Methods for Teaching a First Parallel Computing Course to Undergraduate Computer Science Students

Mike Estep, Feridoon Moinian, Johnny Carroll, Chao Zhao
Computing & Technology Department, Cameron University, Lawton, OK, USA

Abstract – *Parallel computing has gained popularity in recent years. It takes advantage of multiprocessor computing equipment, such as multicore processors, computer networks, clusters, and massively parallel processors. Complex problems are solved more efficiently by overcoming the physical constraints of serial computing. The basic idea of parallel computing is to divide a large problem into smaller ones which can be carried out by mapping onto different processors; therefore those subtasks can be performed simultaneously. However, parallel computing instruction for undergraduate students is still somewhat in its infancy and continues to pose certain challenges in terms of resource availability. This article describes current instruction methods, obtaining of valuable instructor and student resources, and observations in teaching a first parallel computing class to undergraduate students at Cameron University.*

Keywords: Teaching, Parallel Computing, MPI, OSCER

1 Introduction

Parallel computing has gained popularity in recent years. It takes advantage of multiprocessor computing equipment, such as multicore processors, computer networks, clusters, and massively parallel processors (MPP). Complex problems are solved more efficiently by overcoming the physical constraints of serial computing. The basic idea of parallel computing is to divide a large problem into smaller ones which can be carried out by mapping onto different processors; therefore those subtasks can be performed simultaneously.

There are two basic parallel computing forms: data parallelism and task parallelism. Data parallelism processes a portion of the entire data set, such as sorting a big array with multiple processes; each process sorts a part of the array, and then processes interchange data such that all the elements held by process p_i are less than or equal to those held by process p_{i+1}. That is, the elements in the entire array are sorted in a non-descending order. Task parallelism focuses on distributing different functions to different processors with the same or different data.

Due to the increasing importance of parallel computing, Parallel Algorithms was one of the recommended courses in the ACM/IEEE CS2008 curriculum recommendation [1]. Furthermore, parallel and distributed computing was listed as one of recommended knowledge areas in the ACM/IEEE CS2013 curriculum recommendations [2]. Hence, it is essential to offer a parallel computing course at the undergraduate level. However it may be very challenging to teach such a class at a university without any parallel computing equipment. In this article, the authors discuss utilizing the Oklahoma Supercomputing Center for Education and Research (OSCER) [3,4] to teach a parallel computing class at Cameron University.

2 Methods

Cameron University is a five-year public regional university that offers a B. S. degree in Computer Science (CS) in the Computing and Technology department. CS 3813 Parallel Computing is one of elective courses in the CS curriculum, usually offered in the spring semester each academic year. Topics in CS 3813 include: parallel algorithms and implementations for sorting, searching, matrix processing and other problems, and efficiency issues of parallel algorithms on different architectures. In order to teach this class efficiently, several methods are utilized.

2.1 Instructor Training

Teaching a parallel computing class can be quite challenging since available instructional resources are relatively limited. To obtain the most current information on parallel computing, the authors attended OSCER's parallel computing workshop at the University of Oklahoma that is sponsored by the National Science Foundation (NSF). In the workshop, some essential topics in parallel computing were covered, such as parallel logic, basic Message Passing Interface (MPI) and advanced MPI, parallel equipment manufacturing and usage, and MPI programming exercises. Workshop attendants from different academic institutes exchanged ideas over teaching strategies, teaching methods, and teaching content. Through these activities, the authors gained much information and knowledge about parallel computing, useful for class instruction.

2.2 Inviting Parallel Processing Professionals to Visit the Cameron Campus

In order to motivate Cameron University CS student interest in parallel computing, experts were invited to deliver seminars on the subject. Basic topics included, but were not limited to: trends in parallel computing, interprocess communication with memory sharing in a multiple central processing unit (CPU) system, interprocess communication in a distributed system, and job balance in parallel computing efficiency. Meanwhile OSCER offered an online remote

account to each CS student over a supercomputer with thousands of processors and massive storage capability. Each student would run their parallel programs on the machine. The OSCER experts made a special trip to Cameron to teach students how to configure and use their accounts [3].

2.3 Visiting the University of Oklahoma Supercomputing Center

Considering that most Cameron students did not have much opportunity to see a real supercomputer, the authors organized a field trip to the Supercomputing Center located at the University of Oklahoma. The director of the computing center gave the students a well-organized tour, and the students obtained a great deal of information, such as supercomputer physical components, architecture, organization, and power supply. This knowledge was useful in helping students run their programs on a supercomputer.

