-based networking protocol (TCP) with the message-based semantics of RDMA queue pairs. One issue that has not been addressed is the scalability of the TCP connection-based networks. iWARP for message-based transports, namely SCTP [11] and UDP [12], have been proposed. Connectionless communication is also addressed by UDP, making it more scalable. However, at this time only TCP-based iWARP hardware is available, and wide area network RDMA has not been broadly adopted.

Alternative RDMA solutions, such as InfiniBand, have been proposed to run over Ethernet frames. RDMA over Converged Ethernet (RoCE), pronounced "rocky", uses the InfiniBand stack on top of Ethernet frames in order to provide InfiniBand style RDMA operations over Ethernet hardware [13].

6.4 Advanced concepts

6.4.1 Hybrid programming with MPI and OpenMP

Vocabulary: OpenMP - An API for shared memory multi-processing.

The message passing and shared memory programming paradigms are not mutually exclusive. Shared memory programming, e.g., with OpenMP, can be used simultaneously with MPI in the same program. A very simple example is one in which OpenMP is used to perform a parallel computation locally on a single compute node, and MPI is used to coordinate the work of many compute nodes. MPI could also explicitly pass messages between multiple processes on a single node through the network, and MPI can also be used to perform shared memory based message

passing.

A key question that may be asked in introducing hybrid programming is why another API is needed when MPI can communicate through shared memory itself. The reason hybrid programming is useful is that shared memory APIs like OpenMP are capable of handling threads dynamically (i.e., forking and joining threads). Creating new MPI processes on the fly is not trivial. Therefore, OpenMP is used in conjunction with MPI when it is desirable to have threads during a portion of the execution and large numbers of MPI processes are undesirable. Basically, when large amounts of concurrency are desired for only certain parallel portions of the code, hybrid programming can be a good solution. The addition of OpenMP avoids the need for MPI to spawn many more MPI processes, and the costly activity of managing unique IDs for each new process. Such extra MPI processes would also have to be involved in collective operations, when this may not be desirable. Instead a single MPI process per compute node can use OpenMP to create a number of worker threads for a particular parallel portion of code, then gather the result. The MPI processes can then exchange results if necessary (a second tier of concurrency), as shown in Figure 6.11, and illustrated in the idealized simulation example given below:[4]

```
for ( i = 0; i < NUM_SIMULATION_STEPS; i++ )
   #pragma omp parallel
   {
      #pragma omp for
         for ( j = 0; j < NUM_DATA; j++ )
            compute( j );
      #pragma omp master
         communicate_results();
```

[4]The OpenMP directives used in this example assumes familiarity with OpenMP, covered in another chapter of this book on shared memory programming.

```
}
```

Consider executing the example above on a cluster of compute nodes in which each node has multiple processor cores available. A possible hybrid usage of message passing and OpenMP thread parallelism would be to have one MPI process per node, each with an associated team of OpenMP threads equal to the number of processor cores per node. In the example, each simulation step would begin with all threads performing the computation over the data in the inner loop in parallel. Following that compute phase, only the master thread of each MPI process communicates the results of all the threads to the other MPI processes.

This material is useful to summarize the shared memory and distributed memory topics in DS/A, and why they can both be used at the same time. It is appropriate to address when discussing shared memory and also whenever discussing cluster computing or message passing. Students should be aware that such approaches exist, and that hybrid programming is useful. At the core course level students are not expected to understand the technical reasons why hybrid programming is useful, but a high-level understanding of the fact that shared memory approaches can better deal with dynamic concurrency than MPI would be desirable.

6.4.2 Supercomputers: Enabling High Performance Computing

Vocabulary: GPU - Graphics Processing Unit aka. GPGPU, a General Purpose Graphics Processing Unit

Many of the top supercomputers are computing clusters. They are assembled from enterprise components, with high performance (sometimes custom) networks. Individual computing nodes typically have large amounts of memory and multiple fast processors. Very high performance parallel file systems are needed to provide large datasets in a reasonable amount of time. Supercomputers typically use batch systems for job submission to accommodate the execution of multiple simultaneous workloads. Many supercomputers use accelerators like Graphics Processing

Units (GPUs) and co-processors to improve node-level performance. Supercomputers enable High Performance Computing, which is a basis for many basic scientific investigations and discoveries.

The goal of supercomputers is to keep overheads as low as possible, thus keeping programs as close to the "metal" as possible. Effective systems minimize the number of software layers between the application and the hardware. As such, many supercomputers use modified operating systems. Current state of the art approaches use lightweight kernels. Lightweight kernels have been in use for DoE supercomputers for several years and some supercomputer vendors provide lightweight kernels, like Cray's CNK [14]. The goal of these operating systems is to reduce "OS noise", which is slowdown introduced into a parallel system by the delay of processes (typically just a few) due to them being swapped out of a core in order to service operating system tasks (e.g., daemons, scheduled tasks, etc.). Lightweight kernels can also be used to remove overhead due to operating system features that are not required for supercomputers.

The applications used on supercomputers are typically large, complex scientific codes. Many of these applications have been written using MPI, and there is significant momentum to continue to provide MPI support in future generation systems. Radical new architectures occasionally force migration over to new programming environments and methods, provided that the performance benefit for re-writing applications justifies the effort required. An example of this is the BlueGene supercomputers from IBM. BlueGene systems require a different programming technique in order to extract peak performance from the system. GPUs and co-processors are introducing new programming models as well, and the programming of supercomputers is becoming increasingly complex. This is due to the exposure of more low level hardware details than were required in previous generation systems. For example, using a single homogeneous CPU architecture makes programming very large applications easier, as the application can use a single piece of code to run an

Table (6.1) Historical Supercomputer Performance

Name	Year	Processing (Flops)	CPU Cores	Memory
CDC 6600	1964	1 Mega	1 + 10 co-processors	960KB
Cray-2	1985	1.9 Giga	4	2GB
ASCI Red	1997	1.3 Tera	9298	1.2TB
RoadRunner	2008	1.02 Peta	6,912 AMD CPUs (13,824 cores) + 12,960 Cell Processors (116,640 cores)	104TB
Titan	2012	17.59 Peta	18,688 + 18,688 GPUs (560K cores total)	693.5TB
Tianhe-2	2013	33.86 Peta	32,000 CPUs + 48,000 Co-processors (3.12 million cores total)	1.34PB

operation on thousands of CPUs at the same time without issues surrounding the need for different instruction sets, nor any major difference in performance between the CPUs. Using accelerators that are not tightly integrated into the processor die requires mechanisms to move data efficiently to and from the accelerators.

