Collective Communication Patterns

Nicholas Chen * nchen@illinois.edu
Rajesh K. Karmani * rkumar8@illinois.edu
Amin Shali * shali1@illinois.edu
Bor-Yiing Su ** subrian@eecs.berkeley.edu
Ralph Johnson * johnson@cs.uiuc.edu

* Computer Science Department
University of Illinois at Urbana-Champaign
** EECS Department
University of California, Berkeley

April 30, 2009

CHAPTER 1

Introduction

Normal point-to-point communication is the process of sending and receiving messages from one unit of execution(UE)[MSM04] to another UE. Collective communication, on the other hand, is the process of exchanging information between multiple UEs: one-to-all or all-to-all communications.

While arbitrarily complex communication patterns can emerge depending on the parallel algorithms used and the topology of the UEs, most communication between UEs actually follow very regular patterns.

Our pattern language, shown in Table 1.1, provides a catalog of those regular collective communication patterns that are used in scientific computing[1]. While there are various collective communication patterns, we have distilled those that we have found to be used commonly in various algorithms.

Following such patterns whenever possible not only make programs more succinct but also expose their intents better to other programmers. These collective communication patterns are so common in scientific computing that most parallel programming environments such as OpenMP, MPI and Java provide their own built-in constructs to support some, if not all, of them. Programmers are encouraged to browse through the list of APIs on their specific platforms to see if such patterns have already been provided before implementing their own. Using the built-in constructs not only prevents reinventing the wheel but also offers better performance since the constructs have been tuned for their specific platforms.

In this paper, we examine the following parallel programming environments and discuss the facilities that they provide for our patterns:

1. MPI[MPI]
2. OpenMP[Ope]
3. Charm++[Cha]

[1] For this submission, we will only focus on the first two patterns: Broadcast and Reduction.

4. CUDA[CUD]
5. Cilk[Cil]
6. FJ Framework[JSR]
7. Intel TBB[TBB]
8. Java[Jav]

Pattern	Problem	Classification
Broadcast	How does one efficiently share the same data from one UE to other UEs?	One-to-all
Reduction	How does one efficiently combine a collection of values, one on each UE, into a single object?	All-to-one
Scatter	How does one efficiently divide data from one UE into chunks and distribute those chunks to other UEs?	One-to-all
Gather	How does one efficiently gather chunks of data from different UEs and combine them on one UE?	All-to-one

Table 1.1: Collective Communication Pattern Language

These patterns often work together in an algorithm. Some algorithms will use the reduction pattern to combine the values followed immediately by a broadcast to send the values to every other UE. Because this combination of reduce-broadcast is used often, some environments actually provide constructs to support them. In MPI, this is provided as the `MPI_Allreduce` construct.

Our pattern language does not present the combination of patterns in our catalog. Instead, only the basic patterns are presented and the programmers can determine what combinations to use depending on their algorithms and the constructs that are provided in their environments.

CHAPTER 2

Broadcast

2.1 Problem

How does one efficiently share the same data from one UE to other UEs?

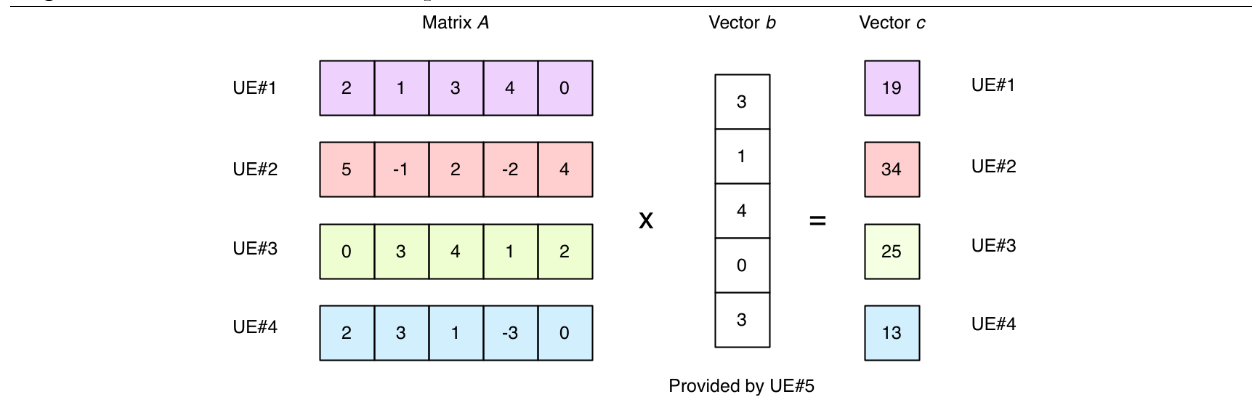
2.2 Context

In parallel computing it is common for one UE, say UE#1 to have the data that other UEs require for their computation. This arises naturally from using the Task Decomposition and Data Sharing patterns[MSM04] where each UE is assigned a different responsibility. For instance, UE#1 might be responsible for reading data from disk and it needs to share the data that it has read with the other UEs. Or UE#1 might be responsible for computing the result of a long running computation that it needs to share with the other UEs once it is done; the other UEs can still proceed with their computations while waiting for UE#1.

Consider the case of matrix-vector multiplication used in solving circuit equations[Boy97]. One simple way to parallelize this algorithm would be to decompose the original matrix A into rows and assign a row to each UE. Another UE would be responsible for obtaining the vector b and sharing the vector b with all the other UE. We call this process of sharing the same data with other UEs a Broadcast.

Once the data has been successfully shared, each UE can then calculate the result for a particular row in the resulting vector c. If necessary, the partial results for each row of the vector c can then be combined using the Gather pattern. The parallelized matrix-vector multiplication is illustrated in Figure 2.1. Same colored elements reside on the same UE.

Figure 2.1 Matrix-vector Multiplication



2.3 Also Known As

In *computer networking*, there are two common terms to describe sharing of data from one UE to other UEs.

A *multicast* operation delivers the data from one node to other nodes in a particular **group** — a collection of nodes — in the network. On the other hand, a *broadcast* operation delivers the data from one node to **every** node in the network.

However, the term broadcast has been used extensively in the parallel computing world to mean *delivery to a group of UEs or to every UE* in the topology. The facility for creating groups depends on the parallel programming environment.