2.4 Parallel Algorithm Development

Parallel algorithm development can be quite difficult for CS undergraduate students who are used to serial programming with single processes. To solve this problem, the authors focused on explaining the major differences between sequential and parallel computing. Sequential computing completes a task step-by-step in a certain order by one process, while parallel computing divides the task into a number of subtasks and then finishes tasks in parallel utilizing different processes. For example, if a search is conducted sequentially using a key item from a very large list, the key must be compared with each item in the list until the first appearance is met, or the end of the list is reached. In a parallel approach, the list can be divided into N sections, and then a section and key are passed to a process, next all processes run simultaneously, and finally execution results are collected by one designated process to reach a conclusion.

Parallel computing can use much less time to complete the same task. That is, if a single process uses time T to complete a task, and n processes may use equal or bigger than time T/n to complete the same task, where n is a positive integer, T/n is less than T if n is equal or larger than 2. This is called the "speed up" of how quick a task can be completed with n processes. The speed up concept can be expressed in the following formula:

$Sp_{(n)} = Ts / Tp$

Ts is the optimized sequential time needed by a single process to complete the task, and Tp is the parallel time needed by n processes running in parallel to complete the same task. For example, if one person can finish a painting job in one hour (Ts), as two (n) persons only use 40 (Tp) minutes to complete the same job, and then the speed up ($Sp_{(n)}$) is 60/40 = 1.5, that is, the job can be completed 1.5 times faster.

However, in general, parallel computing efficiency is less than 100%. Parallel computing efficiency can be defined as:

$Ep_{(n)} = Ts / (Tp * n)$ or
$Ep_{(n)} = Sp_{(n)} / n$

$Ep_{(n)}$ is efficiency, n is the number of processes, $Sp_{(n)}$ is the speed up, Ts is the optimized sequential time needed for a single process, and Tp is the parallel time needed for n processes. From the above example, $Ep_{(n)} = 1.5 / 2 = 0.75$ or $Ep_{(n)} = 60 / (40 * 2) = 60 / 80 = 0.75$. Therefore the efficiency is 75%. In conclusion, a parallel algorithm is faster, but it is also expensive because the interprocess communication always causes some overhead. It is of significant concern to attempt overhead reduction in parallel algorithm development.

Generally speaking, a parallel algorithm may have three parts: a common part, a master process part, and a slave process part. The common part includes necessary variable declaration and initialization, and shared functions declaration and implementation. The master part is designed to divide the given data set into subsections, send a subsection to a slave process, collect the computing results from the slave process if necessary, and finally display the results if required. The slave part is used to complete a subtask that includes: receiving necessary data from the master, completing calculation on the data, and sending the result back to master if required. In summary, a parallel algorithm design could be expressed in the following common function prototypes…

```
main function
{
      variables declaration and initialization;
      if (master)
      {
            code that will be executed by the master
            process;
      }
      else
      {
            code that will be executed by a slave process;
      }
            program termination part;
}
```

2.5 Teaching Methods

In CS 3813, the following topics were covered: parallel computing without communication among slave processes, job balance and dynamic job assignment, parallel computing with communication among slave process, derived data types, and new communicator creation. MPI was used to implement parallel algorithms in C/C++. In order to help students understand and master the above topics, typical examples were given from *Using MPI Portable Parallel Programming with the Message-Passing Interface* (2^{nd} Edition) [5].

2.5.1 Numerical Integration

Numerical integration to compute pi was used to teach parallel computing without communication among slave processes:

Since
$\int_0^1 1/(1+x^2)\,dx = \arctan(x)\,|_0^1 \arctan(1) - \arctan(0) = \pi/4$,
$\pi = 4 * \int_0^1 1/(1+x^2)\,dx$

So that we can integrate the function
$f(x) = 4 / (1 + x^2)$

When
$x = 0$, $f(x) = 4$, and $x = 1$, $f(x) = 2$

The x interval is first divided between 0 and 1 into N sections, each section being a small rectangle with width = 1 /N, and height = $4 / (1 + x^2)$, and then the rectangles are added together to obtain pi's value. To do so, first the master process accepts an N from the keyboard, and then broadcasts it to all processes. Secondly, all the processes receive N, find the width of the small rectangles by 1/N, and then find the mid-point of the width for process p_i using:

mid-point = width * (p_i's rank + 0.5)

For example, if p_i's rank, where p_i's rank is between 0 and N-1, is 1 and N = 10, then the mid-point is 1/10 * 1.5 = 0.15, and therefore the second process will start its calculation at x = 0.15. Each process takes one or more of these small rectangles to compute their areas. Finally after completing a calculation, all processes send the calculation results to a destination process, and then the destination process adds all results together to display pi's value. If the computing result is sufficient enough, the computing process will be stopped; otherwise another round of computing may be conducted as described. It makes sense that the bigger N is, the better pi value will be expected in a certain range. This is a good example because calculation for each section is independent from each other, and therefore these processes can run simultaneously. To enhance student learning, it was given to the students as a programming assignment using n processes to sum a large integer array.