Table 6.1 shows a historical summary of key supercomputers. It is interesting to note that until the 1990s supercomputers were not cluster computer type architectures. Since the 1990s, top supercomputers have been primarily cluster computing type systems, with the latest supercomputers combining both traditional CPUs and accelerators. The most current information on the top supercomputers can be found at www.top500.org [15].

Supercomputers are a vital tool for many important scientific discoveries and important issues facing society. It is used for pharmaceutical research, genetics, climate modeling, weather prediction, fundamental physics and chemistry research, energy discovery, and complex system simulation. Increasingly powerful supercomputers both enhance the speed at which such problems can be solved, and increase the resolution and fidelity of the results. Some problems are so large that mod-

ern supercomputers remain insufficient to solve them. For example, simulation of a human brain in real time is impossible with the supercomputers circa 2014. Supercomputing is an important technology that will continue to enable cutting edge scientific research for the foreseeable future. It also helps to enhance individuals' daily lives through improved health care, new technologies, faster design, and of course, daily weather prediction.

The information presented here can be used by instructors as supplementary information to encourage students' interest in parallel and distributed computing. It can be introduced in DS/A and systems when discussing cross-cutting topics, particularly cluster computing, power consumption and fault tolerance, as these are key concerns for supercomputers. It can also be introduced wherever there is a desire for a real-world example of the potential benefits that parallelism and distributed computing provide and the impact that such technologies have on the world around us.