In our pattern language, we shall use this more general definition of broadcast.

2.4 Forces

One-to-all or One-to-many In a broadcast operation, it is possible to send the data to every UE in the topology (*one-to-all*) or to a particular group of UE (*one-to-many*). Whenever possible, restricting the broadcast to only UEs that require the data is better for performance. In the case of message-passing, doing so reduces the messages that need to be sent on the network. And in the case of shared-memory environments, doing so reduces the overhead of maintaining cache coherence when a shared variable is modified.

Push vs. Pull A broadcast operation is essentially a push operation. Every UE that is involved in the broadcast operation must be prepared at some point to receive the data. Some parallel programming algorithms fit this push model very well. On the other hand, some algorithms might fit better with a pull model where the UEs that are interested in obtaining the data will query the UE with that data *when* it needs it.

2.5 Solution

Use the broadcast construct if one is provided by the parallel programming environment. BROADCAST is such a common pattern in scientific computing that it is included in most parallel program-

ming environments (see Table 2.1). The broadcast construct that is provided by the programming environments has been tuned to satisfy the needs of most programmers.

Environment	Broadcast Construct
MPI	`MPI_Bcast(buffer, count, datatype, root, comm)`
OpenMP	#pragma omp flush
Charm++	Sending a message to a `ChareArray`-type object broadcasts it to all objects contained in that ChareArray
CUDA	NONE
Cilk	NONE
FJ Framework	Can be simulated using the `apply(Ops.Procedure procedure)` method on a `ParallelArray`
Intel TBB	Use the provided atomic operations on atomic<T> datatypes
Java	`volatile`, `synchronized`, explicit and implicit locks, `java.util.concurrent.atomic` classes

Table 2.1: Broadcast constructs in parallel programming environments

Broadcast is essentially a way of exchanging information between UEs. In a message-passing environment, a broadcast operation is done by sending messages to the other UEs – the only way to exchange information between UEs. Thus using the broadcast construct involves thinking about the coordination between different UEs and knowing when to invoke the broadcast construct.

In a shared-memory environment, however, a broadcast operation is actually a read operation on some shared variable. Since the data is stored in a shared variable, it does not need to be sent to different UEs. A UE that is interested in the data can just access it. The challenge, in this case, is to synchronize access to the shared variable so that interested UEs do not access a stale value. Every time a variable is updated, the UEs have to be notified, the cached copies have to be invalidated and the new value has to be fetched from memory when needed.

In an environment such as Java, broadcasting the value of a variable is as simple as annotating it with the `volatile` keyword and letting the environment take care of the rest. A variable declared as `volatile` is guaranteed to have its last written value reflected across all UEs.

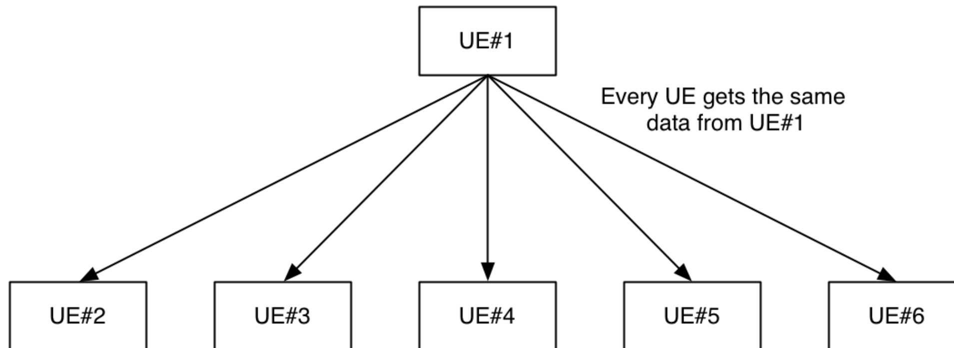
Some environments such as MPI enforce that all the UEs must be created at the start of the program; other environments such as Java and Cilk allow the creation of new UEs dynamically at runtime. In such dynamic environments, the source UE can broadcast its data to the new UEs as it is creating them. For instance, in Java, a parent thread can spawn new threads and broadcast a copy of its data to all its child threads through their constructors.

If the parallel programming environment does not provide a broadcast construct programmers might need to write one of their own. The Actor pattern discussed in [OPL] is a pattern centered around message-passing. However, unlike MPI or Charm++, popular Actor[Agh86] languages such as Erlang and Scala do not provide built-in constructs for broadcast and programmers have to implement them by hand.

We summarize the two common methods — sequential broadcast and recursive doubling — for implementing broadcast operations in a message-passing environment presented in [GGKK03]. Implementing broadcast in a shared memory environment usually requires primitives to be provided by the underlying OS and is beyond the scope of this paper.

2.5.1 Sequential Broadcast

Figure 2.2 Sequential Broadcast



A sequential broadcast is the simplest way to implement a broadcast operation. In Figure 2.2, UE#1 is in charge of sending the same data to the other UEs. While this is a simple approach, it is also inefficient and unscalable because UE#1 becomes a bottleneck; the communication channels between pairs of the other UEs are not being utilized at all.

Nonetheless this approach is simple and convenient when there are not a lot of UEs to broadcast the data to.

And in some cases, this might be the only way to do a broadcast. In an MPI environment, all the UEs are aware of one another and can communicate by using their rank IDs. This works because there is a static number of UEs that are initialized once and remain available throughout the entire computation. However, in a more dynamic environment such as an Actor system, UEs might be created and destroyed as the computation proceeds. Thus, not every UE will be aware of all the newly created or just destroyed UEs. Each UE knows only a subset of the available UEs in the topology. In fact in such a dynamic environment, making each UE aware of the other UEs is prohibitively expensive since it would require an update to be send to every creation and destruction of a UE.

