The Suzaku Pattern Programming Framework

Barry Wilkinson
University of North Carolina, Charlotte
9201 University City Blvd.
Charlotte, NC 28223, USA
abw@uncc.edu

Clayton Ferner
University of North Carolina, Wilmington
601 S. College Rd.
Wilmington, NC 28403, USA
cferner@uncw.edu

***Abstract*— Suzaku is a pattern programming framework that enables programmers to create pattern-based parallel MPI programs without writing the MPI message-passing code implicit in the patterns. The purpose of this framework is to simplify message-passing programming and create better structured programs based upon established parallel design patterns. The focus for developing Suzaku is on teaching parallel programming. This paper covers the main features of Suzaku and describes our experiences using it in parallel programming classes.**

Keywords-pattern programming; parallel programming; distributed computing; MPI; OpenMP

I. INTRODUCTION

Parallel programming, i.e. writing programs that use multiple computers and processors collectively to solve problems at greater speed, has a very long history but still can be very challenging. The idea of using multiple computers for increased speed is obvious in this age of ubiquitous low cost computers, but getting multiple simultaneously executing programs to collaborative is a very difficult venture. Writing a single program is already a challenge with the complexities we ask of it. Now having multiple collaborating programs, the challenge is far greater. Programs can deadlock while waiting for other programs. The overall behavior may not be deterministic. When programs are executed together one cannot say whether one statement in one program executed before or after a statement in another program in the general case.

Although parallel programming is known to be very challenging, the usual way of programming is still to use low-level message-passing libraries such as MPI in which the programmer explicitly specifies the message passing, and low-level thread libraries such as Pthreads or slightly higher level OpenMP. There have been attempts to raise the level of abstraction over the years, for example using parallelizing compilers that recognize parallelism in sequential programs. That approach was less than successful because how we might write an efficient parallel program is not necessarily the same as how we write an efficient sequential program. For example, there are parallel algorithms that can be employed. There have also been attempts at creating parallel languages and parallel extensions to sequential languages and again these attempts have not found universal appeal with most programmers falling back on using low-level libraries. It now appears better not to abstract the parallelism away from the programmer completely. The programmer needs some control on how a parallel program is constructed, but still needs a way to write potentially large parallel programs with some degree of certainty that they are correct, scalable, and maintainable.

In this paper, we draw upon the concept of design patterns in software engineering that establishes good programming practices [1] [2]. In software engineering, a design pattern is a reusable solution to commonly occurring problems [3]. Design patterns provide a guide to best practices but not a final implementation. They provide scalable design structures and one can reason more easily about the resulting programs. Also, in the bigger picture of teams of programmers and interacting programs, design patterns help programmers that did not write the code understand how the program was written. Design patterns historically were associated with object-oriented programming although this is not necessary. Design patterns have been applied to specific application areas such as games [4] and .NET programming [5].

Design patterns can be applied to parallel programming. Parallel programming design patterns describe multiple communicating processes or threads executing at the same time. Here we will focus on communicating message-passing processes and describe them in that fashion although the patterns are applicable to communicating threads. The programmer begins constructing his program by selecting an established pattern that provides a known structure. Patterns are particularly useful for the complexities of parallel and distributed computing. As we will show, the design pattern approach, when applied to parallel programming, can lead to an automatic conversion into executable code avoiding low-level programming altogether. We will use the phrase *pattern programming* to describe writing pattern-based parallel programs with tools that hide the low-level code. Our work focuses on developing pattern programming tools specifically to teach parallel programming at the undergraduate level. Although parallel programming has been taught in undergraduate computer science programs for many years, it has recently become an imperative for all computer science programs to introduce parallel programming at the lower levels with the publication of the 2013 IEEE/ACM “Curriculum Guidelines for Undergraduate Degree Programs in Computer Science” [6], which specifies parallel and distributed computing as a new required knowledge area.

The paper is organized as follows. Previous work is briefly reviewed in Section II. The structure of our pattern

programming framework Suzaku is introduced in Section III. Low-level Suzaku patterns and routines are described. Higher-level Suzaku patterns are described in Section IV (workpool), Section V (iteration synchronous patterns), Section VI (pipeline), and Section VII (generalized patterns). In Section VIII, we describe our classroom experiences, with brief comments about the relationship of the materials with the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing in Section IX. Section X provides conclusions.

II. Previous Work

Applying design patterns to parallel programming has been explored in several university and industrial research projects including [7][8] and is the subject of books [9][10]. Villalobos explored its use in Grid computing and developed a Java-based pattern programming framework called Seeds [11] from which we draw a great deal of inspiration. He created a framework that enabled programmers to construct fully distributed programs without any low-level message-passing code. We have used the Seeds framework in undergraduate parallel programming courses [12] introducing the advantages of pattern programming to students. The particular advantage of Seeds is that it is Java based. Most students already know Java, resulting in a low learning curve. Seeds also self-deploys on any single computer or distributed platform. Objects are used to contain the data exchanged between processes and a very elegant programming interface exists with three principal methods, diffuse, compute, and gather. We will use a similar elegant interface but with the C programming language, which leads to a direct path to the lower-level MPI. Although objects are not used, as we shall describe, we can simulate their use with routines that are similar in effect to the get and put methods in Seeds.