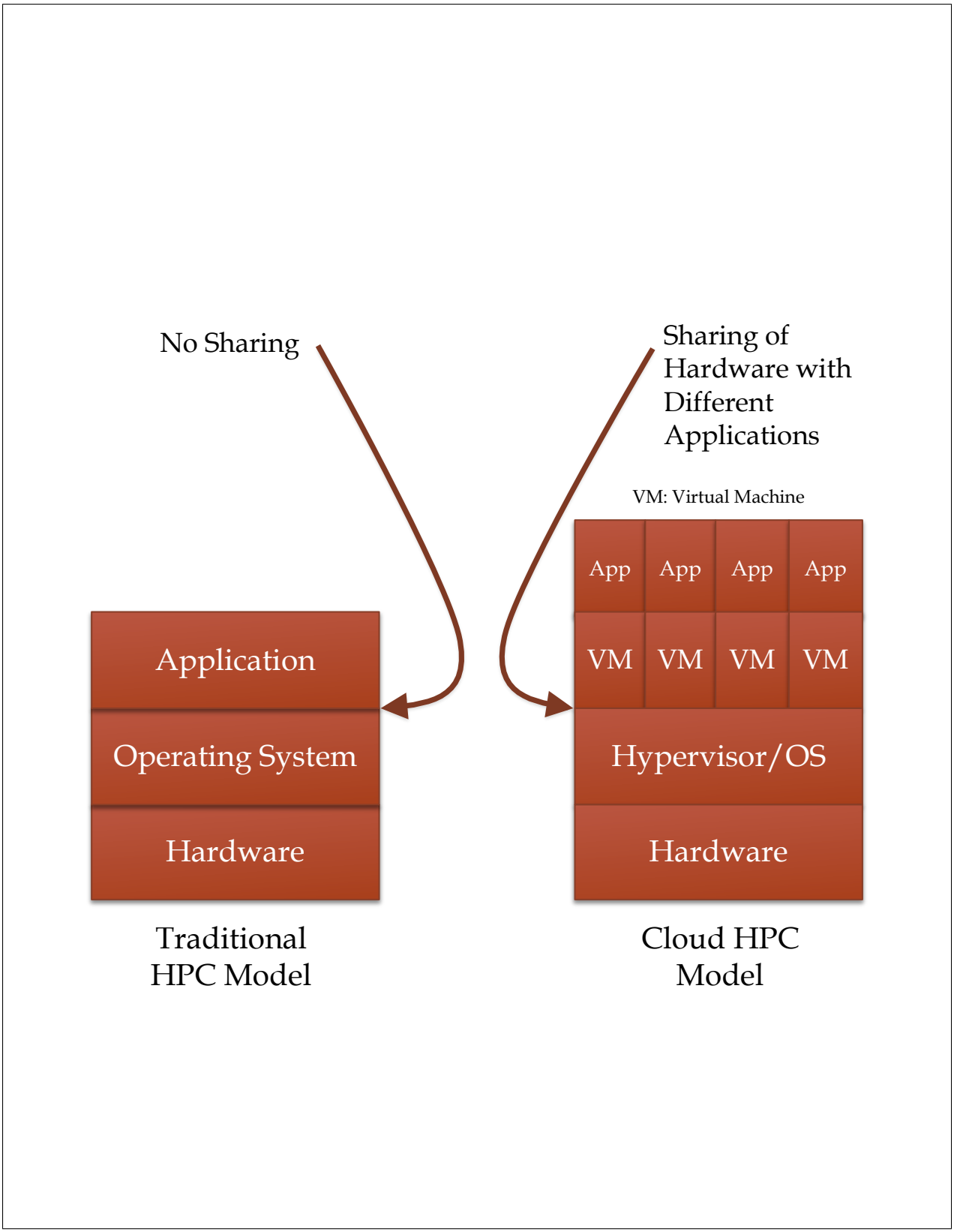

Figure (6.1) Traditional HPC compared to the Cloud HPC Model

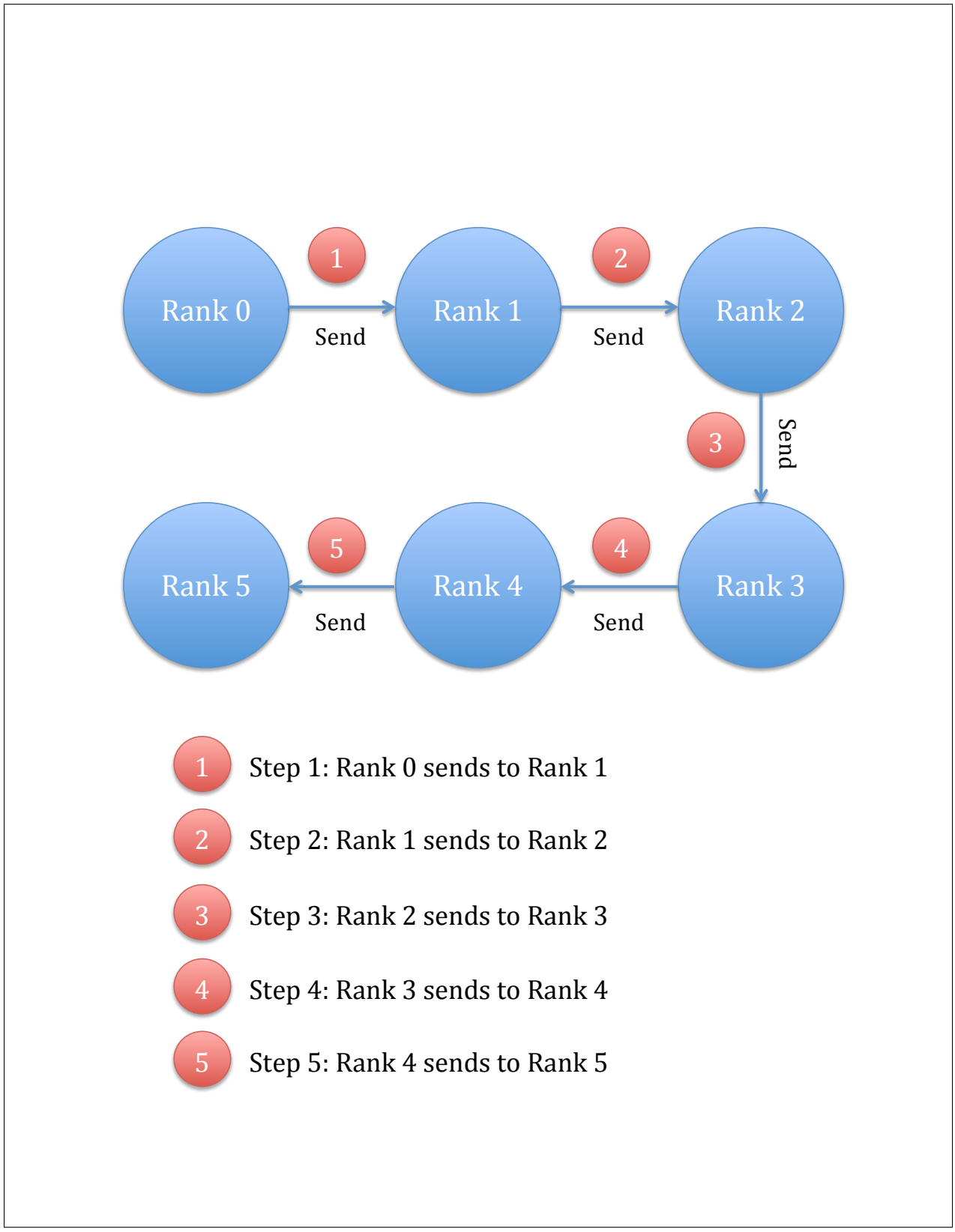

Figure (6.2) Point-to-point Communication Example.