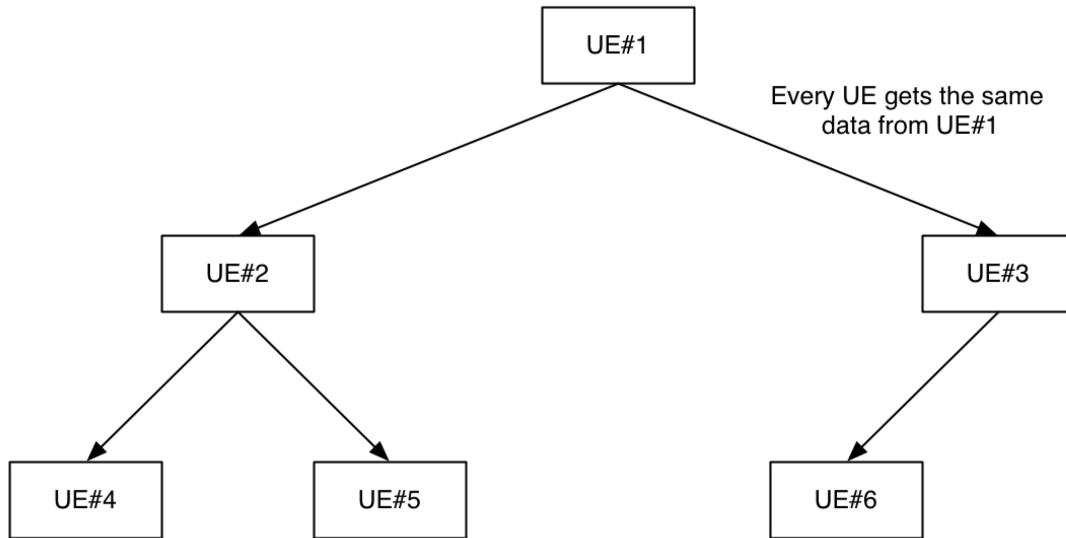
So in Figure 2.2, it might be the case that UE#1 is the only UE that knows about UE#2, UE#3, UE#4, UE#5 and UE#6. If so, only UE#1 can broadcast the data and we cannot use the more efficient recursive doubling technique described in Section 2.5.2.

2.5.2 Recursive Doubling

The recursive doubling technique is a more efficient way to broadcast data from one UE to other UEs. In Figure 2.3, UE#1 sends the data first to either UE#2 or UE#3. After receiving the data, UE#2 can start sending the data to UE#4 and UE#5 while UE#3 can start sending its copy of the data in parallel to UE#6. Thus the broadcast operation completes in logarithmic time instead of linear time.

Recursive doubling is useful when there are a lot of UEs to broadcast the data to. However, as mentioned in Section 2.5.1, this technique only works if every UE is capable of sending a message to every other UE **and** if it is possible to determine an order for sending the messages so that there isn't any duplicate message to any UE.

Figure 2.3 Recursive Doubling



2.6 Invariants

Precondition A value on one UE that needs to be distributed to other UEs.

Invariant The initial value on the source UE.

Postcondition The value from the source UE is now distributed on all the destination UEs.

2.7 Examples

2.7.1 Matrix-vector Multiplication in MPI

Listing 2.1 shows an example of using the built-in `MPI_Bcast` construct in MPI to solve the matrix-vector multiplication problem discussed in Section 2.2 and illustrated in Figure 2.1.

In our simple implementation, we assume that 5 UEs are available. The UE with the `VECTOR_PROVIDER_NODE` ID is responsible for obtaining the value of the vector. Each of the other UEs is in charge of a particular row of the matrix. Once the `VECTOR_PROVIDER_NODE` UE has obtained the value of the vector, every UE (including itself) calls the `MPI_Bcast` function. The following code snippet shows the arguments to the function.

```
67  if(my_id == VECTOR_PROVIDER_NODE)
68    obtain_vector();
69
70  MPI_Bcast(vector, VECTOR_SIZE, MPI_INT, VECTOR_PROVIDER_NODE, MPI_COMM_WORLD);
```

The `vector` argument is the data to be broadcasted to all the UEs. In this case, it is an array of integers. The `VECTOR_SIZE` argument is the number of elements in the array to broadcast. The `VECTOR_PROVIDER_NODE` argument tells the implementation that the value of the `vector` argument resides on the node with `VECTOR_PROVIDER_NODE` as its ID. And finally the `MPI_COMM_WORLD` argument tells the implementation to broadcast the value to every UE in the topology.

Listing 2.1 Matrix-vector Multiplication Example Using `MPI_Bcast`

```
39 int calculate_result_for_row(int row){
40   int result = 0;
41 
42   int* matrix_row = get_matrix_row(row);
43 
44   int i;
45   for(i = 0; i < VECTOR_SIZE; i++) {
46     result += matrix_row[i] * vector[i];
47   }
48 
49   return result;
50 }
51 
52 int main(int argc, char* argv[]) {
53 
54   int my_id;
55   int number_of_processors;
56 
57   //
58   // Initialize MPI and set up SPMD programs
59   //
60   MPI_Init(&argc,&argv);
61   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
62   MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
63 
64   //
65   // One of the node obtains the vector and broadcasts it
66   //
67   if(my_id == VECTOR_PROVIDER_NODE)
68     obtain_vector();
69 
70   MPI_Bcast(vector, VECTOR_SIZE, MPI_INT, VECTOR_PROVIDER_NODE, MPI_COMM_WORLD);
71 
72   //
73   // All other UEs should calculate their partial values
74   //
75   if(my_id != VECTOR_PROVIDER_NODE) {
76     int my_partial_result = calculate_result_for_row(my_id);
77     printf("The partial result for row %d is %d\n", my_id, my_partial_result);
78   }
79 
80   MPI_Finalize();
81 
82   return 0;
83 }
```

After the vector has been successfully broadcasted, each UE calculates the partial value for each row in the resulting matrix and prints it out to the console.

