Teaching Parallel Programming Using Java

Aamir Shafi, Aleem Akhtar, Ansar Javed
School of Electrical Engineering & Computer Science (SEECS), National University of Sciences and Technology (NUST), Pakistan
{*aamir.shafi, aleem.akhtar, ansar.javed*}*@seecs.nust.edu.pk*

Bryan Carpenter
School of Computing, University of Portsmouth, UK
bryan.carpenter@port.ac.uk

***Abstract*—This paper presents an overview of the "Applied Parallel Computing" course taught to final year Software Engineering undergraduate students in Spring 2014 at NUST, Pakistan. The main objective of the course was to introduce practical parallel programming tools and techniques for shared and distributed memory concurrent systems. A unique aspect of the course was that Java was used as the principle programming language. The course was divided into three sections. The first section covered parallel programming techniques for shared memory systems including multicore and Symmetric Multi-Processor (SMP) systems. In this section, Java threads API was taught as a viable programming model for such systems. The second section was dedicated to parallel programming tools meant for distributed memory systems including clusters and network of computers. We used MPJ Express—a Java MPI library—for conducting programming assignments and lab work for this section. The third and the final section introduced advanced topics including the MapReduce programming model using Hadoop and the General Purpose Computing on Graphics Processing Units (GPGPU).**

***Keywords*-Parallel Programming Education; MPJ Express; Java MPI;**

I. Introduction

The emergence of multicore hardware has brought parallel computing into the limelight. It has also put the burden of improving performance of applications on the software programmers [1]. The only option to increase performance of existing sequential applications is to utilize some form of parallelism. This implies that the software development community—including current and future software engineers—must learn parallel programming models to write optimized code for multicore processors and High Performance Computing (HPC) hardware.

Realizing the importance of teaching concurrency at the undergraduate level, a 2 + 1 credit hours elective course titled "Applied Parallel Computing" was added to the Bachelors of Software Engineering program at NUST, Pakistan. The program spans four years—distributed in eight semesters—and 136 credit hours. This particular course on parallel computing was taught in the eighth and the last semester. Course contents were mostly adapted from a Parallel and Distributed Computing (PDC) course taught at the University of Portsmouth, UK.

The course began with an introduction of parallel computing, which motivated the need for such computing to solve some of the biggest possible problems in the least possible time. Some important concepts including shared/distributed memory systems, performance measurement metrics, and hardware accelerators were introduced. After the initial introduction, the course was divided into three sections. The first section covered programming techniques for shared memory systems including multicore processors and Symmetric Multi-Processors (SMPs). These included Java threads [2], OpenMP [3], and Intel Cilk Plus [4]. Note that all practical work including assignments, labs, and code samples during lectures were Java-based. The second section covered programming tools and APIs for distributed memory systems including commodity clusters. For this section, the course focused on writing parallel applications using a Java MPI-like software called MPJ Express [5], which implements the mpiJava 1.2 API specification [6]—this is equivalent to MPI version 1.1. Being a Java MPI library, MPJ Express allows writing parallel Java applications for clusters and network of computers. To allow easy configuration and installation for students, MPJ Express provides a *multicore mode*, where Java threads are used to simulate parallel MPI processes in a single JVM. Our students found this to be an extremely useful feature since initially they were able to write, execute, and test parallel Java code on their personal laptops/PCs. Once stable, the same code would also execute on a cluster or a network of computers using the MPJ Express *cluster mode*. The third and the final section gave introduction to advanced topics including the MapReduce programming model using Hadoop and the General Purpose Computing on Graphics Processing Units (GPGPU).

A. Motivation for using Java

An interesting and unique aspect of this parallel computing course was preferring Java over traditional

978-1-4799-7021-6/14 $31.00 © 2014 IEEE
DOI 10.1109/EduHPC.2014.7

native HPC languages like C and Fortran for the practical part of the course. The advantages of the Java programming language include higher-level programming concepts, improved compile time and runtime checking, and, as a result, faster problem detection and debugging. In addition, Java's automatic garbage collection, when exploited carefully, relieves the programmer of many of the pitfalls of lower-level languages. The Java Development Kit (JDK) includes a large set of libraries that can be reused by developers for rapid application development. Another, interesting, argument in favour of Java is the large pool of developers—the main reason is that Java is taught as one of the major languages in many Universities around the globe. A highly attractive feature of applications written in Java is that they are portable to any hardware or operating system, provided that there is a Java Virtual Machine (JVM) for that system.