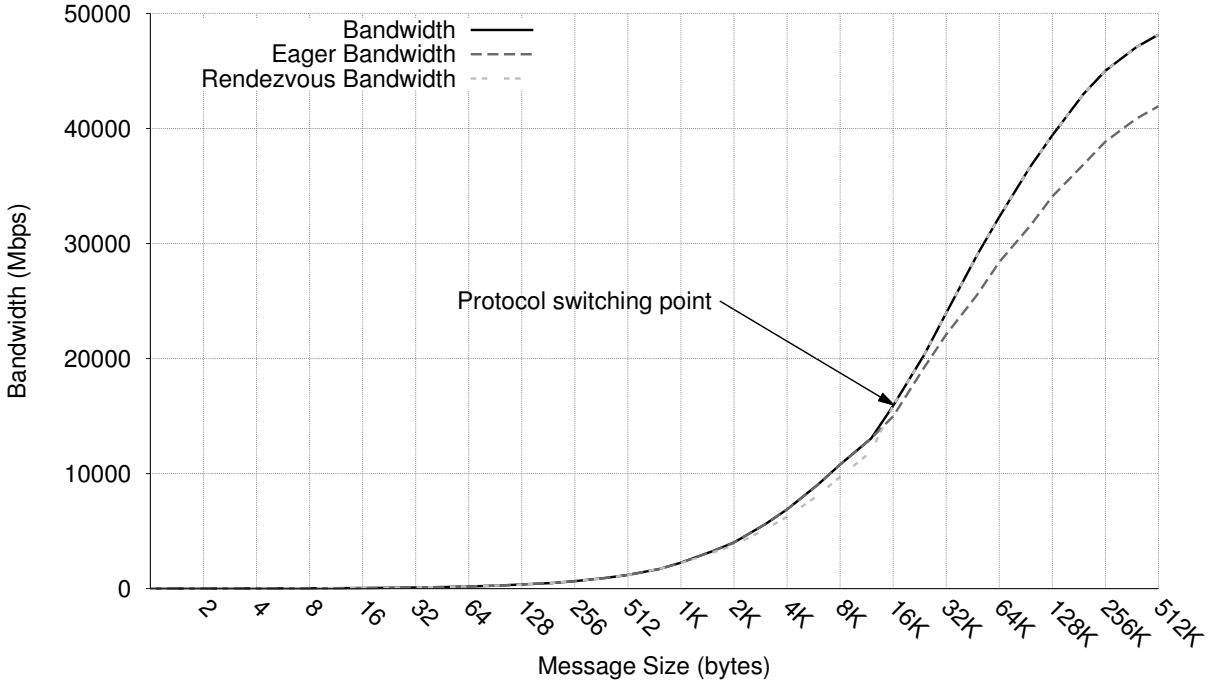

Figure (6.3) Point to Point protocol switching from Eager to Rendezvous

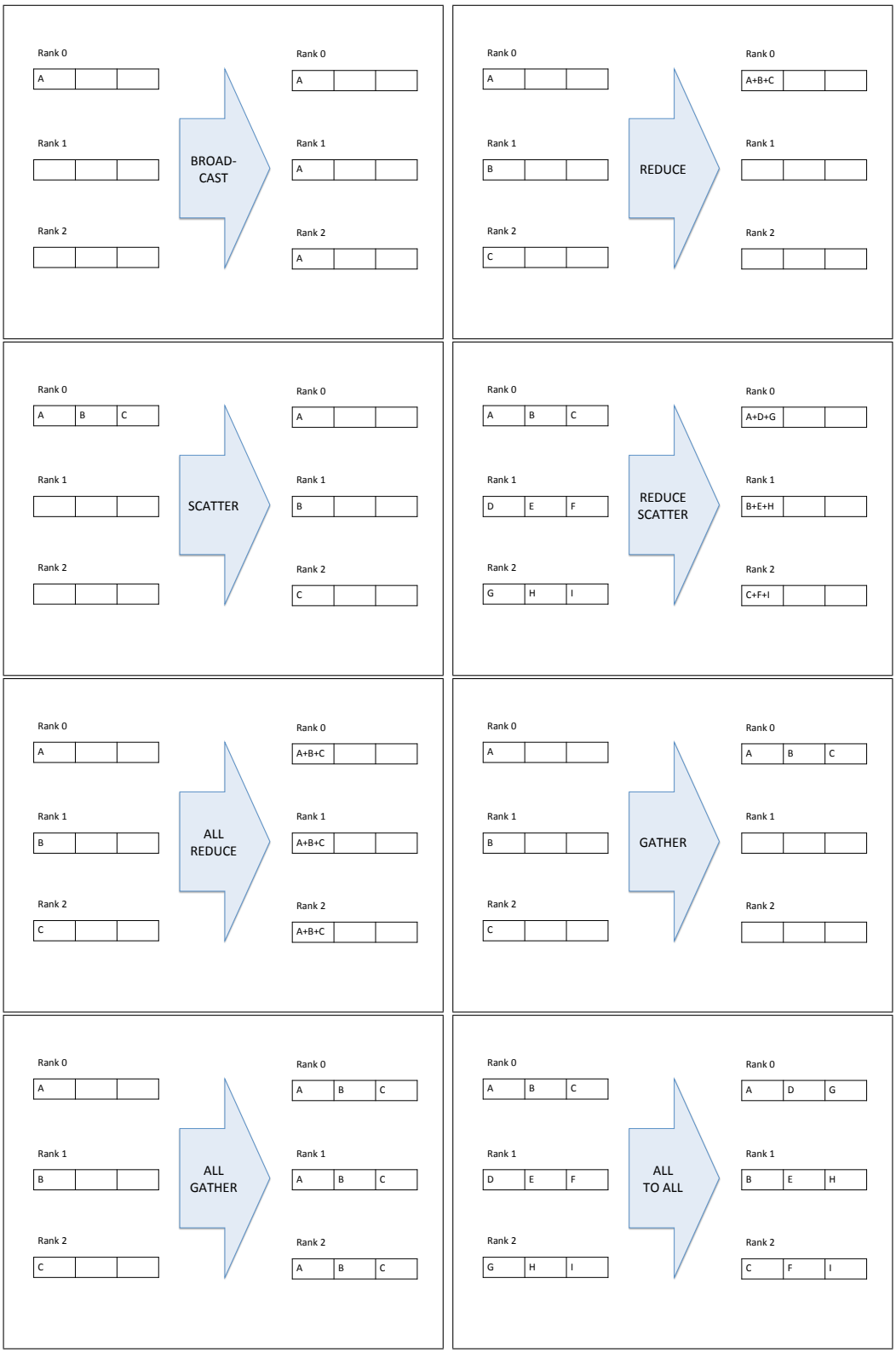

Figure (6.4) Collective operations for message passing programs.

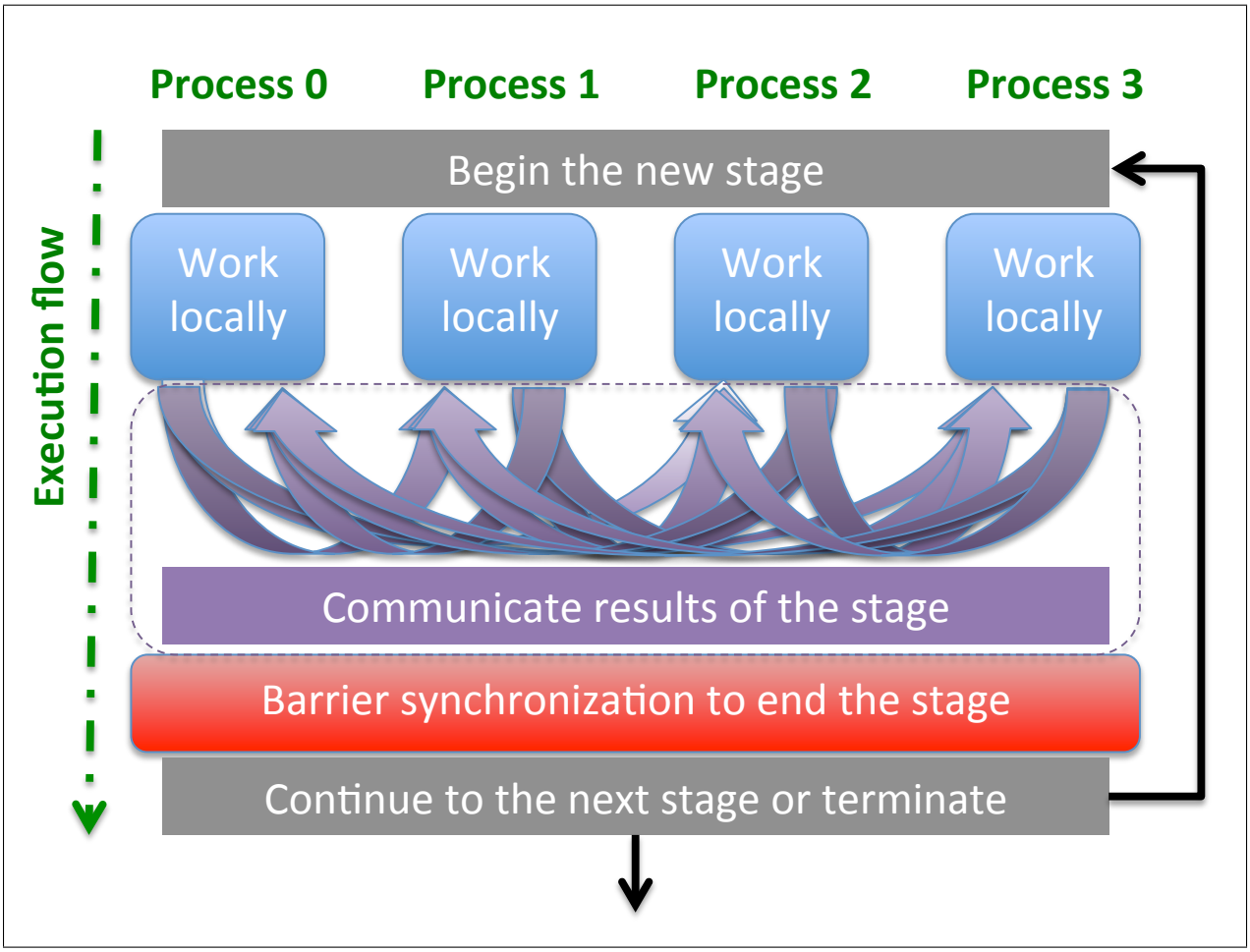

Figure (6.5) A BSP superstep.