III. Suzaku Structure and Low-level Patterns

Suzaku provides routines that enable programmers to create pattern-based parallel programs without low-level MPI code. After compilation, the program is executed as an MPI program. However all the complexities of MPI routines and their parameters are avoided and well-structured parallel programs can be created. The overall programming structure of a Suzaku program is shown in Fig. 1 and is purposely similar to OpenMP but using processes instead of threads. We draw upon the OpenMP programming structure as it is usually taught in parallel programming classes and OpenMP is simple and very easy to learn. However, it is not necessary to know OpenMP first. With the process-based MPI model, there is no shared memory. As shown in Fig. 1, the program begins by declaring variables that are duplicated in each process. The purpose of SZ_Init(p) is to initialize the message-passing environment. SZ_Init(p) is required and sets p to be the number of processes. (As a macro, the parameter, p, does not require an address operator, see later.) After SZ_Init(p), all code is only executed by the master process, just as in OpenMP where a single thread executes the code initially. One or more parallel sections can be created with the SZ_Parallel_begin and SZ_Parallel_end

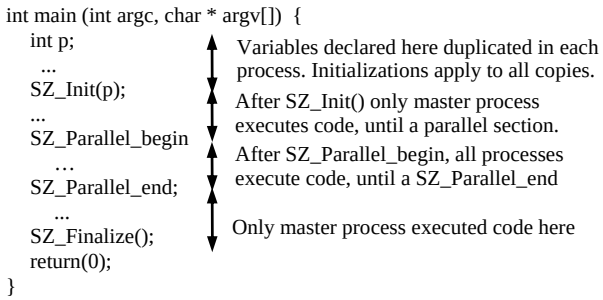

Figure 1. Suzaku program structure.

constructs. All the processes will execute the code within a parallel section, including the master process. Outside parallel sections, the computation is only executed by the master process. The purpose of SZ_Finalize() is to close the message-passing environment and it is required. No Suzaku routines must be placed after SZ_Finalize(). All processes still exist and any code placed after SZ_Finalize() will be executed by all processes. Typically one does not want to do that. This is the same as MPI_Finalize() in MPI.

The routines available in Suzaku are divided into low-level routines that implement basic functions or low-level message-passing patterns and routines that implement high-level patterns. The low-level routines are given in Table 1. The low-level routines in Table 1 are implemented with macros to enable the type and size of arguments to be unspecified and determined within the macros. The basic data type is a double and for those routines that indicate double argument(s), that is the only type allowed although the size does not need to be specified. The pointer arguments can be for variables (using the & address operator) or arrays. Arrays need to be declared either statically or as variable length arrays (i.e. not dynamically allocated) to be able to establish the size without specification. All message-passing routines are made synchronous for ease of use, which means all the processes involved do not return until the whole operation has been completed. This is not the same as MPI collective routines. Two routines in Table 1, SZ_Point_to_point() and SZ_Broadcast(), allow data to be sent with a wide range of data types for added flexibility—characters, integers, doubles, one-dimensional arrays of

TABLE 1. Low-level Suzaku routines

SZ_AllBroadcast(double *a)
SZ_Barrier()
SZ_Broadcast(void *a)
SZ_Finalize()
SZ_Gather(double *a, double *b)
SZ_Get_process_num()
SZ_Init(int p)
SZ_Master <structured block>
SZ_Parallel_begin … SZ_Parallel_end;
SZ_Point_to_point(int p1, int p2, void *a, void *b)
SZ_Process(int ID) <structured block>
SZ_Scatter(double *a, double *b)
SZ_Wtime()

characters, integers, and doubles, and multi-dimensional arrays of doubles. The type and size of the data does not have to be specified. Fig. 2 shows a program that illustrates various types and sizes being used without specification with SZ_Point_to_point().

A basic parallel pattern is the master-slave pattern. In this pattern, the computation is divided into parts, which are then distributed to slaves for each slave to perform one part and return their result. The master-slave pattern can often be implemented with the low-level broadcast, scatter, and gather routines. The function that the slaves execute is placed after the scatter and broadcast and before the gather. For efficient mapping to collective MPI routines, the master also acts as one of the slaves. Matrix multiplication using the master-slave pattern is shown in Fig. 3. SZ_Scatter() determines the size of each transfer by the size of the destination and SZ_Gather() determines the size of each transfer by the size of the source.

The matrix multiplication algorithm in Fig. 3 is given to illustrate the use of the broadcast, scatter, and gather routines. It is not necessarily the best way to do matrix multiplication in parallel because we are copying the entire B array to each process rather than distributing parts of the B array to processes. Fig. 3 is very easy to explain to students and easy to implement given that broadcast, scatter, and gather routines are available. We use the algorithm to give an example of the master-slave pattern using these routines and to start the discussion on partitioning a problem. Scatter and gather can confuse students as they rely on understanding how arrays are stored in memory in row-major order and this example helps clarify the routines. Performing more efficient matrix multiplication in parallel with large array sizes requires a deeper understanding of computer architecture, including the effects of cache memory and memory size. We do explore this later in our parallel programming course.