Listing 2.2 Matrix-vector Multiplication Example Using `AtomicIntegerArray` in Java

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.atomic.AtomicIntegerArray;
3
4  public class MatrixVectorMultiplication {
5    private static AtomicIntegerArray vector;
6    ....
7
8    static class RowMultiplier implements Runnable {
9
10     ....
11
12     public void run() {
13       waitForVectorBroadcast();
14
15       int sum = 0;
16       for (int i = 0; i < row.length; i++) {
17         sum += row[i] * vector.get(i);
18       }
19       System.out.println("The partial result for row " + rowNumber + " is " +
20                          sum);
21     }
22
23   }
24
25   public static void createMatrixVectorMultiplicationUEs() {
26
27     ExecutorService executor = Executors.newFixedThreadPool(VECTOR_SIZE);
28
29     for (int row = 0; row < VECTOR_SIZE; row++) {
30       RowMultipliers[row] = new RowMultiplier(row);
31       executor.execute(RowMultipliers[row]);
32     }
33
34   }
35
36   public static void main(String[] argv) {
37
38     createMatrixVectorMultiplicationUEs();
39
40     obtainVector();
41
42   }
43
44 }
```

2.7.2 Matrix-vector Multiplication in Java

Listing 2.2 shows an example of using the `AtomicIntegerArray` class from the `java.util.concurrent.atomic` package in Java 5 to solve the same matrix-vector multiplication problem.

The implementation is similar to the MPI version from Listing 2.1. `RowMultiplier` objects are

spawned and run in their own threads while waiting for the main thread to broadcast the value of the vector to them. Once the main thread obtains the value of the vector, the cached value of the vector in each of the threads is invalidated and they all see the new value from the main thread.

As mentioned previously, a programmer can rely on the `volatile` keyword automatically to immediately broadcast the modified value of a shared variable across all UEs in Java, However, when applied to an array the `volatile` keyword doesn't extend the broadcast capabilities to the individual elements of the array. Thus, updates to the elements of the array might not be broadcasted to the other UEs which might still have cached copies of the values.

The `AtomicIntegerArray` class solves that problem by extending the semantics of the `volatile` keyword. We use it in our Listing 2.2 to store the values of the `vector` variable used for matrix multiplication. Any updates to the `vector` variable from the main UE will be broadcasted to the other UEs.

2.8 Known Uses

Broadcast is used for many important parallel algorithms such as matrix-vector multiplication, Gaussian elimination, shortest paths and vector inner product.

2.9 Related Patterns

Message-Passing In message-passing environments such as MPI and Charm++, Reduction is always implemented using the Message-Passing pattern[OPL].

Scatter Broadcast sends the same data to other UES; Scatter sends different chunks of data to other UEs.

CHAPTER 3

Reduction

3.1 Problem

How does one efficiently combine a collection of values, one on each UE, into a single object?

3.2 Context

Most parallel algorithms divide the problem that needs to be solved into tasks that are assigned to different UEs. Doing so allows the different tasks to be performed in parallel on different UEs. Once those tasks are done, it is usually necessary to combine their partial results into a single object that represents the answer to the original problem.

Consider the problem of numerical integration of some function. One way to parallelize this algorithm would be to divide the domain of the function into different parts and assign a UE to integrate each part. Once all the UEs are done with the integration of their sub-domains, we combine their results using a *sum operator* to obtain the total value. This pattern of combining the results is called a REDUCTION.

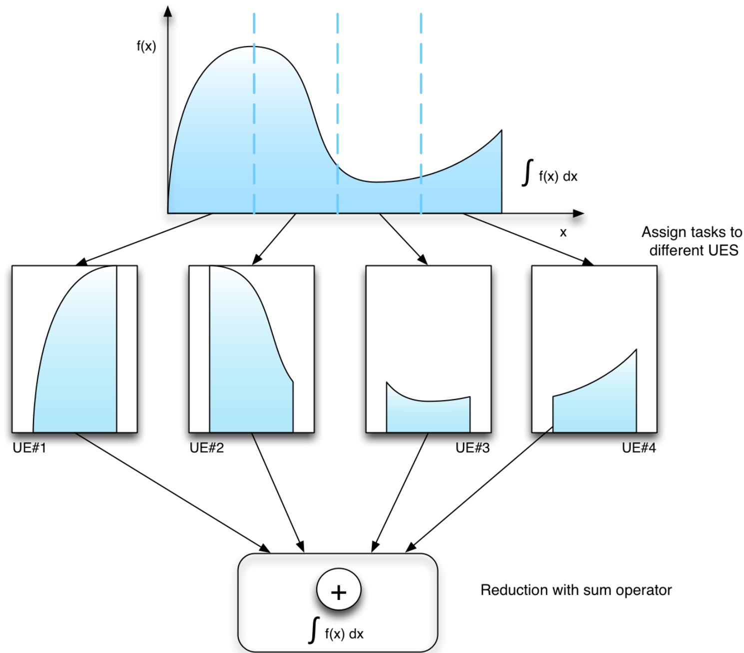
The example in Figure 3.1 divides the numerical integration problem into tasks and maps those tasks unto four different UEs.

A simple approach for combining those values involves waiting for the results from all four tasks to finish before computing the sum. While this works, it also fails to exploit any inherent parallelism between the tasks. In the example, it is actually *unnecessary* to wait for the results of the four tasks before calculating the sum.

We could do better and partially sum the results as they arrive. For instance, we could compute the partial sum from UE#1 and UE#2 and the partial sum of UE#3 and UE#4 in *parallel* and then combine both partial sums to find the total sum. We perform the summation operation on each piece of data as it arrives.

The inherent parallelism arises from the properties of the operator that is used to combine the

Figure 3.1 Numerical Integration



partial results. In our example, the summation operation is both associative[1] and commutative[2]. Those properties allow the operation to be performed as each partial result arrives. Whenever possible, a properly implemented REDUCTION allows the programmer to efficiently combine a collection of values into a single value by exploiting any inherent parallelism in the process.

Nonetheless, combining a collection of values does not always require using REDUCTION. Consider an application that stores its results in a bit array. Each UE computes the value (0 or 1) for a range of positions with no overlaps in the bit array. Using REDUCTION to combine the values would require merging the values from each UE and building up partial bit arrays. Because the value from each UE corresponds to a range of positions in the final bit array, care must be taken while merging and building up the partial bit arrays. Moreover, in message-passing environments, all those partial bit arrays incur the overhead of being sent to different UEs in order to compute the result.

Instead, for this application, it might be better to rely on the GATHER pattern. One UE, say UE#1, is responsible for merging all the results. The other UEs compute the values for non-overlapping ranges in the bit array and send their results as a tuple to UE#1. After receiving all the values, UE#1 produces the complete bit array in the desired order.