In order to facilitate writing parallel Java code for shared memory systems, Java is equipped with a feature-rich threading API. In order to teach programming distributed memory systems, we had the option to choose between various Java MPI libraries including MPJ Express [5], FastMPJ [7], and Open MPI Java [8]. In this context, we choose MPJ Express that is being developed and maintained at NUST, Pakistan. For the third and the final section of the course we choose the Apache Hadoop software [9] for implementing MapReduce applications, which is a popular open-source software to process—through automatic parallelization—large amounts of data.

Rest of this paper is organized as follows. Section II introduces HPC workloads used as sample applications throughout the course. Section III outlines the course syllabus followed by a detailed discussion on course contents. Section IV concludes the paper.

II. HPC Workloads

This section introduces HPC workloads used as sample applications throughout the course. We first introduce sequential version of these applications. Students were invited to develop shared and distributed memory versions using threads and messaging in the first and second sections of the course.

Typically parallel computations can be roughly divided into two categories based on their requirements to communicate during the computation phase. Applications that do not require any communication in computation phase are called *embarrassingly parallel* computations, while others that require periodic communication in computation phase are generally referred to as *synchronous* computations. We choose three embarrassingly parallel computations, which included π Calculation, Mandelbrot Set Calculation, and Matrix-Matrix Multiplication. Also, we choose two synchronous computations that included Conway's Game of Life and Laplace Equation Solver. Rest of the section presents an overview of each of the sample application.

A. π Calculation

Calculation of mathematical constant π is an embarrassingly parallel application and it provides good starting point in learning parallel programming techniques. There are many ways to approximately calculate the mathematical constant π. One "brute force" method is based on the following formula:

$$\pi \approx h \sum_{i=0}^{N-1} \frac{4}{1 + (h(i + \frac{1}{2}))^2}$$

where N is the number of steps and h is the size of a single step. We set N to 10 million, which is sufficiently large to get an accurate estimate of π. The above formula for calculating π is simply a large sum of independent terms. Listing 1 shows the sequential version of the π calculation code.

```
1 for(int i=0; i<numSteps; i++) {
2   double x=(i+0.5) * step;
3   sum += 4.0/(1.0 + x*x);
4 }
5 double pi=step * sum ;
```

Listing 1. Serial π Calculation Code

B. The Mandelbrot Set

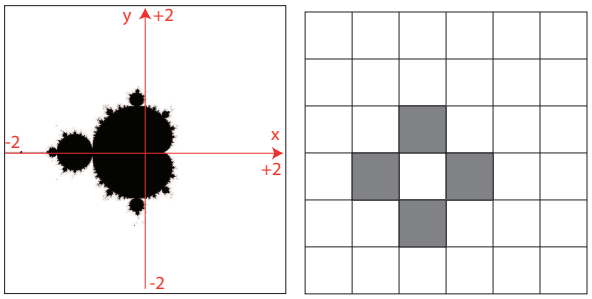
The Mandelbrot Set is a collection of complex numbers that are quasi-stable—values increase and decrease but do not exceed a particular limit. The set is computed by iterating the function:

$$z_{k+1} = z_k^2 + c$$

Iterations continue until magnitude of z is greater than 2, or the number of iterations reaches arbitrary limit. The Mandelbrot Set can be seen in Figure 1(a) where mathematically the set is the black area within part of x, y plane with $-2 \leq x, y \leq 2$. Listing 2 depicts the serial pseudo code for the Mandelbrot Set. As shown, the innermost while loop repeats forever if we are in the black region; in practice, stop the loop after some `CUTOFF` number of iterations.

```
1 for(int i=0; i<N; i++)
2   for(int j=0; j<N; j++) {
3     double x=step * i - 2.0; //  -2<=x<=2
4     double y=step * j - 2.0; //  -2<=y<=2
5     complex c=(x, y), z=c;
6     int k = 0;
7     while (k<CUTOFF && abs(z)<2.0) {
8       z=c + z * z;
9       k++ ;
10    }
11    set[i][j]=k;
12  }
```