The low-level routines can be used to parallelize many sequential programs easily. As we will discuss in Section VIII, we ask our students to write a sequential program for the astronomical *N*-body problem and then ask them to

```
int main(int argc, char *argv[]) {
    char m[20], n[20];
    int p, x, y, xx[5], yy[5];
    double a, b, aa[10], bb[10], aaa[2][3], bbb[2][3];
    ...
    SZ_Init(p);                 // initialize environment,
    SZ_Parallel_begin   // parallel section - from process 0 to process 1:
       SZ_Point_to_point(0, 1, m, n);        // send a string
       SZ_Point_to_point(0, 1, &x, &y);    // send an int
       SZ_Point_to_point(0, 1, &a, &b);    // send a double
       SZ_Point_to_point(0, 1, xx, yy);      // send 1-D array of ints
       SZ_Point_to_point(0, 1, aa, bb);      // send 1-D array of doubles
       SZ_Point_to_point(0, 1, aaa, bbb);  // send 2-D array of doubles
    SZ_Parallel_end;    // end of parallel section
    ...
    SZ_Finalize();
    return 0;
}
```

Figure 2. Point-to-point pattern with various data types.

```
#define N 256
int main (int argc, char *argv[] ) {
    int i, j, k, p, blksz;
    double A[N][N], B[N][N], C[N][N], sum;
     ...
    SZ_Init(p);
    ...
    SZ_Parallel_begin
       blksz = N/p;                  // assumes N is a multiple of p
       double A1[blksz][N];          // for slaves to hold scattered A
       double C1[blksz][N];          // for slaves to hold their result
       SZ_Scatter(A,A1);             // scatter blksz rows of A array
       SZ_Broadcast(B);              // broadcast B array
       for (i = 0 ; i < blksz; i++) { // matrix multiplication.
         for (j = 0 ; j < N ; j++) {
           sum = 0;
           for (k = 0 ; k < N ; k++) {
              sum += A1[i][k] * B[k][j];
           }
           C1[i][j] = sum;
         }
       }
       SZ_Gather(C1,C);              // gather results, blksz rows of C1
    SZ_Parallel_end;
       ...
    SZ_Finalize();
    return 0;
}
```

Figure 3. Matrix multiplication using master-slave pattern.

parallel it with the master-slave pattern. It turns out to be easy to do just by adding a few Suzaku routines and removing one sequential loop. The master-slave implementation has a significant disadvantage that the number of bodies must be the same as the number of processes. One process calculates the position and velocity of one body at each time interval. However, master-slave pattern leads us onto more powerful patterns, which we discuss next starting with the workpool pattern.

IV. WORKPOOL PATTERN

The workpool pattern is a very widely applicable pattern. It is similar to the master-slave pattern but has a task queue as shown in Fig. 4. A task from the task queue is given to each slave by the master process. When a slave finishes a task and returns the result, it is given another task from the task queue until the task queue is empty. At that point, the master waits until all outstanding results are returned. The termination condition is when the task queue is empty and all outstanding results are collected. The number of slaves does not need to be the same as the number of tasks. Indeed one typically would limit the number of slaves to the number of the physical cores available (or double that number with hyperthreading). The number of tasks could be much greater and would be for many applications.

A very important feature of the workpool is its load balancing quality. Slaves are kept busy with tasks irrespective of the speed of the slaves. Faster slaves will return quicker but are given more work. It is up to the master to determine how much work a slave receives. The tasks do not need to be all of the same computational effort although in our basic implementation the tasks are not differentiated.

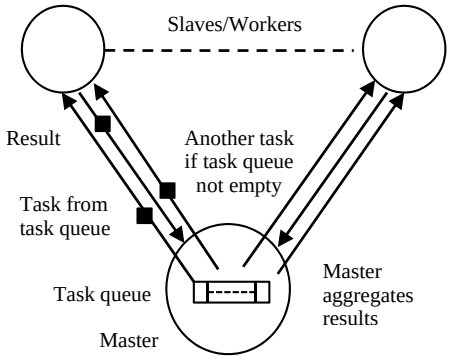

Figure 4. Workpool pattern.

The programmer's interface for the Suzaku workpool is modeled on the Seeds framework. The programmer must implement four routines:

- init() Sets up initial values. Called once by all processes at the beginning of the computation.
- diffuse() Generates the next task when called by the master.
- compute() Executed by a slave, takes a task generated by diffuse and generates the corresponding result.
- gather() Accepts a slave's result and develops the final answer. Called by the master.

The framework generates a task ID for each task from zero onwards. The task ID is an input parameter for diffuse(), compute(), and gather(). It is available to the programmer but might or might not be used depending upon the application. The number of tasks and actual tasks are determined by the programmer, which provide great flexibility on how best to divide the problem taking into account the task computation and the communication overhead. The number of tasks is established in init(). The programmer determines what data will be sent in each task in diffuse() and what is returned in compute(). OpenMP provides so-called static, dynamic, and guided scheduling algorithms with their parallel for directive to divide for loops into parallel parts. The scheduling algorithms attempt to share the work in an equable way. The expectation is that all processors are similar in capability, which would typically be the case in a shared-memory system. In contrast, the Suzaku workpool provides automatic load balancing with its task queue that takes into account that each slave may perform at different speeds but does require the programmer to determine the tasks performed by each slave.

Three versions of the workpool have been implemented, of increasing sophistication. The basic version (version 1) limits task data to be held in a one-dimensional array of doubles, which is sufficient for simple applications and is the most efficient. However, the Seeds workpool is much more powerful. In Seeds, multiple data items of different types can be held in a task object using a Java hashmap that associates a key (a string) with each data item. The programmer refers to the data by the key and uses the put method to add data to the task in diffuse() and get method to extract the data in compute(). This concept has been modeled in the Suzaku workpool version 2 using C but without objects. Fig. 5 shows sample code. The workpool is implemented in SZ_Workpool2() and the four required routines from the programmer are specified as function parameters. As such, they can be renamed to accommodate multiple workpools and other patterns in a single program. This feature is not available in Seeds.