3.3 Forces

Common operator REDUCTION requires that the same operator is used to combine the values from all the different UEs. This is a common pattern in scientific computing and will fit

[1] Associative Property: (A + B) + C = A + (B + C)
[2] Commutative Property: A + B = B + A

the needs of most algorithms. However, if an algorithm requires combining the results from different UEs using different operations, then it is better to orchestrate the communication using other patterns.

Associative and/or commutativity of operator Exploiting any inherent parallelism in REDUCTION relies on the fact the operator used in combining the partial result is at least associative and/or commutative. Most operators that are used in scientific computing such as maximum, minimum, sum and product are both associative and commutative. Using REDUCTION with non-associative and non-commutative operations is possible but offers no potential parallelism.

Load-balancing Reduction operations with associative and/or commutative operators offer potential for load-balancing because the task of partially combining the values can be executed on different UEs as each partial value is computed. This is useful especially for operations that are computationally-intensive.

Floating-point numbers Care has to be taken while using REDUCTION with floating-point numbers. Certain operations such as addition on floating-point numbers are neither strictly associative nor commutative. Round-off errors can easily accumulate depending on the order in which the operator is applied to the partial values. In cases where the partial values have roughly the same magnitude, the loss of precision is usually acceptable for the programmer. However, if the partial values have vastly different magnitudes, the error could be unacceptable; programmers should not rely on REDUCTION to combine the partial values but should instead orchestrate the communication using other patterns.

3.4 Solution

Use the reduction construct if one is provided by the parallel programming environment. REDUCTION is such a common pattern in scientific computing that it is included in most parallel programming environments (see Table 3.1). The reduction construct that is provided by the programming environments has been tuned to satisfy the needs of most programmers. It is designed to exploit the inherent concurrency that is possible with associative and commutative binary operators.

Environment	Reduction Construct
MPI	`MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)`
OpenMP	#pragma omp reduction (operator: list)
Charm++	`void contribute(int nBytes, const void *data, CkReduction::reducerType type)`
CUDA	NONE
Cilk	`inlet`
FJ Framework	Generic `T reduce(Ops.Reducer<T> reducer, T Base)` method & various predefined operations (max, min, sort, etc) in the `ParallelArray` class
Intel TBB	`void parallel_reduce(...)`
Java	NONE

Table 3.1: Reduction constructs in parallel programming environments

Deciding what operator to use in combining the the collection of values is also part of the process of applying the REDUCTION pattern. Most environments already provide some built-in operators for common tasks such as determining the maximum, minimum, sum, product, etc. that programmers can use directly in their reduction operations.

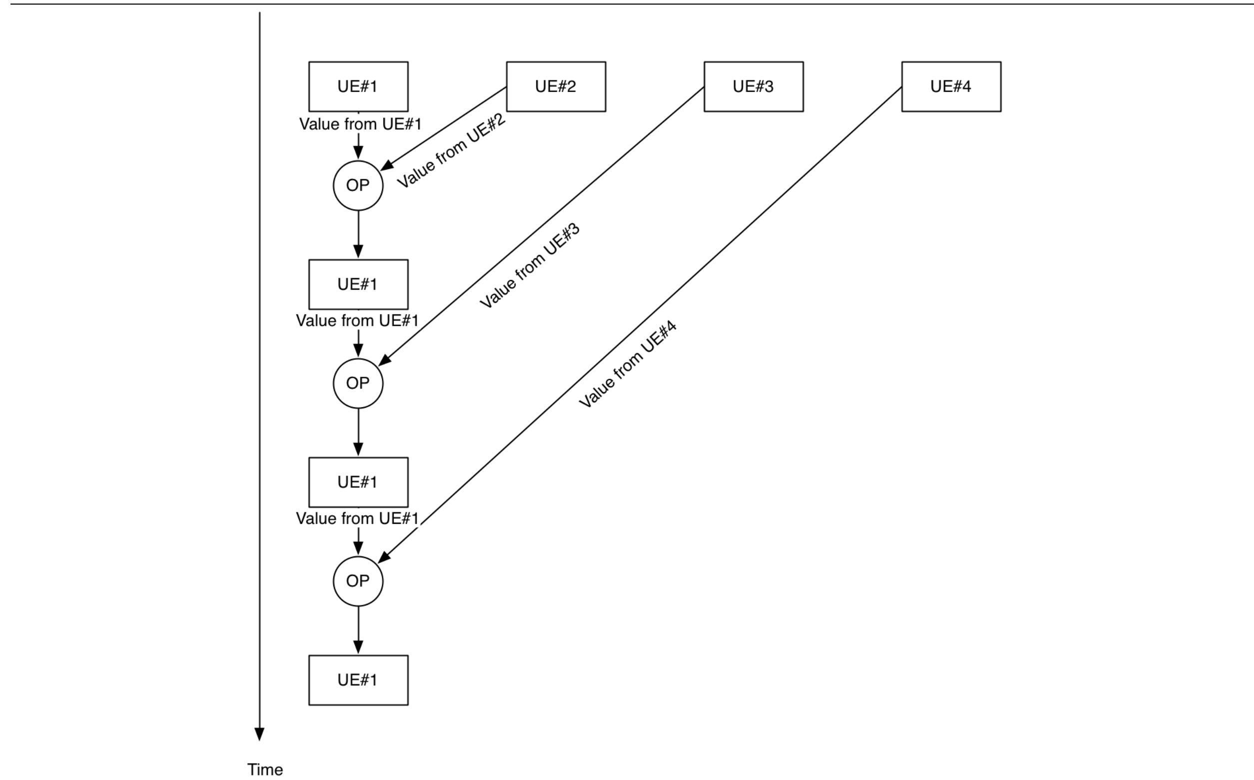
Moreover, most environments also provide support for user-defined operators that programmers can write. When writing their own operators, programmers have to consider whether their operators are associative and/or commutative. Programmers also have to decide on the data structure — single value, tuples, etc — that is used to store the partial result.

On the other hand, if the parallel programming environment does not provide a reduction construct programmers might need to write one of their own. Incidentally, the reduction construct provided by the environment might make the implicit assumption that the operator needs to be commutative and/or associative. If the operator doesn't have those those properties, programmers need to write their own reduction construct.

We summarize the two common methods — serial computation and tree-based reduction — for implementing reduction operations presented in [MSM04]. More complex implementation techniques are presented in [GGKK03].

3.4.1 Serial Computation

Figure 3.2 Serial Computation

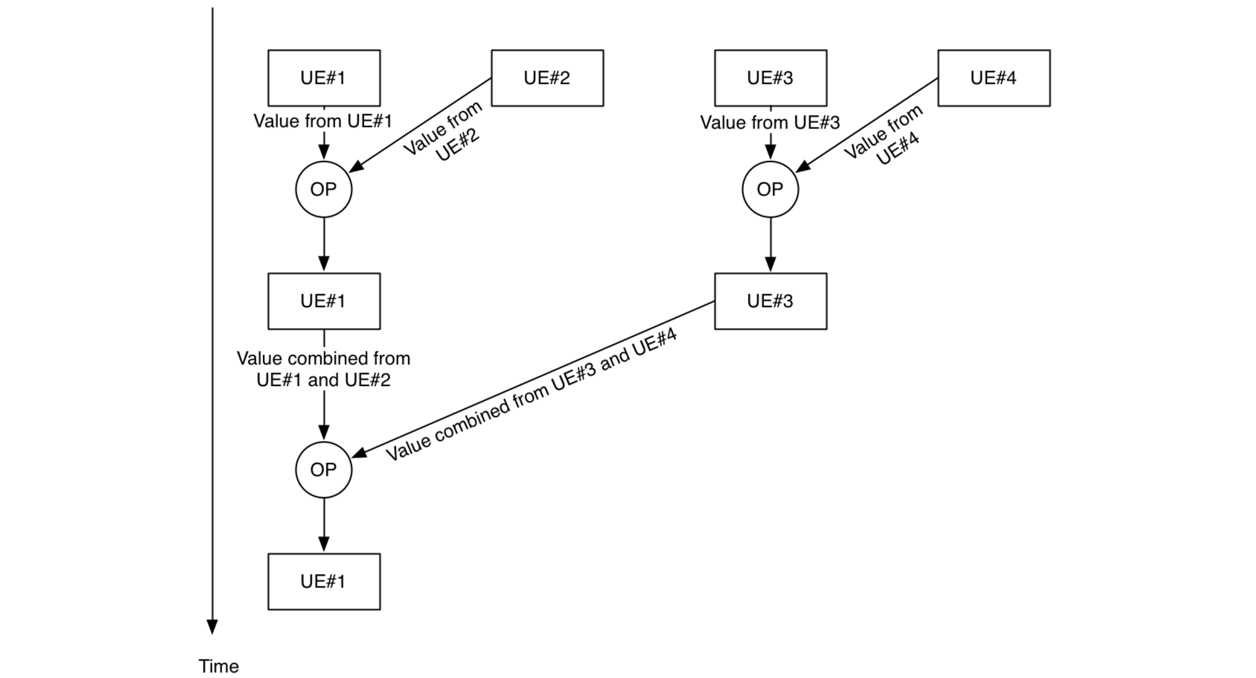


If the reduction operator is not associative then the programmer needs to "serialize" the computation. One way of doing so is shown in Figure 3.2. In this method, all UEs send their values to

UE#1 in a predetermined order. And UE#1 is in charge of combining those values using the OP operator. At the end of the computation, only UE#1 contains the combined value.

3.4.2 Tree-based Reduction

Figure 3.3 Tree-based Reduction



However, if the reduction operator is associative, we can take advantage of the inherent parallelism by performing the reduction operation in parallel using a tree-based reduction as shown in Figure 3.3.

In this specific example, we assume that we **only** have 4 UEs. UE#1 applies the OP operator to its own value and the value that it receives from UE#2. It then stores that value temporarily. Similarly, UE#3 will apply the OP operator to the value it receives from UE#4 and stores that value temporarily. Finally, UE#1 combines its value and the value it receives from UE#3 using the OP operator and stores the combined results.

3.5 Invariants

Precondition A collection of values on different UEs that need to be combined using the same operator.

Invariant The initial values on each of the different UEs.

Postcondition The values from different UEs are combined using the operator and are contained in an object on **one** of the UEs.

3.6 Examples

3.6.1 Numerical Integration with MPI

Listing 3.1 Numerical Integration Example Using `MPI_Reduce`