Listing 2. Serial Mandelbrot Set Calculation Code



(a) Mandelbrot Set (b) Conway's Game of life

Figure 1. HPC Workloads

C. Matrix Multiplication

Another embarrassingly parallel computation discussed in the course included the Matrix-Matrix multiplication. It is a useful scientific kernel where parallelism not only helps in improving code performance but also allows solving larger matrices on parallel hardware. Elements of the resultant matrix C are produced by multiplying matrices A and B as follows:

$$C_{i,j} = \sum_{k=0}^{l-1} A_{i,k} B_{k,j}$$

Listing 3 shows pseudo code for multiplying two square matrices A and B.

```
1 for (i=0; i<n; i++) // for A's rows
2   for (j=0; j<n; j++) {// for B's columns
3     c[i][j] = 0;
4     for (k = 0; k < n; k++)
5       c[i][j]=c[i][j]+a[i][k]*b[k][j];
6   }
```

Listing 3. Serial Matrix Multiplication Code

D. Conway's Game of Life

Conway's Game of Life is a cellular automaton on a $2D$ grid as shown in Figure 1(b), where each cell takes the value 0 (dead) or 1 (alive). As part of the simulation, newer generations of cells are evolved according to a pre-defined criteria. At each timestep, the new value of each cell depends on its old value and old values of the neighbouring cells. Listing 4 shows the pseudo code for the sequential version of the Game of Life. The `cells` array is the main data-structure while the auxiliary array `sums` holds the sum of cell elements neighbouring the cell (i, j) after the sum phase. As the `while` loop executes, cell values—stored in the `cells` array—evolve from one generation to another.

```
1 while(true) {
2   // Sum Phase
3   for(int i=0; i<N; i++)
4     for(int j=0; j<N; j++)
5       sums[i][j]=sum of cells values neighbouring (i,
            j);
6   // Update Phase
7   for(int i=0; i<N; i++)
8     for(int j=0; j<N; j++)
9       cells[i][j]=update(cells[i][j], sums[i][j]);
10 }
```

Listing 4. Serial Conway's Game of Life Code

E. Laplace Equation Solver

The two-dimensional Laplace equation is an equation that crops up in several places in physics and mathematics. We choose Laplace equation as a sample application in this course because it is a relatively simple numerical problem—in science and engineering—that can be tackled by parallel programming. The *discrete* version of the Laplace equation on a two-dimensional grid of points can be stated as:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Listing 5 shows the sequential code for solving Laplace equation. Here the main data-structure is the `phi` array, which stores unknown variables of the equation as its elements. An *iterative* numerical approach to solving the equation is just to initially set all elements of `phi` that we have to solve for to some value like zero, then repeatedly change individual `phi [i][j]` elements to be the average of their neighbours. If we repeat this local update sufficiently many times, the `phi` elements converge to the global solution of the equations. This is called the *relaxation method*.

```
1 for(int iter=0; iter<NITER; iter++) {
2   // Calculate new phi
3   for(int i=1; i<(N-1); i++)
4     for(int j=1; j<(N-1); j++)
5       phi[i][j] = 0.25F * (phi[i][j-1] + phi[i][j+1] +
              phi[i-1][j] + phi[i+1][j]);
6 }
```

Listing 5. Serial Laplace Equation Solver

III. Course Contents

This section discusses details of course contents taught in this course. We begin by presenting an overview of the course syllabus. This is followed by covering shared and distributed memory parallel programming techniques. The section concludes with a discussion on advanced topics, which were part of this course.

A. Course Syllabus

The weekly distribution of lectures, labs, and assignments is shown in Table I. The duration of the course was eighteen weeks. There are two One Hour Tests (OHTs) in week six and twelve. In addition, there is one End Semester Exam (ESE) that took place in the

last week, that is the eighteenth week. This was a 2 + 1 credit hours course, which implies two one hour weekly lectures and one lab. There are three contact hours for the weekly lab. In each lab the students were provided with a lab script, which invited them to develop parallel code for a sample HPC workload. The students were later graded on the basis of design/implementation of their parallel code. The shared memory part of the course was covered in the first four weeks followed by the first OHT in week six. Similarly, the distributed memory part of the course was covered from week five to eleven followed by the second OHT in week twelve. The last week teaching weeks—from thirteen to seventeen—covered advanced topics followed by the ESE in week eighteen.