Two routines are provided to the programmer, SZ_put() to pack the data into the task and SZ_get() to retrieve the

```
void init(int *tasks) {  // sets number of tasks
   *tasks = 4;
   return;
}
void diffuse(int taskID) {
   char w[] = "Hello World";
   int x;
   double y = 5678, z[2][3];
   …
   SZ_Put("w",w);
   SZ_Put("x",&x);
   SZ_Put("y",&y);
   SZ_Put("z",z);
   return;
}
void compute(int taskID) {
   char w[12] = "----------";
   int x = 0;
   double y = 0, z[2][3];
   …
   SZ_Get("z",z);
   SZ_Get("x",&x);
   SZ_Get("w",w);
   SZ_Get("y",&y);
   … // compute
   SZ_Put("xx",&x);
   SZ_Put("yy",&y);
   SZ_Put("zz",z);
   SZ_Put("ww",w);
   return;
}
void gather(int taskID) {
   char w[12] = "----------";
   int x = 0;
   double y = 0, z[2][3];
   …
   SZ_Get("ww",w);
   SZ_Get("zz",z);
   SZ_Get("xx",&x);
   SZ_Get("yy",&y);
   return;
}
int main(int argc, char *argv[]) {
   int p;
   SZ_Init(p);

   SZ_Parallel_begin
      SZ_Workpool2(init,diffuse,compute,gather);
   SZ_Parallel_end;

   SZ_Finalize();
   return 0;
}
```

Figure 5. Program using Suzaku workpool version 2.

data from the task. These routines can accept characters, integers, doubles, one-dimensional arrays of characters, integers, or doubles, and multi-dimensional arrays of doubles. The type and size do not have to be specified. The first parameter in SZ_put() and SZ_get() is a programmer-defined string to identify the data. The second parameter is the associated data. SZ_put() and SZ_get() are macros to enable arguments without type and size. Once the type and size have been determined, they call internal routines. MPI pack and unpack routines are then used. Each message sent includes information that maps the data to their names so that each message is self-contained with all the information the destination needs to extract the data. It is possible for each task to have completely different named data items although the application code would need to differentiate between the names when the task is received.

Version 3 of the workpool extends version 2 considerably. This version of the workpool implements what we call a *dynamic workpool* in which new tasks can be added to the task queue during the computation as might be needed for search problems such as the shortest path problem. To differentiate, we use the term *static workpool* when the queue holds a fixed number of tasks. Version 3 has been implemented in several ways including with multiple threads and multiple processes for the master. Fig. 6 shows our dynamic workpool algorithm using a single process for the master, which turns out to be very reliable and without the need for critical sections to handle contention for the task queue. Initially the task queue is loaded with at least one task by the routine init(). Then a task is retrieved from the task queue, diffuse() is executed, and the complete task with any addition information added by diffuse() is sent to a free slave if there is one. When there are no more free slaves or no more tasks, the master process waits for one slave to return a result. Slaves accept tasks, execute compute(), and return results, which could include new tasks. The master picks up the results of one slave, and executes the gather() routine. The gather routine might find new tasks to add to the task queue. The master then repeats the complete sequence taking tasks from the task queue and sending tasks to free slaves, etc. A slave is considered free when it has returned its result for a task it was given and the result has been collected by the master. The sequence stops when there are no new tasks and all slaves are free. Then all slaves are terminated with termination messages from the master and the master terminates.

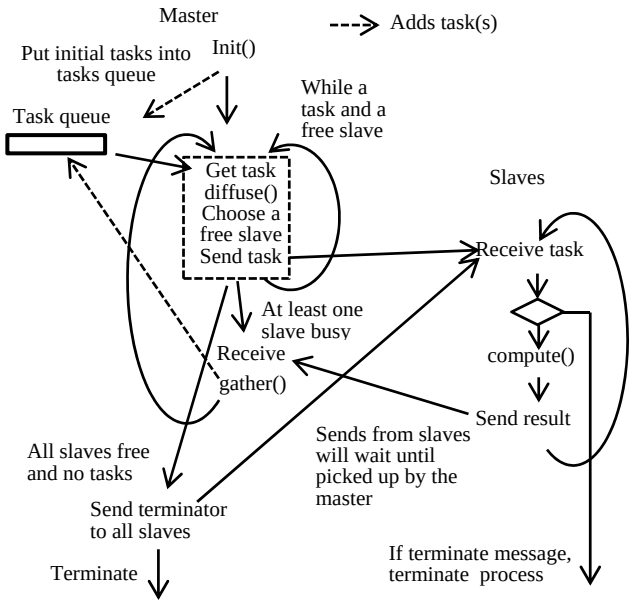

Figure 6. Dynamic workpool algorithm.