```
30 int main(int argc, char* argv[]) {
31
32   int my_id;
33   int number_of_processors;
34
35   //
36   // Initialize MPI and set up SPMD programs
37   //
38   MPI_Init(&argc,&argv);
39   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
40   MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
41
42   double interval = (UPPER_LIMIT - LOWER_LIMIT) / number_of_processors;
43   double my_lower_bound =  LOWER_LIMIT + my_id * interval;
44   int intervals_per_processor = NUMBER_OF_INTERVALS / number_of_processors;
45   double delta = (UPPER_LIMIT - LOWER_LIMIT) / NUMBER_OF_INTERVALS;
46
47   double my_partial_integration = calculate_integral(my_lower_bound, delta,
48                                       intervals_per_processor);
49
50   //
51   // Combine the results from different UEs
52   //
53   double total_integration;
54   MPI_Reduce(&my_partial_integration, &total_integration, 1, MPI_DOUBLE, MPI_SUM,
55     MASTER_NODE, MPI_COMM_WORLD);
56
57   if(my_id == MASTER_NODE) {
58           printf("The result of the integration is %f\n",total_integration);
59   }
60
61   MPI_Finalize();
62
63   return 0;
64 }
```

Listing 3.1 shows an example of using the built-in `MPI_Reduce` construct in MPI to solve the numerical integration problem discussed in Section 3.2 and illustrated in Figure 3.1.

Our strategy involves dividing the domain of the function into several non-overlapping sub-domains. The numerical integration for each sub-domain is calculated in parallel and their values are reduced to obtain the value of integrating the function over the entire domain.

The domain (`UPPER_LIMIT - LOWER_LIMIT`) of the function to integrate is divided evenly between the number of available UEs as determined by the `MPI_Comm_size` function. Each UE handles a particular sub-domain and none of the sub-domains overlap. In our example, each UE is alloted the same number of intervals i.e. dx to calculate[3].

[3] A better scheme could use the rate of change of the function to determine how many intervals to calculate for

Each UE starts integrating beginning from its own lower bound. The partial result is stored in the my_partial_integration variable. After the partial result is available, each UE calls the MPI_Reduce function. The following code snippet shows the arguments to the function.