Table I
Parallel Programming Course Syllabus

Week	Topic
Week-1	Introduction to Parallel Computing Review of Java Threads Lab 1: π Calculation using threads
Week-2	Introduction to Parallel Hardware Parallel Programming Approaches Lab 2: Array Operations using threads
Week-3	Embarrassingly Parallel Computations Shared Memory Programming Lab 3: Mandelbrot Set Calculation using threads Assignment 1: Monte Carlo π calculation
Week-4	Introduction to OpenMP Introduction to Cilk Lab 4: Parallelizing Game Of Life using threads
Week-5	Distributed Memory Systems and MPI Data Decomposition and MPI Communication Lab 5: Solving Laplace Equation using threads Assignment 2: Dense matrix multiplication
One Hour Test-1	
Week-7	Features of MPI Lab 6: π Calculation using MPI
Week-8	MPJ Express Programming Lab 7: Mandelbrot Set calculation using MPI
Week-9	Global and Local Synchronization Lab 8: Solving Game of life using MPI Assignment 3: Monte Carlo π calculation using MPI
Week-10	MPI Point to Point Communication Lab 9: Array Operations using MPI
Week-11	MPI Collective Communication Lab 10: Solving Laplace Equation using MPI
One Hour Test-2	
Week-13	GPU Programming - I Lab 11: Mandelbrot Set using MPI Collectives
Week-14	GPU Programming -II Lab 12: N-body Simulations
Week-15	Motivation of MapReduce Lab 13: Array operations using GPUs
Week-16	Apache Hadoop Lab 14: Word count using MapReduce
Week-17	Course Review
End Semester Exam	

Table II shows our grading policy clearly depicting the weightage assigned to theoretical and practical parts of the course. The practical part of the course was evaluated through weekly labs and a final lab exam. The lab exam was programming-based and required students to parallelize a serial code. Weekly labs were conducting in a typical teaching lab, which included forty PCs connected via Gigabit Ethernet to one another. Each PC comprised of Intel® Core™ i5-3470 CPU and 4 GBytes of main memory.

Table II
Grading

Theoretical (70%)		Practical (30%)	
One Hour Tests	35%	Weekly labs	80%
End Semester Exam	45%		
Quizzes	15%	Lab Exam	20%
Assignments	5%		

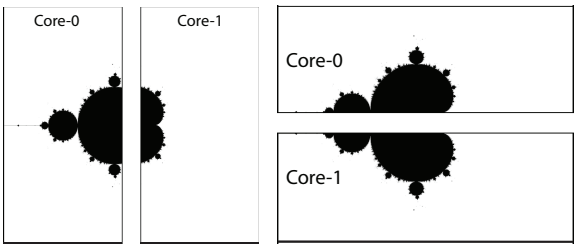
B. Shared Memory Parallel Programming

This sub-section outlines shared memory parallel programming techniques covered in this course. In this context, the course reviewed the Java threads API as a viable option for writing shared memory concurrent programs. Lectures covered as part of this section demonstrated using threads for implementing embarrassingly parallel and synchronous computations introduced earlier in Section II. While covering these HPC workloads, the instructor illustrated important parallelism concepts including problem decomposition/partitioning, load balancing, and synchronization. Two main partitioning techniques namely block-wise and cyclic distributions were covered— see Listing 6 for code patterns for the two distributions.

```
1 // Original for loop
2 for (int i=0; i<N; i++)}
3 // me=current thread; P=total threads
4 // block-wise distribution of each thread
5 for (int i=me*N/P; i<(me+1)*N/P; i++)
6 // cyclic distribution of each thread
7 for (int i=me; i<N; i+=P)
```