A sample program is given in Fig. 7 to solve the shortest

```
#define N 6       // number of nodes
int w[N][N], dist[N], newdist_j;  // Adjacency matrix, dist.
void init(int *T) {
   …                              // initialize dist[], w[][]
   SZ_Master {
      SZ_Insert_task(0);          // insert first node 0 into queue
   }
   return;
}
void diffuse(int taskID) {         // diffuse attaches current distances
   SZ_Put("dist",dist);            // from global array dist[] in master
   return;
}
void compute(int taskID) {
   int i, j, new_tasks[N];         // max of N new tasks
   SZ_Get("dist",dist);            // update array dist[] in slave
   for (i = 0; i < N; i++) new_tasks[i] = 0;
   i = 0;
   for (j = 0; j < N; j++)         // Moore's algorithm
      if (w[taskID][j] != -1) {
         newdist_j = dist[taskID] + w[taskID][j];
         if (newdist_j < dist[j]) {
            dist[j] = newdist_j;
            if (j < N-1)          // do not add last vertex (destination)
               new_tasks[i] = j;
               i++;
         }
      }
   SZ_Put("result",new_tasks);
   SZ_Put("dist",dist);
   return;
}
void gather(int taskID) {
   int i,dist_recv[N],new_tasks[N];  // max of N new task
   SZ_Get("result",new_tasks);       // get the first added task
   SZ_Get("dist",dist_recv);
   for (i = 0; i < N; i++)
      if (dist_recv[i] < dist[i]) dist[i] = dist_recv[i];
   for (i = 0; i < N; i++)
      if (new_tasks[i] != 0) SZ_Insert_task(new_tasks[i]);
   return;
}
int main(int argc, char *argv[]) {
   int p;
   SZ_Init(p);

   SZ_Parallel_begin
      SZ_Workpool3(init,diffuse,compute,gather);
   SZ_Parallel_end;
   …                                 // print final results in dist[]

   SZ_Finalize();
   return 0;
}
```

Figure 7. Shortest path problem using a dynamic workpool.

path problem using Moore's algorithm. This problem comes from [13]. The routines SZ_Put() and SZ_Get() are available from workpool version 2 to add data to a task and extract the data. In addition one new routine, SZ_Insert_task(), is provided to add a task to the task queue.

V. Iterative Synchronous Patterns

When forming a complete program, often a pattern is repeated until a termination condition occurs. At the end of each iteration, there is a synchronization point where all processes need to wait for each other before continuing. The termination condition typically is when all the computed values have converged sufficiently on the solution or a specific number of iterations have occurred. We call these patterns *iterative synchronous patterns*. An example is the iterative synchronous workpool pattern shown in Fig. 8, which we ask students to use to solve the *N*-body problem. Each iteration is one time interval of the simulation. Note iterative synchronous patterns consist of two patterns merged together sequentially if we call iteration a pattern. Merging patterns can be more general using a *pattern operator* to combine patterns [14].

VI. Pipeline Pattern

In the pipeline pattern, the computation is divided into a series of tasks that have to be performed one after the other, with the result of one task passed on to the next task. The pipeline pattern can be compared to an assembly line in manufacturing and as in an assembly line, a pipeline generally makes sense to use if we have multiple computations each of which can be divided into a series of sequential tasks. Hence the pipeline is usually an iterative synchronous pattern in which the pipeline is within an iteration loop as illustrated in Fig. 9. The pipeline is implemented in Suzaku. The programmer's interface is purposely similar to other patterns. The master executes the diffuse and gather routines and slaves execute the compute routine. Task data is held in a one-dimensional array as in the workpool version 1 as this is the most likely data structure and most efficient although there is no technical reason why the version 2 put and get mechanism could not be incorporated. The Suzaku pipeline will terminate naturally

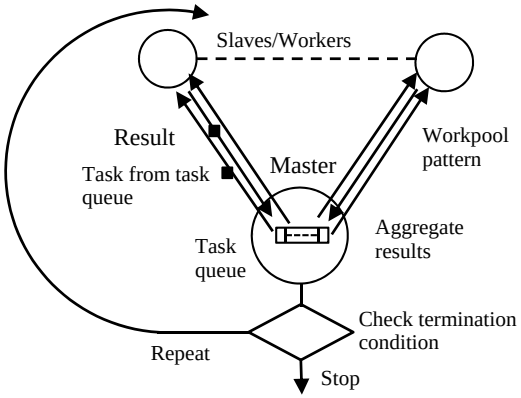

Figure 8. Iterative synchronous workpool pattern.

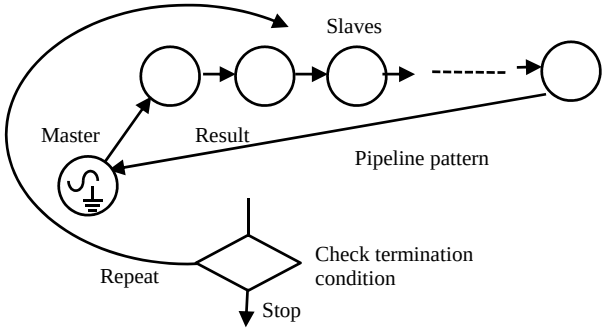

Figure 9. Iterative synchronous pipeline pattern.

after $T \times (P - 1)$ steps when there are T tasks and P processes. A routine, SZ_Terminate(), is provided to be able to terminate the pattern earlier when a specific termination condition exists. This routine would be called by the gather routine.