```
53  double total_integration;
54  MPI_Reduce(&my_partial_integration, &total_integration, 1, MPI_DOUBLE, MPI_SUM,
55    MASTER_NODE, MPI_COMM_WORLD);
```

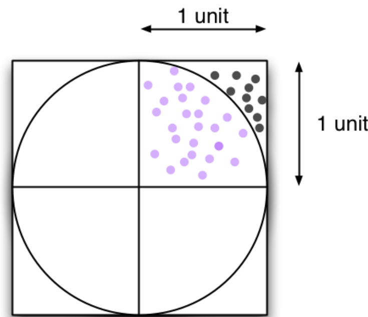
The MPI_SUM argument tells the reduction operation to sum each of the my_partial_integration variables on each UE and to store the result in the total_integration variable. The result will be stored on the MASTER_NODE UE.

Internally, the MPI implementation would take advantage of the associative and commutative properties of the MPI_SUM operation and perform the summation in parallel yielding better performance.

3.6.2 Monte Carlo Simulation with MPI

Listing 3.2 shows another example of using the built-in MPI_Reduce construct in MPI to estimate the value of π using the Monte Carlo pattern.

Figure 3.4 Estimating π



Estimating π using this method involves randomly "throwing" points in the first quadrant of the square shown in Figure 3.4. Some of those points will land inside the circumference of the unit circle and some will land outside the circumference. The ratio of the points inside the circle to total points thrown gives an estimate for π based on the following formula:

$$\begin{aligned}
\frac{\text{Area of quarter unit circle}}{\text{Area of quarter of square}} &= \frac{\frac{(\pi)(1\ \text{unit})^2}{4}}{(1\ \text{unit})^2} \\
&\approx \frac{\text{Points in circle}}{\text{Total points in first quadrant}} \\
\Rightarrow \pi &\approx 4 * \frac{\text{Points in circle}}{\text{Total points in first quadrant}}
\end{aligned}$$

The function count_points_in_circle(int *) on lines 17 - 27 of Listing 3.2 shows how the points are randomly generated and tested to see if they are inside the circumference.

each sub-domain. A slower changing function would require less intervals.

Listing 3.2 Estimating π Example Using `MPI_Reduce`

```
17 int count_points_in_circle(int* stream_id){
18   int i, my_count = 0;
19   double x, y, distance_squared;
20   for(i = 0; i < ITERATIONS; i++) {
21     x = (double)sprng(stream_id);
22     y = (double)sprng(stream_id);
23     distance_squared = x*x + y*y;
24     if(distance_squared <= RADIUS) my_count++;
25   }
26   return my_count;
27 }
28
29 int main(int argc, char* argv[]) {
30
31   int my_id;
32   int number_of_processors;
33
34   //
35   // Initialize MPI and set up SPMD programs
36   //
37   MPI_Init(&argc,&argv);
38   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
39   MPI_Comm_size(MPI_COMM_WORLD, &number_of_processors);
40
41   //
42   // Initialize pseudo parallel random number generator
43   //
44   generate_seed();
45   int* stream_id = init_sprng(SPRNG_LFG, my_id, number_of_processors, rand(),
46                               SPRNG_DEFAULT);
47
48   int my_count = count_points_in_circle(stream_id);
49
50   //
51   // Combine the results from different UEs
52   //
53   int total_count;
54   MPI_Reduce(&my_count, &total_count, 1, MPI_INT, MPI_SUM, MASTER_NODE,
55     MPI_COMM_WORLD);
56
57   if(my_id == MASTER_NODE) {
58     double estimated_pi = (double)total_count /
59                                   (ITERATIONS * number_of_processors) * 4;
60     printf("The estimate of pi is %g\n", estimated_pi);
61   }
62
63   MPI_Finalize();
64
65   return 0;
66 }
```

Each UE is initialized to perform the same algorithm in parallel albeit with different random points. To ensure that each UE receives a set of random points with minimal chances of repetition, we rely on the SPRNG[SPR] pseudo parallel random number generator library.

After each UE has successfully completed the algorithm, `MPI_Reduce` function is called to combine the estimates of π from each UE.

3.6.3 Histogram Accumulation with CUDA

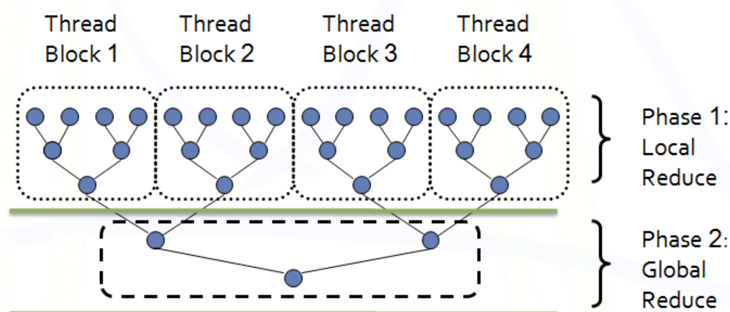
Listing 3.3 shows an example of using CUDA for accumulating histograms of an integer array. CUDA does not have a built-in construct for performing a reduction operation. Therefore, programmers have to devise their own approach. And because CUDA expects a SIMD programming model, the reduction implementation has to conform with that model to achieve good performance.

The problems starts with an input array of size `n` consisting of integers ranging from 0 to `HISTOLAYER - 1`. We wish to determine the frequency of each integer in the input array.

To perform the computation, we generate n threads — one for each element in the input array. Each thread represents a histogram of one particular integer value; the histogram accumulated by each thread will have value 1 for one particular integer, and value 0 for all other integers. We apply the tree-based reduction introduced in section 3.4.2 to accumulating the values from each thread and construct the overall histogram.

For CUDA programming, threads are packed into independent thread blocks. Threads within a thread block can synchronize with each other using the intrinsic function `__syncthreads()`. The `__syncthreads()` function acts as a barrier. However, there is no explicit synchronization methods for synchronizing threads in different thread blocks. We propose a two phase reduction in listing 3.3. In the first phase, we do the local reduction within each thread block. In the second phase, we summarize all the local reduction results in one thread block, and then do the overall global reduction. The two phase reduction is summarized in Figure 3.5. Depending on the size of the input array, sometimes more phases are required for better performance.

Figure 3.5 Two phase reduction using CUDA.



Listing 3.3 is the routine for the histogram accumulation. Listing 3.4 is the routine for the first phase reduction. Listing 3.5 is the routine for the second phase reduction. Listing 3.6 is the routing for tree-based reduction.

Listing 3.3 Routine for Histogram Accumulation