2.5.2 Job Balance

Job balance is an important consideration in developing a parallel algorithm because CPUs involved in parallel computing may have different computing capacities. Therefore some CPUs may be busy all time, while some CPUs may be idle. CPUs are the most valuable computing resource, and it is desirable to keep them as busy as possible. To do so, CPUs should receive a new sub job once the current sub job is finished, until the entire job is completed. That is, it is important to allow faster CPUs to complete more jobs than slower CPUs. Matrix multiplication was used as an example to explain this topic. Matrix multiplication can be expressed as:

A (m, n) * B (m', n') = C (m, n')

where m and m' are the number of rows in A and B and n and n' are the number of columns in A and B respectively, and n must be equal to m'.

Master process performs the following in order:

A. Broadcasting matrix B to all slave processes;
B. Sending a row of matrix A to each process.
C. Receiving a row of matrix C from a slave process.
D. Copying the received row into matrix C
E. If the number of sent rows is less than the number of rows in matrix A, repeat B, C, D, and E until the job is done.

Slave process performs the following in order:

A. Receiving matrix B;
B. Receiving a row r of matrix A;
C. Multiplying row r to matrix B to produce a row of matrix C
D. Sending the resulted row back to the master process
E. Repeating B, C, and D until the completion notice is received.

This algorithm allows "self scheduling", that is, after a process completes its job, another job will be given until the entire job is done. Matrix multiplication was assigned to the students as a programming project.

2.5.3 Communicator Creation

In parallel computing, processes are commonly assigned to a number of groups, and each group completes a specific task. There may be many groups, but a process only belongs to one group. A group of processes with a unique context assigned by the system is called a communicator.

The Monte Carlo method to compute pi was introduced to explain this concept. The Monte Carlo method is a class of computational algorithms that relies on repeated random sampling to compute results. This method may not be the best way to compute pi, but can be an efficient way to show how to create a new communicator. The basic mathematical algorithm is:

If a circle of radius r = 1 is inscribed inside a square with side length 2, then the area of the circle will be $\pi * r^2 = \pi$ and the area of the square will be 2 * 2 = 4. So, the ratio of the area of the circle to the area of the square will be $\pi/4$. This can be expressed as:

ratio = $\pi / 4$ such that
$\pi = 4 *$ ratio

where ratio can be determined by the number of points inside of the circle over the number of points inside of the square. For any point (x, y), if $x^2 + y^2 < r^2$, where x and y are

coordinates of the point and r = 1, the point is inside of the circle and square, and otherwise only inside of the square. Therefore ratio is:

ratio = the number of points inside of the circle / (the sum of points in the circle and the square)

Since the calculation for each point is independent, it can be completed perfectly in parallel. There are two different tasks in computing pi: generating a set of random numbers and calculating points from a given random number set. All processes can be assigned into two groups: one group is a server group that is responsible to generate a set of random numbers, and another group calculates points based on the random number set. Then the formula $\pi = 4 * \text{ratio}$ is used to determine pi's value. A communicator can be created for each group. In the MPI, a set of routines are offered to create a new communicator from an existing one. The algorithm is: (1) create a new work group, world group, from the default communicator that includes all the processes, (2) exclude the server process from the world-group to form another work group, worker-group, (3) create a new communicator using the worker-group that includes all the processes except the server process, (4) free the world-group and worker-group. The results are two communicators: a world-communicator (default) that contains all the processes and a worker-communicator that only includes all the worker processes. In the worker-communicator, worker processes cooperate together to complete the calculation. The server process generates a random number set if noticed. The algorithm is shown below:

Server process:

A. generates a set of random numbers
B. sends the set to worker processes
C. if job is not done, repeat A and B, otherwise terminates.

Worker processes:

A. receive a set of random numbers,
B. compute points from the set
C. determine if the job is complete; if not complete, request server to produce another set of random numbers, and then repeat A and B; otherwise send the server a job complete signal to end the calculation

The worker communicator contains fewer processes than the world communicator, such that it may save time on collective operations, therefore it may result in a better performance.