Fig. 10 shows pipeline code to sort numbers using the pipeline version of insertion sort. Numbers to sort are fed into the beginning of the pipeline. Each slave keeps the largest number it receives passing on smaller numbers. (In

```
#define N 1   // Size of data being sent
#define P 4   // Number of processes and number of numbers
void init(int *T, int *D) {
   *T = 4;            // number of tasks
   *D = 1;            // number of doubles in each size of task
   srand(999);
   return;
}
void diffuse (int taskID, double output[N]) {
   if (taskID < P) output[0] = rand()% 100;
     else output[0] = 999;          // otherwise terminator
return;
}
void compute(int taskID, double input[N], double output[N]){
   static double largest = 0;
   if (input[0] > largest) {
      output[0] = largest;     // copy current largest into send array
      largest = input[0];      // replace largest with rec. number
   } else {
      output[0] = input[0];      // copy recv into send array
   }
return;
}
void gather(int taskID, double input[N]) {
   if (input[0] == 999) SZ_terminate();
return;
}
int main(int argc, char *argv[]) {
   int p;
   SZ_Init(p);
   …
   SZ_Parallel_begin
      SZ_Debug();
      SZ_Pipeline(init,diffuse,compute,gather);
   SZ_Parallel_end;
   …
   SZ_Finalize();
   return 0;
}
```

Figure 10. Insertion sort using the Suzaku pipeline.

class, we also discuss partitioning the problem so that groups of numbers are sent to adjacent slaves.) SZ_Debug() is optional and causes debug messages to be displayed during execution and is placed before start of the pattern. With pre-implemented patterns it is really important to be able to watch the execution steps as the programmer does not have access to the underlying implementation. SZ_Debug() sets a flag inside Suzaku and can be applied to other patterns.

VII. Generalized Patterns

Message-passing patterns connect sources and destinations together in various ways. For a particular application, a specific interconnection pattern might offer advantages. The pipeline pattern is one such specialized pattern. Other patterns include the stencil pattern in which slaves are arranged in a two or three dimensional mesh and each slave connects to its neighbors in the mesh. The stencil pattern can be generalized into what we call *overlapping connectivity patterns* in which each slave (in this context) connects to a group of other slaves in the proximity and the groups overlap [15]. The extreme case is when each slave connects to all the other slaves in an all-to-all pattern. The other extreme is when each slave connects to only one other slave, such as the pipeline pattern, which connects each slave to the next slave in a linear fashion. There are many other possible connection patterns for example binary tree, hypercube, and arbitrary connection patterns for specific problems. Rather than implement every pattern in a unique way, the approach taken in Suzaku is to implement a pattern based upon a directed graph called here a *connection graph*. Any connection pattern can be created this way. We call the approach a *generalized pattern*. Of course, one has to avoid messaging deadlock in the pattern implementation. We use MPI message passing with explicit buffers. It may be the implementation is not as efficient as a specific implementation for a specific pattern. The motive of the Suzaku development is not for ultimate code execution speed and efficiency but as a tool that can be used with students to teach good programming practices, possibly at lower levels of the curriculum. If one wants the best possible execution speed, one could first choose the pattern and get the program to work with the Suzaku framework, and then re-engineer it in MPI. This is one advantage of Suzaku. There is a close relationship to MPI.

The Suzaku generalized pattern is an iterative synchronous pattern with a master-slave structure as illustrated in Fig. 11. The master sends initial data to all slaves at the beginning of the computation and collects results from all slaves at the end of the computation. The slaves compute and send values to those slaves that are interconnected, repeatedly until the termination condition exists. The master also acts as one slave as in the master-slave pattern. For greatest flexibility, the programmer implements the iteration loop and low-level Suzaku routines are used for broadcast, scatter, and gather in Fig. 11.

The overall program structure is shown in Fig. 12. SZ_Pattern_init() initializes the connection graph for a standard pattern (all-to-all, pipeline, stencil). The routine compute() in Fig. 12 is executed by each slave. The actual

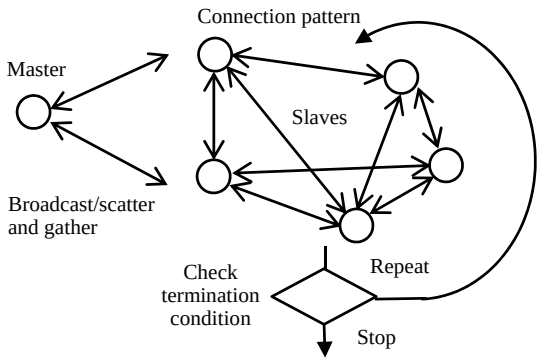

Figure 11. Generalized Suzaku pattern.

```
SZ_Parallel_begin                  // parallel section, all processes do this
   SZ_Pattern_init("pattern_name",size_of_data); // set up pattern
   SZ_Broadcast(input);            // broadcast initial data to all slaves
   for (s = 0; s < steps; s++) {   // in this case a fixed # iterations
      compute(s,input,output);     // master and slaves execute compute
      SZ_Pattern_send(output,input); // sent results  to connected procs
    }
    SZ_Gather(input,result);       // collect results from slaves
SZ_Parallel_end;                   // end of parallel
```

Figure 12. Suzaku program structure for generalized pattern.

computation depends upon the application. SZ_Pattern_send() sends data from each slave to all connected slaves according to the connection graph. The source data being transferred is a one-dimensional array of doubles, output[N] where N is the number of doubles in a task. The destination array is a two-dimensional array, input[P][N] where there are P processes. The graph entry at connection_graph[i][j] indicates whether is a connection between process i and process j (–1 for no connection) and if so the row in the destination array where the data is to be placed, i.e. if connection_graph[i][j] = x, the location is input[x][N]. This allows slaves to receive data from every other slave and place the data in different rows. The 3 × 3 stencil pattern has the connection graph shown in Fig. 13. Processes are numbered in natural order (row major order). Apart from slaves at the edges, each slave connects to the four neighbors to the left, right, up, and down.