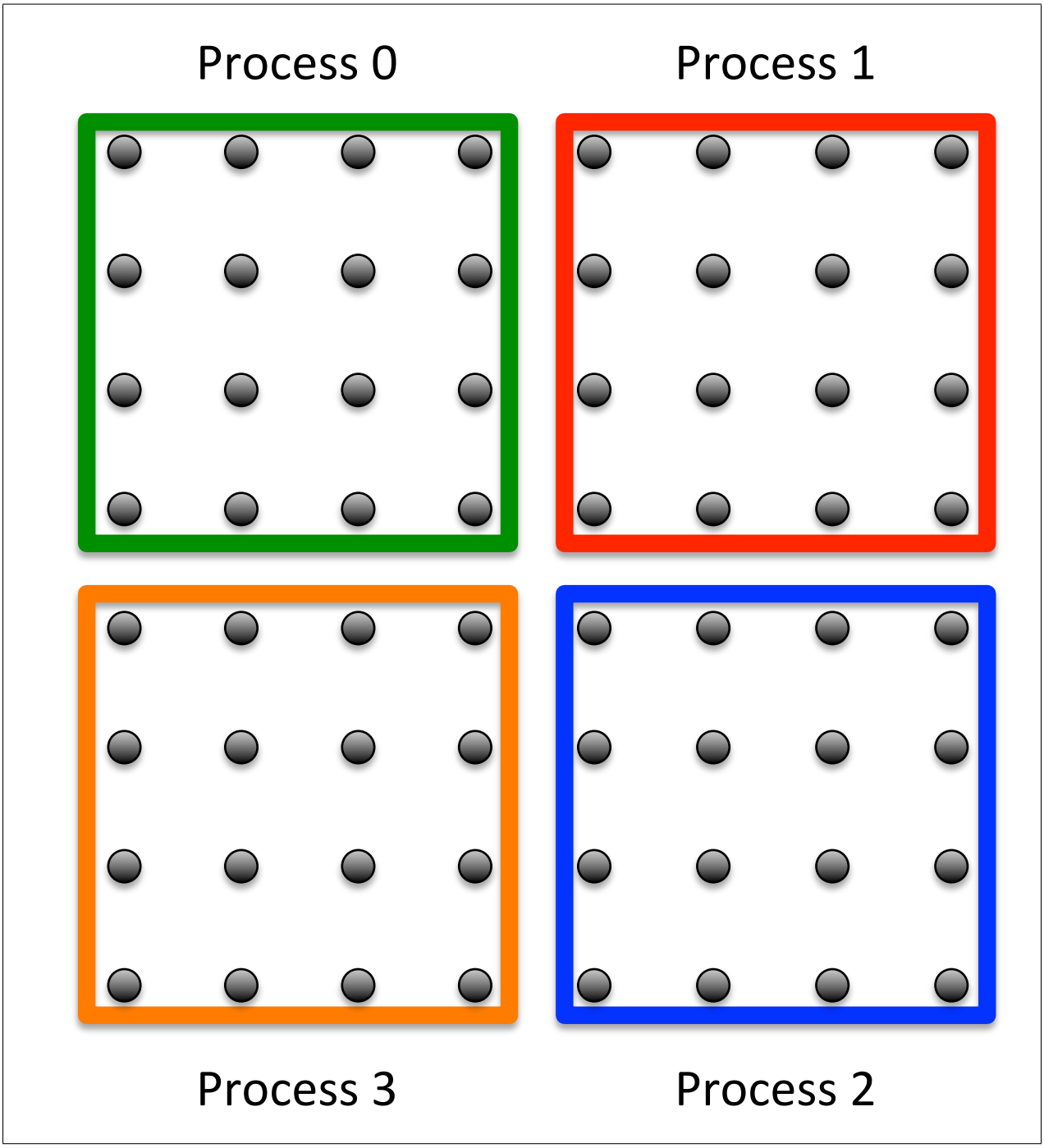

Figure (6.6) Example division of grid points among processes in the BSP temperature program.