Listing 6. Decomposition using Block/Cyclic Distributions

The Mandelbrot Set: As part of our coverage on parallelizing embarrassingly parallel computations using Java threads on shared memory platforms, there was a discussion on multicore-enabling the Mandelbrot Set code. This dialog also demonstrated key topics including partitioning and load balancing. Our initial attempts to develop a multi-threaded Mandelbrot Set calculation code were based on dividing the (x, y) plane into two halves both vertically and horizontally. Poor load balancing was observed in the vertical division as shown in Figure 2(a). The reason is that core 0 had more substantially work than core 1. By dividing the (x, y) plane horizontally into two halves, we noted perfect load balancing due to symmetry of the Mandelbrot Set—this can be seen in Figure 2(b). However, if the



(a) Vertically (b) Horizontally

Figure 2. Partitioning the (x, y) plane

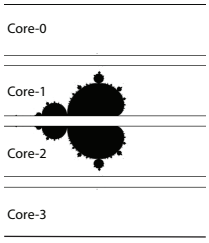


Figure 3. Partitioning the (x, y) plane in four horizontal blocks

horizontal partitioning is carried out on four cores, then we also observed poor load balancing as core 0 and 3 have little fraction of the total computational work—this is depicted in Figure 3. It was discussed during lectures that this particular issue can be tackled by using cyclic distribution, which is difficult to implement and sometimes less efficient due to poor usage of cache but also has merits in certain applications like the Mandelbrot Set calculation.

Matrix Multiplication: Parallelizing Matrix-Matrix computations with Java threads was discussed next. For dense matrices, it is possible to achieve good speedups by exploiting traditional vertical block-wise partitioning. However, we noted that this traditional partitioning strategy does not work well for sparse matrices. In addition, sparse matrices unnecessarily waste memory if stored in two-dimensional array format due to large numbers of zeroes. This is solved by utilizing a special data-structure that keeps track of row, column, and value of each non-zero element in the matrix. This information can be stored as *triples* in the form of an array list or a linked list. Parallelization can now be achieved by partitioning the list of triples instead of the original sparse matrix.

Conway's Game of life: As part of our coverage on parallelizing synchronous computations using Java threads, we started off with Conway's Game of Life. The discussion began with a review of the sequential code—shown in Listing 4—that includes two computational phases called sum and update. In the sum phase, the code calculates sums of all neighbours of `cells[i][j]`—this sum is stored in `sums[i][j]`. In the update phase, the new value is written to `cells[i][j]`. In our first attempt, we developed a threaded version that used conventional block-wise partitioning strategy to execute sum and update phases concurrently in multiple threads. However, this version introduced a race condition in the code that resulted in an unpredictable behavior. The reason was that some threads ran ahead of other threads perhaps by several generations due to lack of any synchronization/communication between concurrent threads. This issue is depicted in Figure 4 where core 0 is writing the border black cell (in the update phase) and while doing so, it is also reading all grey cells (in the sum phase). In general, all cells with vertical stripes are written by the *owner* thread and read by the left neighbour. Similarly all cells with horizontal stripes are written by the owner thread and read by the right neighbour thread. This issue was tackled by employing *barrier synchronization*. For this purpose, the parallel code instantiated an object of the `java.util.CyclicBarrier` class. This object was used to call the `await()` in co-operating threads to achieve barrier synchronization. This function call blocks until all P threads—taking part in the computation—have made this call. Parallelized version of Conway's Game of Life with barrier synchronization can be seen in Listing 7.

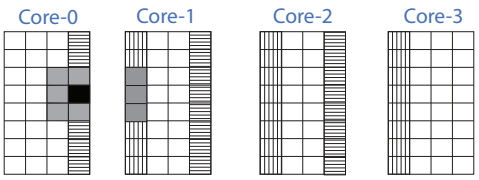


Figure 4. Partitioning in Multi-threaded Conway's Game of Life

```
1 Class LifeThread {
2   void run() {
3     while(true) {
4       // Sum Phase
5       for(int i=begin; i<end; i++)
6         for(int j=0; j<N; j++)
7           sums[i][j]=sum of cells values neighbouring(
                i, j);
8       barrier.await() ;
9       // Update Phase
10      for(int i=begin; i<end; i++)
11        for(int j=0; j<N; j++)
12          cells[i][j]=update(cells[i][j], sums[i][j]);
13      barrier.await() ;
14    }
15  }
16 }
```