		Destination								
		0	1	2	3	4	5	6	7	8
Source	0	-1	0	-1	2	-1	-1	-1	-1	-1
	1	1	-1	0	-1	2	-1	-1	-1	-1
	2	-1	1	-1	-1	-1	2	-1	-1	-1
	3	3	-1	-1	-1	0	-1	2	-1	-1
	4	-1	3	-1	1	-1	0	-1	2	-1
	5	-1	-1	3	-1	1	-1	-1	-1	2
	6	-1	-1	-1	3	-1	-1	-1	0	-1
	7	-1	-1	-1	-1	3	-1	1	-1	0
	8	-1	-1	-1	-1	-1	3	-1	1	-1

Figure 13. Connection graph for the stencil pattern.

A sample program segment is given in Fig. 14 for solving the two-dimensional heat distribution problem (static heat equation). For simplicity here, only 16 points are shown and one process for each point. In a more realistic situation each slave would handle a block of points. The best partitioning can be explored in the class. Note as with all Suzaku code, the code is *much* simpler than if it had been written in MPI, but it creates MPI code.

Two additional generalized pattern routines are available to the programmer, SZ_Set_conn_graph(int *graph) and SZ_Print_conn_graph(). SZ_Set_conn_graph() enables the programmer to create any personalized pattern in the connection graph by supplying a two-dimensional array with the connections specified. SZ_Set_conn_graph() replaces SZ_Pattern_init(). SZ_Print_conn_graph() prints out the current connection graph.

VIII. Classroom Experiences

Suzaku creates MPI message-passing programs but hides all the complexities of MPI. As such, one can start with patterns and Suzaku and study MPI later (or not even learn MPI at all) in a top-down approach or consider MPI first and patterns and Suzaku afterwards in a bottom-up approach. In our senior parallel programming class at UNC-Charlotte, we start with OpenMP, then MPI, and then patterns with Suzaku. This appears to be the best way for a full course on parallel programming. It is an open question whether to use a bottom-up approach or a top-down approach for a course in which pattern parallel programming is introduced as a small part of a lower-level programming course. Apart from instilling good software engineering principles, a top-down approach using Suzaku first has the advantage that the lower-level OpenMP and MPI do not need to be covered at all. That can be left to a later course.

Suzaku was introduced in the parallel programming class at UNC-Charlotte in Spring 2015 and Fall 2015, and we will report on students' responses for these two offerings. Both were taught as online courses. Students were provided with a VirtualBox virtual machine with all the software pre-installed for their own computer, as well as access to a departmental cluster for final testing. In Spring 2015, there were 65 students registered in the class (35 undergraduate students and 30 graduate students). In Fall 2015, there were 62 students registered in the class (27 undergraduate students and 35 graduate students). In both cases, there were seven 2-week programming assignments. For each assignment, students submit a report with their code and results. The Suzaku assignment was the fifth assignment after two OpenMP assignments and two MPI assignments.

```
SZ_Parallel_begin                // parallel section, all processes do this

   SZ_Pattern_init("stencil",1);  // set up slave interconnections
   SZ_Broadcast(pts);             // Set up initial values in each process
   …                              // copy initial values into B[][]
   for (t = 0; t < T; t++) {      // compute values over time T
      A[0] = 0.25 * (B[0][0]+B[1][0]+B[2][0]+B[3][0]);// computation
      SZ_Generalized_send(A,B); // sent compute results in A to B
   }
   SZ_Gather(A, temp);            // collect results into temp

SZ_Parallel_end;                 // end of parallel
```

Figure 14. Program segment for the two-dimensional heat distribution problem.

In Spring 2015, the Suzaku assignment was divided into four parts. Part 1 was a tutorial where sample Suzaku code was given. In Part 2, students had to write a sequential program to solve the astronomical *N*-body problem given skeleton code. Part 3 asked students to convert the sequential program into a Suzaku program having a master-slave pattern. Part 4 was on MPI/OpenMP hybrid programs. Students were asked to convert an MPI/OpenMP hybrid program into a Suzaku/OpenMP hybrid program, and graduate students were also asked to convert the *N*-Body program into a hybrid Suzaku/OpenMP program.

After completing Part 3, students were asked to give a brief evaluation of using the low-level Suzaku routines, comparing and contrasting using Suzaku with MPI. They were also asked to describe their experiences and opinions, and give any suggestions for improvement. There were 28 responses, all but one highly positive (96.4% positive) with quite extensive evaluations. The most common comments were along the lines of "easier to use", "user friendly", "less time to write code", "more concise." The disadvantage most notably mentioned was that the message data type had to be a double. (We had yet to develop the typeless feature of Suzaku in Spring 2015.) Students were also asked to make conclusions on using Suzaku in Part 4 after using Suzaku for the hybrid program. There were 28 responses, again highly positive with one negative response (96.4% positive). Comments included "using Suzaku was a lot of fun." It is good to know we can make hybrid parallel programing fun.

Assignment 5 in Fall 2015 was similar except we dropped the hybrid programming and added the Suzaku workpool, which had been fully developed by then. Part 4 in Assignment 5 asked graduate students to reformulate the *N*-body program as a Suzaku workpool. This required developing the code for the diffuse(), compute(), and gather() routines. Again, students were asked after Part 1 (Suzaku tutorial) to give a brief discussion on using Suzaku compared to MPI, describing the advantages and disadvantages. There were 47 responses, typically saying "much easier to use with simple commands as compared to MPI", "Suzaku is more user friendly." Disadvantages pointed to included not having as much control of the code as in MPI.