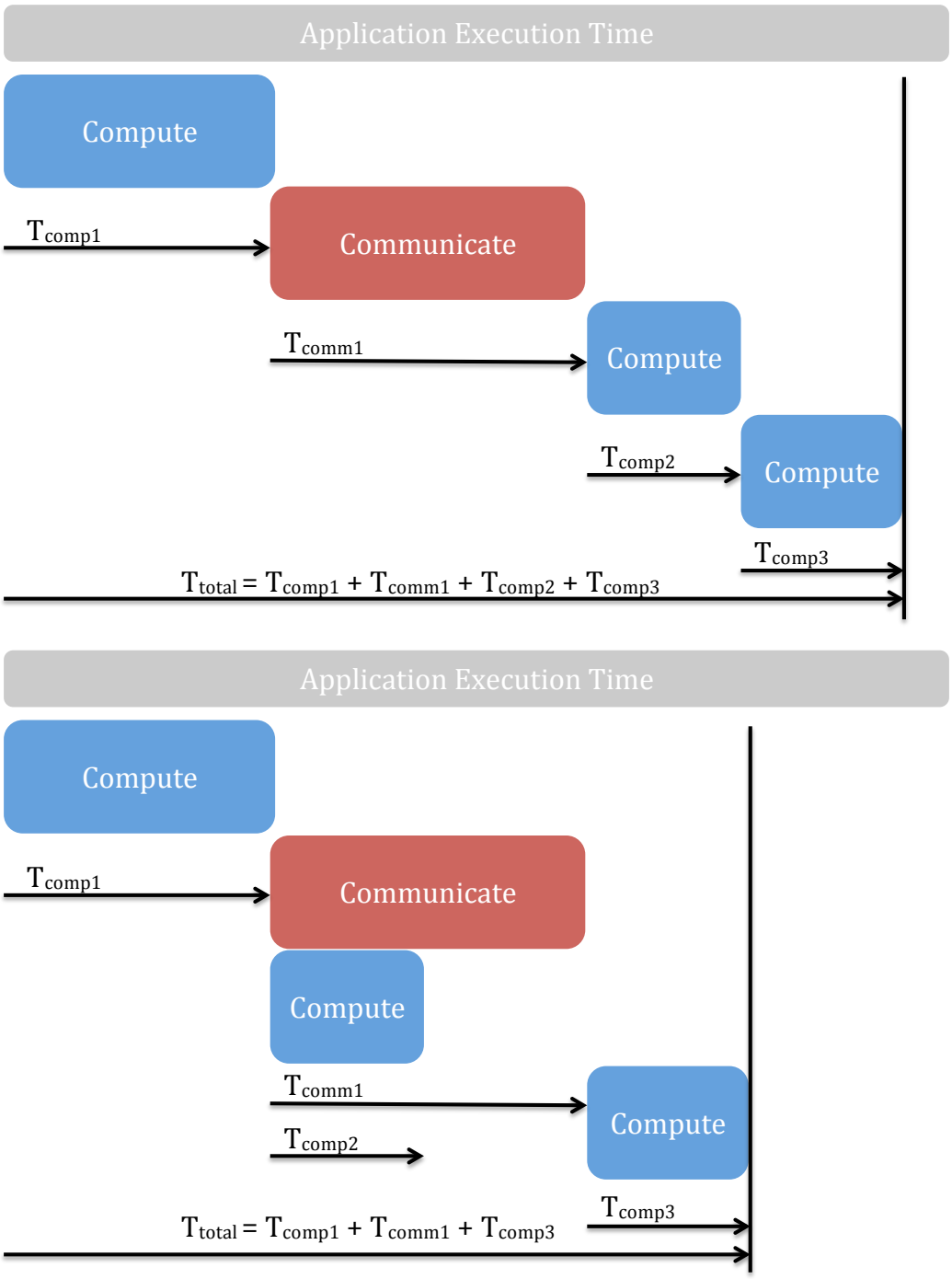

Figure (6.7) Computation-Communication Overlap Execution Comparison

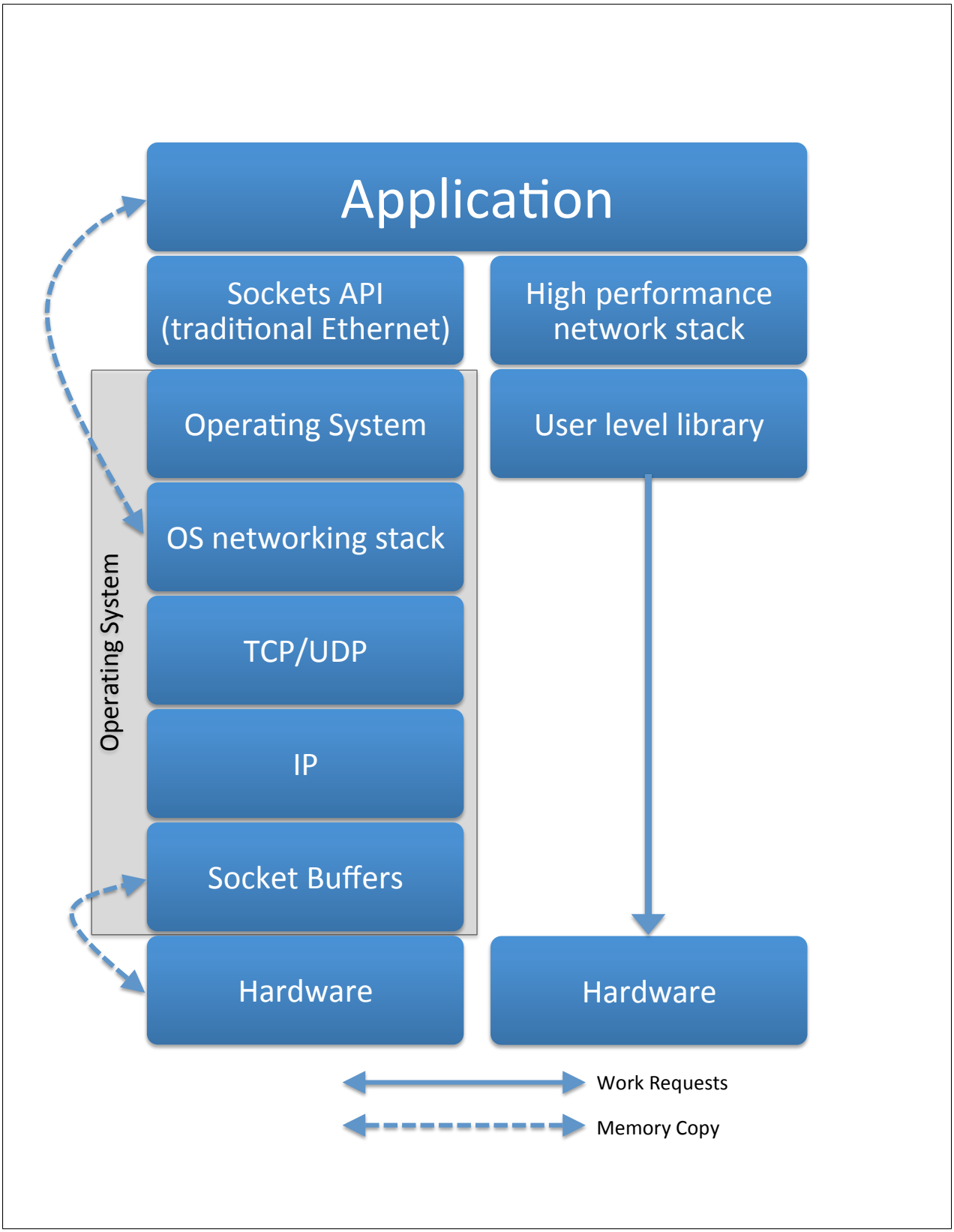

Figure (6.8) Comparison of networking stacks - traditional vs. OS bypass

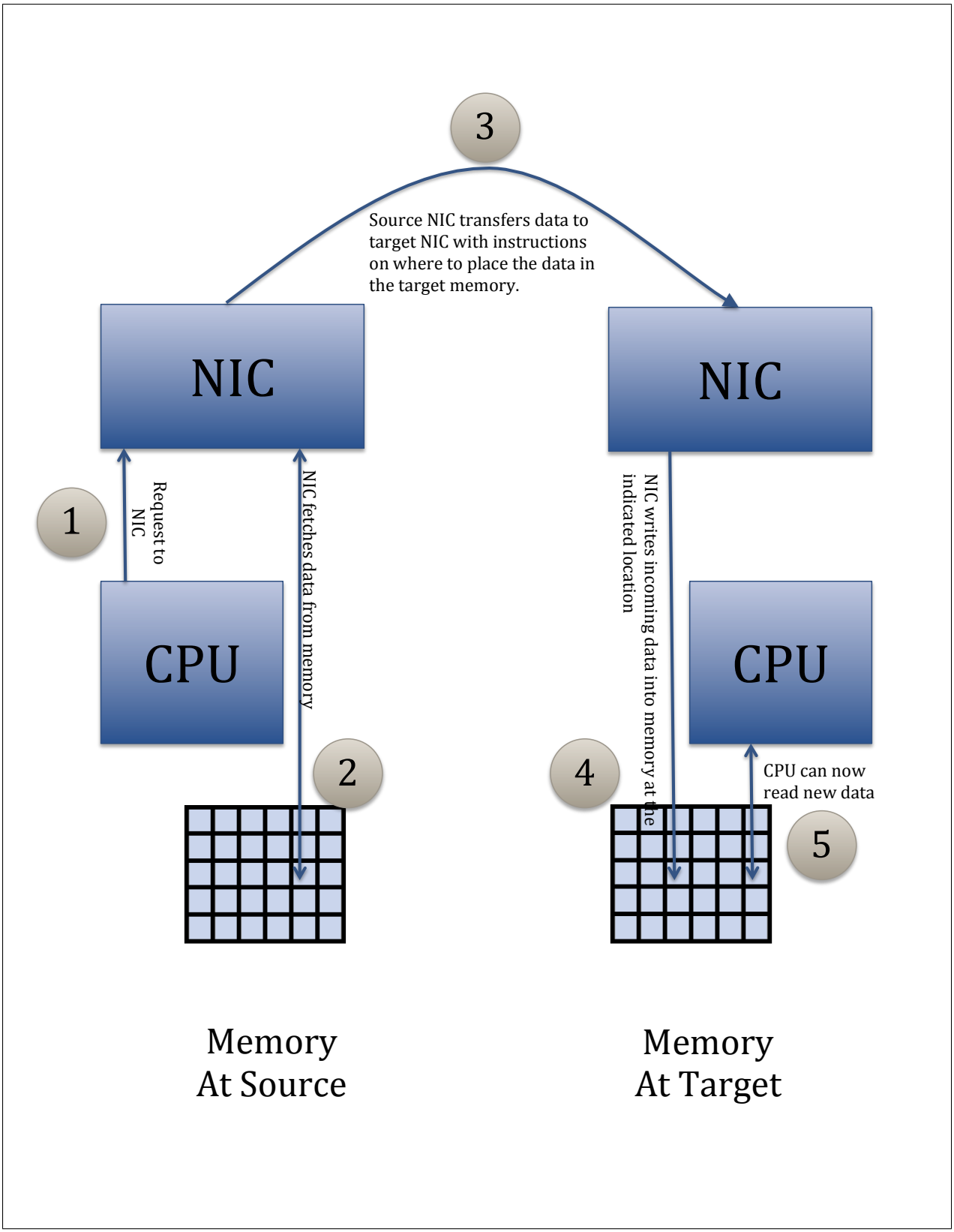

Figure (6.9) Remote Direct Memory Access Overview

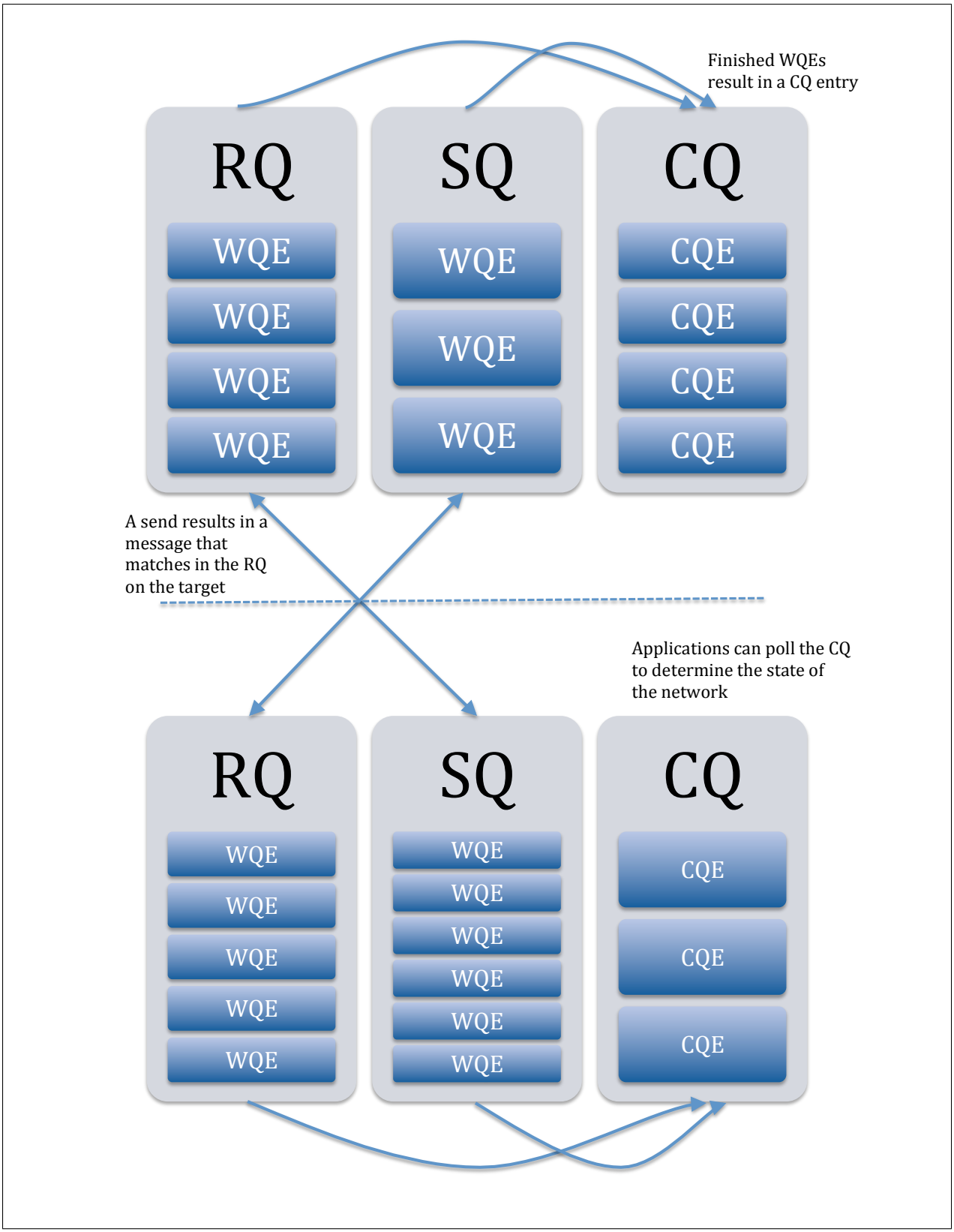

Figure (6.10) The queue pair model

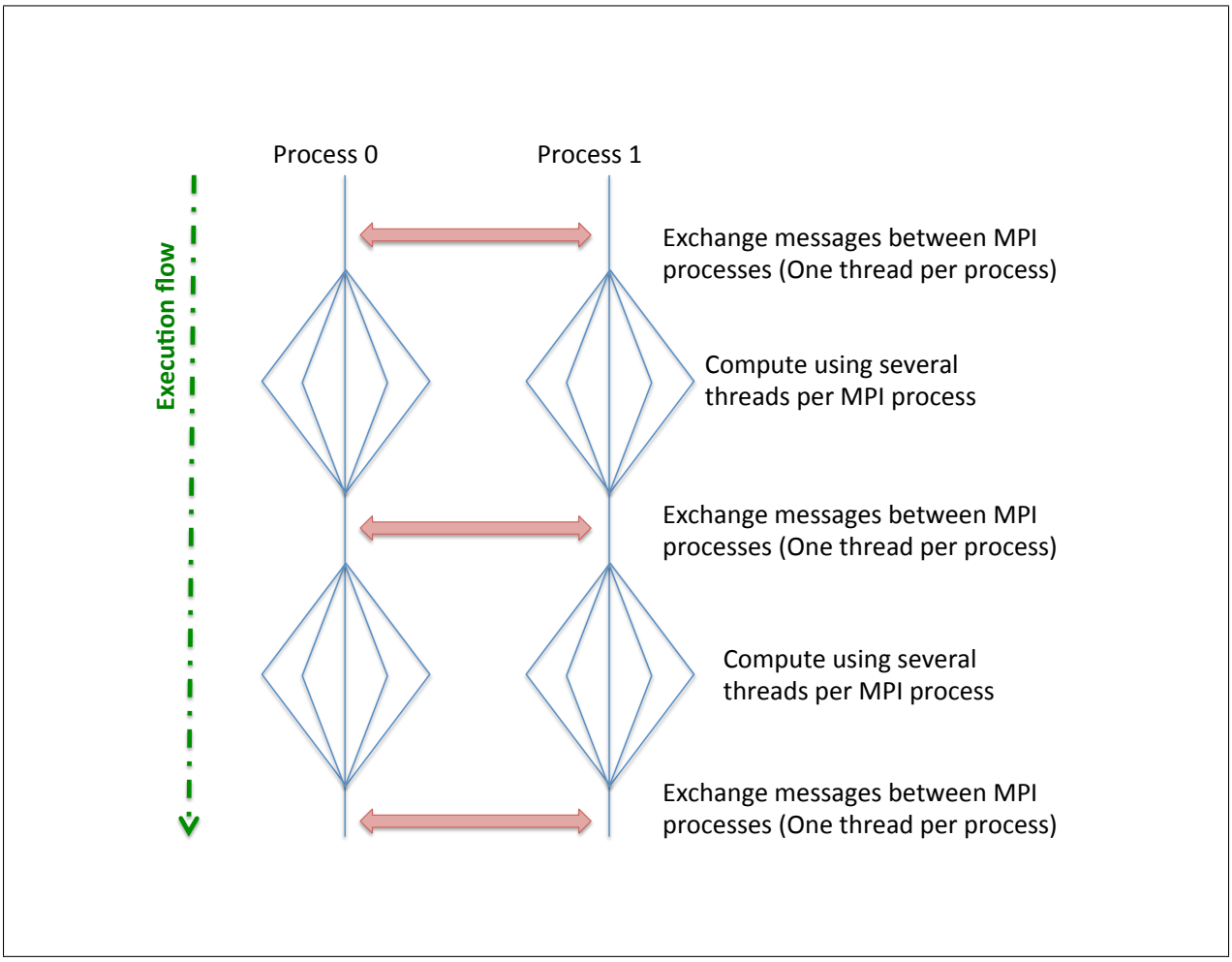

Figure (6.11) Hybrid programming using threads and MPI

REFERENCES

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on a million processors," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2009, pp. 20–30.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[3] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[4] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," September 2012.