```
117 void CalcHistoGram(int n, int* devFeature, int* devHistoResult, int reductionLevel)
118 {
119   int blockNum = (n + BLOCKSIZE - 1) / BLOCKSIZE;
120   dim3 blockDim(BLOCKSIZE, 1);
121   dim3 gridDim(blockNum, 1);
122   int* devHisto = 0;
123   cudaMalloc((void**)&devHisto, blockNum*HISTOLAYER*sizeof(int));
124
125   //First Phase Reduction
126   CalcHistoPerBlock<<<gridDim, blockDim>>>(n, devFeature, devHisto, reductionLevel);
127   dim3 oneGrid(1, 1);
128
129   //Second Phase Reduction
130   CalcHistoPerGrid<<<oneGrid, blockDim>>>(blockNum, devHisto, devHistoResult, reductionLevel);
131   cudaFree(devHisto);
132 }
```

Listing 3.4 Routine for First Phase Reduction

```
46 __global__ void CalcHistoPerBlock(int n, int* inputArray, int* blockHistogram,
47   int reductionLevel)
48 {
49   int index = IMUL(blockDim.x, blockIdx.x) + threadIdx.x;
50   __shared__ int devResult[BLOCKSIZE*HISTOLAYER];
51
52   if (index < n)
53   {
54     //Initialization
55     int color = inputArray[index];
56     for (int i = 0; i < color; i++)
57       devResult[BLOCKSIZE * i + threadIdx.x] = 0;
58     devResult[color * BLOCKSIZE + threadIdx.x] = 1;
59     for (int i = color + 1; i < HISTOLAYER; i++)
60       devResult[BLOCKSIZE * i + threadIdx.x] = 0;
61
62     __syncthreads();
63     TreeReduction(devResult, n, reductionLevel);
64
65     if (threadIdx.x == 0)
66     {
67       for (int i = 0; i < HISTOLAYER; i++)
68       {
69         int resultIdx = i*gridDim.x+blockIdx.x;
70         blockHistogram[resultIdx] = devResult[i*BLOCKSIZE];
71       }
72     }
73   }
74 }
```

Listing 3.5 Routine for Second Phase Reduction

```
76 __global__ void CalcHistoPerGrid(int blockNumber, int* blockHistogram, int*
77   finalHistogram, int reductionLevel)
78 {
79   __shared__ int devResult[BLOCKSIZE * HISTOLAYER];
80 
81   //Initialization
82   if (threadIdx.x < blockNumber)
83   {
84     for (int i = 0; i < HISTOLAYER; i++)
85     {
86       devResult[i*BLOCKSIZE + threadIdx.x] = 0;
87     }
88   }
89 
90   //Apply serial reduction for taskPerTh elements
91   int taskPerTh = (blockNumber + BLOCKSIZE - 1)/BLOCKSIZE;
92 
93   for (int i = 0; i < taskPerTh; i++)
94   {
95     int index = threadIdx.x + i*BLOCKSIZE;
96     if (index < blockNumber)
97     {
98       for (int j = 0; j < HISTOLAYER; j++)
99       {
100        devResult[j*BLOCKSIZE + threadIdx.x] += blockHistogram[index + j *
101                                                     blockNumber];
102      }
103    }
104  }
105  __syncthreads();
106  TreeReduction(devResult, blockNumber, reductionLevel);
107 
108  if (threadIdx.x == 0)
109  {
110    for (int i = 0; i < HISTOLAYER; i++)
111    {
112      finalHistogram[i] = devResult[i*BLOCKSIZE];
113    }
114  }
115 }
```

3.7 Known Uses

In addition to the examples mentioned, Reduction is used in various scientific computations such as dot product calculations and L2 Norm[L2N].

3.8 Related Patterns

Barrier A Barrier can be implemented as a Reduction with a null-operator. The programmer does not care about the values on each UE or the operator to be used in combining them. As

Listing 3.6 Tree-Based Reduction

```
22 __device__ void TreeReduction(int* devArray, int n, int reductionLevel)
23 {
24   //Tree-Based Reduction
25   int mask = 1;
26   for (int level= 0;level< reductionLevel; level++)
27   {
28     if ((threadIdx.x & mask) == 0)
29     {
30       int index1 = threadIdx.x;
31       int index2 = (1 << level) + threadIdx.x;
32       if (IMUL(blockDim.x, blockIdx.x) + index2 < n)
33       {
34         for (int i= 0; i < HISTOLAYER; i++)
35         {
36           devArray[BLOCKSIZE * i + index1] += devArray[BLOCKSIZE * i + index2];
37         }
38       }
39
40     }
41     mask = (mask<<1)|1;
42     __syncthreads();
43   }
44 }
```

the reduction operation proceeds, all UEs block while waiting for each other to reduce their values. Once all UEs have completed the reduction, they can proceed with their computation again. A null-operator REDUCTION acts as a barrier that synchronizes all UEs. This technique is used in Charm++ which does not provide a BARRIER construct, only a REDUCTION construct.

Message-Passing In message-passing environments such as MPI and Charm++, REDUCTION is always implemented using the MESSAGE-PASSING pattern[OPL].

Bibliography

[Agh86] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[Boy97] Robert L. Boylestad. *Introductory Circuit Analysis.* Prentice Hall, 8th edition, 1997.

[Cha] Charm++ Parallel Programming Model. http://charm.cs.uiuc.edu/.

[Cil] The Cilk Project. http://supertech.csail.mit.edu/cilk/.

[CUD] NVIDIA CUDA. http://www.nvidia.com/object/cuda_what_is.html.

[GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing.* Addison Wesley, 2003.

[Jav] Java. http://www.java.com/en/.

[JSR] JSR-166. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[L2N] L2-Norm. http://mathworld.wolfram.com/L2-Norm.html.

[MPI] Message Passing Interface Forum. http://www.mpi-forum.org/.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming.* Addison Wesley, 2004.

[Ope] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[OPL] Berkeley Pattern Language for Parallel Programming. http://parlab.eecs.berkeley.edu/wiki/patterns/patterns.

[SPR] Scalable Parallel Pseudo Random Number Generators Library. http://sprng.cs.fsu.edu/.

[TBB] Intel threading building blocks. http://www.threadingbuildingblocks.org/.