Listing 7. Multi-threaded Conway's Game of Life Code

C. Distributed Memory Parallel Programming

This sub-section outlines distributed memory parallel programming techniques covered in this course. In this context we began by reviewing MPJ Express—a Java MPI library. Most parallel programs designed

to run on large clusters utilize MPI for messaging. We noted that MPI implements the Single Program Multiple Data (SPMD) model—each process runs the same program, but of course operates on its own local memory (data). Lectures covered as part of this section demonstrated using messaging—as provided by MPJ Express—for implementing embarrassingly parallel and synchronous computations introduced earlier in Section II. While covering these HPC workloads, we reviewed and practiced using point-to-point and collective communication routines provided by the MPJ Express software.

Listing 8 shows the most basic MPJ Express program. The MPJ Express library is initialized and finalized using the `MPI.Init(args)` and `MPI.Finalize()`, respectively. Once initialized, the MPI library provides access to a special data-structure called *communicator*, which encapsulates all processes taking part in the parallel execution. In our code this data-structure is represented by the `MPI.COMM_WORLD` object, which can be used for calling various routines including querying total number of parallel processes—via `MPI.COMM_WORLD.Size()`—and a process' own *rank*—via `MPI.COMM_WORLD.Rank()`. The `MPI.COMM_WORLD` object can also be used for invoking communication routines including point-to-point and collective communication. Listing 9 shows signatures for the most basic blocking send and receive primitives provided by MPJ Express.

```
1 import mpi.*;
2 public class HelloWorld {
3   public static void main(String args[]) throws
        Exception {
4     MPI.Init(args) ;
5     // Get total number of processes
6     int P = MPI.COMM_WORLD.Size();
7     // Get rank of each process
8     int me = MPI.COMM_WORLD.Rank();
9     MPI.Finalize() ;
10  }
11 }
```

Listing 8. MPJ Express Hello World Code

```
1 void Comm.Send(Object buf, int offset, int count,
      Datatype type, int dest, int tag)
2 Status Comm.Recv(Object buf, int offset, int count,
      Datatype type, int src, int tag)
```

Listing 9. MPJ Express Send and Receive Methods

π Calculation: Listing 10 shows the sketch of the MPI version of the π calculation code. This code utilizes the primitive messaging functions like `Send()` and `Recv()`. Each MPI process independently calculates its own contribution to the `sum` variable, which is communicated to the master process (rank 0). Once the master process has received all contributions from *slave/worker* processes, it calculates the final sum that is used to generate the value of π.

```
1 if (rank != 0) {
2   double[] sendBuf=new double[]{sum};
3   //1-element array containing sum
4   MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0,
        10);
5 }
6 else { //rank == 0
7   double[] recvBuf=new double[1] ;
8   for (int src=1 ; src<P; src++) {
9     MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src
          , 10);
10    sum += recvBuf [0] ;
11  }
12 }
13 double pi = step * sum ;
```

Listing 10. π Calculation using MPJ Express

The Mandelbrot Set: Another embarrassingly parallel computation discussed in the class was the Mandelbrot Set. Here, students were invited to develop a distributed memory version of the Mandelbrot Set code using similar partitioning and communication patterns used in the π calculation code. Also, non-blocking (asynchronous) communication primitives that allow communication/computation overlap were introduced. Table III summarizes the blocking and non-blocking point-to-point communication routines provided by MPJ Express. Another utility function `Sendrecv()` was introduced, which essentially combines the `Send()` and `Recv()` functionality in a single call.

Table III
Point-To-Point Send Modes

Blocking	Non-blocking
`Send`	`Isend`
`Recv`	`Irecv`
`Bsend` (Buffered)	`Ibsend` (Buffered)
`Ssend` (Synchronous)	`Issend` (Synchronous)
`Rsend` (Ready)	`Irsend` (Ready)