[5] InfiniBand Trade Association, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[6] RDMA Consortium, "Architectural specifications for RDMA over TCP/IP," *URL http://www.rdmaconsortium.org*, 2005.

[7] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*. IEEE, 2010, pp. 83–87.

[8] Barrett, B.W. and Brightwell, R. and Grant, R.E. and Hemmert, S. and Pedretti, K. and Wheeler K. and Underwood, K.D. and Reisen, R. and Maccabe, A.B., and Hudson, T., *The Portals 4.0.2 network programming interface*, Sandia National Laboratories, October 2014, technical Report SAND2014-19568.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *Micro, IEEE*, vol. 15, no. 1, pp. 29–36, 1995.

[10] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network: High-performance clustering technology," *Micro, IEEE*, vol. 22, no. 1, pp. 46–57, 2002.

[11] M. Tsai, B. Penoff, and A. Wagner, "A hybrid MPI design using SCTP and iWARP," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[12] R. E. Grant, M. J. Rashti, A. Afsahi, and P. Balaji, "RDMA capable iWARP over datagrams," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 628–639.

[13] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun, "Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options," in *High Performance Interconnects, 2009. HOTI 2009. 17th IEEE Symposium on*. IEEE, 2009, pp. 123–130.

[14] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski, "Experiences with a lightweight supercomputer kernel: lessons learned from blue gene's cnk," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 2010, pp. 1–10.

[15] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 supercomputing sites," 2013. [Online]. Available: http://www.top500.org/