Students were asked after Part 3 (parallelizing the *N*-body program) to give an evaluation of using the Suzaku comparing and contrasting with MPI, and describe their experience and opinions, and any suggestions for improvement. There were 42 responses, all but three responses highly positive. Comments again focused on the ease of use. The three negative responses (7%) pointed to lack of documentation and lack of understanding what the Suzaku routines did. These students had problems getting their code to execute. There was documentation provided but more may be needed especially in an on-line setting. However, overwhelmingly, students appreciated the ease that parallel programs could be constructed with established patterns.

IX. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing

The report "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates" describes the topics in parallel and distributed computing that are expected to be covered in BS degrees in Computer Science or Computer Engineering. [16] The report promotes spreading basic concepts into core CS courses (CS1, CS2, Systems, and Data Structure and Algorithms). Our pattern-based approach is not specifically described in the report although one can find a sprinkling of patterns (divide and conquer, recursion, scan, reduction, stencil, etc) mostly proposed for Data Structure and Algorithms. The report also accepts that it is sometimes necessary to place all the materials in one upper-level parallel programming elective as we currently teach parallel programming. We support the notation of distributing key concepts throughout lower-level CS core courses and would advocate introducing parallel programming in the lower levels using our design pattern approach. Our low-level patterns correspond to the communication patterns listed in the report for Data Structure and Algorithms. Suzaku communication patterns can be used by students without learning MPI. Some of our higher-level patterns also appear for Data Structure and Algorithms. But some are missing, including the workpool. The Suzaku workpool can reasonably be done in lower-level courses because it does not require students to implement the workpool code.

X. Conclusions

In this paper, we introduced a new pattern programming tool called Suzaku that makes it much easier to create MPI message-passing code based upon established parallel design patterns and to teach good parallel programming design. Suzaku provides low-level routines that map directly to MPI routines but avoiding the long list of parameters needed in MPI routines (including MPI communicators, tags, and specifying the data type and the data size), yet it leads to a path to or from MPI programming. The feature of Suzaku not needing to specify the type or size of the data in certain routines (with limitations), makes it effectively typeless and offers ease of programming. Suzaku also provides routines for higher-level patterns including the static workpool, dynamic workpool, pipeline, and through a generalized pattern feature many patterns including user-defined patterns. We chose to use a bottom-up approach in teaching, starting with OpenMP and MPI and introducing Suzaku afterwards in this study. Most students find pattern programming easier to learn after learning MPI, whereas from a software engineering perspective starting with higher-level abstraction of patterns should be more desirable. Pattern programming tools certainly constrain the programmer and some students will definitely not like that. However it is our experience that students appreciate the power of higher-level tools. Student comments about using this framework are very positive. We strongly believe that ad-hoc parallel programming should be avoided. Students should be trained in methods that are conducive to good programming and lend themselves to program maintenance and large collaborative applications.

Acknowledgment

This material is based upon work supported by the National Science Foundation under the collaborative grant #1141005/1141006. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] E. Gamma, R. Helm., R. Johnson, and V. Vlissides, Design Patterns, New York: Addison-Wesley, 1995.

[2] O. Astrachan, "Design Patterns: An Essential Component of CS Curricula," SIGCSE Bulletin and Proceedings. Vol. 30, no. 1, 1998, pp. 153-160.

[3] "Design Patterns," Wikipedia. https://en.wikipedia.org/wiki/Design_Patterns.

[4] R. Nystrom, Game Programming Patterns, Genever Benning Publishers, 2014.

[5] S. Toub, "Patterns of Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4," 2009. http://www.microsoft.com/downloads/details.aspx?FamilyID=86b3d32b-ad26-4bb8-a3ae-c1637026c3ee&displaylang=en

[6] "Computer Science Curriculum 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science," The Joint Task Force on Computing Curriculum ACM/IEEE Computer Society, Dec. 2013. http://www.acm.org/education/CS2013-final-report.pdf

[7] Fastflow. University of Torino, Italy /Università di Pisa. http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

[8] K. Keutzer and T. Mattson, Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software. http://parlab.eecs.berkeley.edu/wiki/_media/patterns/paraplop_g1_1.pdf

[9] T. G. Mattson, B. A. Sanders, and B. L. Massingill, Patterns for Parallel Programming, Addison Wesley, 2004.

[10] M. McCool, J. Reinders, and A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Waltham MA: Morgan Kaufmann, 2012.

[11] J. Villalobos, Running Parallel Applications on a Heterogeneous Environment with Accessible Development Practices and Automatic Scalability, PhD diss. University of North Carolina Charlotte, 2011.

[12] B. Wilkinson, J. Villalobos, and C. Ferner, "Pattern Programming Approach for Teaching Parallel and Distributed Computing," SIGCSE 2013 Technical Symposium on Computer Science Education. Denver, Colorado, 2013.

[13] B. Wilkinson and M. Allen, Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers 2nd ed., Upper Saddle River, New Jersey: Prentice Hall, 2005, p. 214.

[14] J. F. Villalobos and B. Wilkinson, "Skeleton/Pattern Programming with an Adder Operator for Grid and Cloud Platforms," The 2010 International Conference on Grid Computing and Applications (GCA'10), Las Vegas, Nevada, USA, July 12-15, 2010.

[15] B. Wilkinson, "Overlapping Connectivity Interconnection Networks for Shared Memory Multiprocessor Systems," Journal of Parallel and Distributed Computing, vol. 15, no. 1, May 1992, pp. 49–61.

[16] Sushil K Prasad, et al., "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates," Version I, December 2012. http://www.cs.gsu.edu/~tcpp/curriculum/index.php