2.5.4 Derived Data Types

Derived data types are very important because they provide a useful means to pack related data that could be homogenous or heterogeneous together as a single data unit to avoid multiple data exchanging among processes. In MPI, data will be sent as is. Hence, it is crucial to pack data together properly before sending them. The data type in MPI is an object that consists of a sequence of the basic data types, such as MPI_INT, MPI_CHAR, MPI_FLOAT, MPI_DOUBLE, and displacements of each of these data types. These sequences and displacements can be described in the typemap:

typemap = {(type0, disp0), (type1, disp1), …, (type $_{(n-1)}$, disp $_{(n-1)}$)}

Data type tells MPI how to interpret data when data is sent or received, while displacement tells MPI where to find data bits when sending them or where to store data bits when receiving them. For example, if typemap = {(double 0), (int 8)}, then the low bound that can be considered the location of the first byte described by the data type is 0, and the upper bound that can be considered as the location of the last byte described by the data type is 16 (8 +4 +4), therefore 8 is the number of bytes occupied by the variable with double data type, the first 4 is the length of an integer, and the second 4 is the number of bytes used as padding to meet system alignment. The difference between the lower bound and upper bound is called the extent which defines as the length of one derived data. So the extent is 16 in this instance. In C, one of the most common alignment requirements is that the address of an element in bytes is a multiple of the length of that element in bytes. If there are 4 such elements in a buffer, then double variable addresses will be 0, 16, 32, and 48. MPI provides a set of routines to get the lower bound, upper bound, and extent.

There are two cases: all elements in the derived data type have the same data type and elements have different data types. In the first case, all data have the same size so that there is no need to add padding bytes to meet the system alignment requirements. Therefore, one can simply pack the elements by using an MPI routine MPI_Type_continuous, commit the new data type, and then use it and free it after usage. The idea is shown in the following code segment:

```
typedef struct
{
float x, y;
} point_type;

point_type Points[100];
…
MPI_Type_continuous (2, MPI_FLOAT, &new_type);
MPI_Type_commit (&new_type);
MPI_send (Points, 100, new_type, dest, tage, comm)
…
MPI_Type_free (&new_type);
…
```

In the second case, a derived data type contains different default data types such that one has to determine the number of data types, the number of elements in each data type, and displacement for each data type. Fortunately, MPI offers a set of routines to complete this complex task. This was introduced to students as following:

```
typedef struct
{
      int pid, priority;
      double arrival_time, cpu_time, start_time;
}job_type;

job_type jobs [100]; //declare an array of jobs
int blocks [2] = {2, 4}; //set up 2 blocks
MPI_Datatype types [2];
MPI_Aint displacements [2];
MPI_Datatype jobs_type;
/*initialize displacements */
MPI_Address (&job_type.pid, displacements);
MPI_Address (&job_type.arrival_time, displacement +
                  1);
/*initialize datatypes */
types [0] = MPI_INT;
type [1] = MPI_DOUBLE;
/*make displacements relative */
displacements [1] = displacements[0];
displacements [0] = 0;
MPI_Type_struct (2, blocks, displacements, types,
                        &job_struct);
MPI_Type_commit (&job_struct);
/*use the derived data type just as default data types */
for (int i = 0; i < 100; i ++)
{
      MPI_Send( &jobs + i, 1, job_struct, i+1, tag, comm);
      …..
}
```

The derived data types in parallel may prove more difficult to understand than data types in sequential computing. Hence, it can require more effort and patience on the part of the instructor to help students understand and use derived data types in parallel.

3 Conclusions

From the authors' teaching practices and observations, the following conclusions are drawn: (1) instructor training in parallel computing is essential in ensuring prerequisite knowledge and skills in instruction of CS undergraduate students, (2) OSCER offers valuable resources that can be used to improve the quality of teaching and learning, and (3) Proper teaching methods provide instructors with an efficient way to deliver their teaching content. Further study should be helpful toward the understanding and development of sufficient student learning outcomes.

5 References

[1] **Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force, includes update of the CS2001 body of knowledge plus commentary**. December 2008, Association for Computing Machinery IEEE Computer Society http://www.acm.org//education/curricula/ComputerScience2008.pdf.

[2] **Computer Science Curricula 2013, Curriculum Guidelines for Undergraduate Degree Programs in Computer Science**. December 20, 2013, The Joint Task Force on Computing Curricula Association for Computing Machinery (ACM) IEEE Computer Society, http://www.acm.org/education/CS2013-final-report.pdf.

[3] **OU Supercomputing Center for Education & Research (OSCER)**. http://www.oscer.ou.edu/.

[4] **Using a Shared, Remote Cluster for Teaching HPC**. C. Carley, K. Larry, B. McKinney, C. Zhao, and H. Neeman, IEEE2013_indianapolis, ISBN: 978-1-4799-0896-7.

[5] **Using MPI Portable Parallel Programming with the Message-Passing Interface (2nd Edition)**. William Group, Ewing Lusk, and Anthony Skjellum, MIT Press, Cambridge, MA.