Conway's Game of life: A distributed memory version of the Conway's Game of Life code was also discussed in the class. Once partitioning has been performed, each of the MPI process executes the sum and update phases in a concurrent fashion. However communication needs to take place during the sum phase for all the border cells belonging to the top and bottom row. Performing explicit communication for each cell during the sum phase is an expensive operation. This can be optimized by exchanging border rows amongst neighbouring processes before an MPI process enters the sum phase. To implement this, each MPI process introduces an additional row at the top and bottom of the grid—these are called *ghost* rows and are represented by grey horizontal stripped elements in Figure 5. These so-called ghost rows were exchanged using the `Sendrecv()` primitive and are supposed to contain vertically stripped rows of their left neighbour. For simplicity reasons, only one-sided exchange of

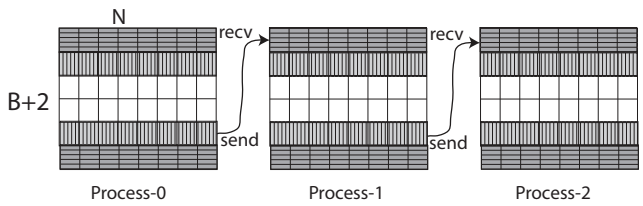


Figure 5. Partitioning in MPI Version of Conway's Game of Life

rows is shown in Figure 5. Listing 11 depicts a sketch of the parallel implementation of Conway's Game of Life code.

```
1  int cells[][]= new int[B+2][N];
2  int sums[][] = new int[B][N];
3  while(true) {
4    int next=(me + 1)%P;
5    int prev=(me - 1 + P)%P;
6    MPI.COMM_WORLD.Sendrecv(cells[B], 0, N, MPI.INT,
         next, 0, cells[0], 0, N, MPI.INT, prev, 0);
7    MPI.COMM_WORLD.Sendrecv(cells[1], 0, N, MPI.INT,
         prev, 0, cells[B+1], 0, N, MPI.INT, next, 0)
         ;
8    // Sum Phase
9    for(int i = 1 ; i < B+1 ; i++)
10     for(int j = 0 ; j < N ; j++)
11       sums[i][j]=sum of all neighbouring cells
12   // Update Phase
13   for(int i = 0 ; i < B ; i++)
14     for(int j = 0 ; j < N ; j++)
15       cells[i][j]=update(cells[i][j], sums[i][j]);
16 }
```

Listing 11. Conway's Game of Life using MPJ Express

Laplace Equation Solver: Another synchronous computation discussed in the context of distributed memory parallel programming was the Laplace Equation Solver. The partitioning and communication patterns exhibited by this application is similar to Conway's Game of Life. Ghost regions were introduced as an optimization and were communicated using `Sendrecv()` communication primitive before the actual computational loop executes.

Towards the end, the course also introduced collective communication primitives provided by MPJ Express—these are depicted in Table IV. Students were invited to re-write distributed memory parallel codes to make use of MPI collective communication instead of relying on point-to-point communication. Students observed that number of lines of code significantly reduced by exploiting specialized collective operations like `Bcast()`, `Reduce()`, `Gather()`, `Allreduce()`, and `Allgather()`. MPJ Express also provides a global synchronization primitive called `Barrier()`.

D. Advanced Topics in Parallel Programming

This sub-section outlines advanced topics including the MapReduce programming model using Hadoop and the General Purpose Computing on Graphics Processing Units (GPGPU).

Table IV
Collective Communication Primitives

Operation	Description
`bcast()` (Broadcast)	One process sends message to all other processes
`reduce()`	Reverse operation of `bcast()`
`Scatter()`	Distributes distinct messages from one process to all other processes
`Gather()`	Reverse operation of `Scatter()`
`Allgather()`	Combined operation of `Gather()` and `Scatter()`
`Barrier()`	Creates a barrier synchronization in a group

Apache Hadoop: As part of this section, the students were motivated with the need for MapReduce [10] programming model. The central idea behind this programming model is to process large amounts of data in a fault-tolerant manner on inexpensive hardware. This is especially attractive for small-to-medium enterprises and researchers who cannot afford expensive hardware typically used in HPC environments. This method of automatic parallelization is getting very popular in the industry. One obvious advantage of MapReduce over MPI is that the application developer is not responsible for explicit data placement and communication. In our course we reviewed the Apache Hadoop [9] software, which is an open-source implementation of the MapReduce programming paradigm in the Java language. We also discussed that the end-user is responsible for writing application code, which consists of the map and reduce functions. The map stage typically processes the input data and produces intermediate key-value pairs. These key-value pairs are then fed to reduce functions, which combine/filter/sort intermediate data to produce the final result. When compared with MPI, the application code is remarkably simpler.

Word Count Example: The usage of the MapReduce API was demonstrated by discussing the classic word count example originally presented in [10]. In this example, the application code is responsible for counting the frequency of unique words in a collection of documents. The map function in the example prepares a list of all words and sets the value to 1. This data comprising of intermediate key-value pairs is later fed to the reduce function. As part of reduce phase computation, repeated words in the intermediate data are summed up to produce the final result, which contains the frequency of each word present in the input data.

GPU Programming: This part of the course introduced GPUs as a massively-parallel parallel programming platform. Initial parts of lectures were dedicated to noting differences between CPUs and GPUs. On one hand CPUs are latency optimized but are built

with complex and power inefficient hardware. On the other hand, GPUs have simpler hardware and are bandwidth optimized. GPUs are typically a good choice for applications that involve minimum branching and have low communication to computation ratio. Various programming APIs including CUDA [11] and OpenCL were introduced to students. We also introduced JCUDA [12], which consists of Java bindings for CUDA. Key concepts of GPU programming were introduced to students through very simple and primitive examples. This part of the course was not extensively covered. In future offerings of the course, we plan to provide more coverage by discussing elaborate examples using the JCUDA library.

IV. Conclusions

This paper reviewed a parallel computing course taught as part of the Software Engineering program [1] at NUST, Pakistan. A unique feature of the course was that Java was used as the principle parallel programming language throughout the course.

This course was offered for the very first time as a final semester elective—a total of 45 students out of 77 opted to take this course. From the instructor's point of view, the students found the course very practical and useful. Most students appreciated the hands-on approach taken for the first two sections of the course namely the shared and distributed memory systems programming sections. Most of the practical work was done during weekly labs where students were initially provided with serial versions of various workloads. Later the lab script invited them to develop shared/distributed versions of the same code. Formal feedbacks were also collected twice by instructor's academic department. According to these, the instructor scored 86.9% and 89.8% percent marks, which is an indication of high student satisfaction.

In the future course offerings, we plan to improve the last section on Advanced Topics including GPGPU and Apache Hadoop by increasing lectures and weekly labs dedicated for these two topics. Lab scripts for this section used in this year's course were very basic and required students to only understand and execute concurrent codes. For the next offering, we plan to develop custom lab scripts where students get more hands-on experience by developing GPU and Hadoop applications using Java.

In the context of the distributed memory system programming section, the course did not rely on any dedicated HPC platform. Instead the students used lab computers connected using Gigabit Ethernet for executing parallel Java codes. For this purpose, a custom version of the MPJ Express software was released[2] that was capable of executing on a network of computers with no shared filesystem. Typically MPI libraries rely on shared storage medium to execute parallel jobs.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[2] S. Oaks and H. Wong, *Java Threads, Third Edition*, 3rd ed. O'Reilly Media, Inc., 2004.

[3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, 1998, pp. 212–223.

[5] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core HPC systems using Java," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 532 – 545, 2009.

[6] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, " mpiJava 1.2: API Specification," Northeast Parallel Architectures Center, Syracuse University, Tech. Rep., October 1999, http://www.hpjava.org/reports/mpiJava-spec/mpiJavaspec/mpiJava-spec.html.

[7] G. L. Taboada, J. Touriño, and R. Doallo, "F-MPJ: Scalable Java Message-passing Communications on Parallel Systems," *J. Supercomput.*, vol. 60, no. 1, pp. 117–140, Apr. 2012.

[8] O. Vega-Gisbert, J. E. Roman, S. Groß, and J. M. Squyres, "Towards the Availability of Java Bindings in Open-MPI," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. ACM, 2013, pp. 141–142.

[9] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[10] J. Dean and S. Ghemawat, "Mapreduce: A Flexible Data Processing Tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.

[11] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[12] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Springer-Verlag, 2009, pp. 887–899.

[1] http://seecs.nust.edu.pk/academics/doc/bese.php

[2] http://sourceforge.net/p/mpjexpress/mailman/message/32311993/