Parallelization of the Knapsack Problem as an Introductory Experience in Parallel Computing

Michael Crawford †
University of Mary Washington
1301 College Avenue
Fredericksburg, VA 22401
+1 518-338-5739
mcrawfor@umw.edu

David Toth
University of Mary Washington
1301 College Avenue
Fredericksburg, VA 22401
+1 540-654-1693
dtoth@umw.edu

ABSTRACT
As part of a parallel computing course where undergraduate students learned parallel computing techniques and got to run their programs on a supercomputer, one student designed and implemented a sequential algorithm and two versions of a parallel algorithm to solve the knapsack problem. Performance tests of the programs were conducted on the Ranger supercomputer. The performance of the sequential and parallel implementations was compared to determine speedup and efficiency. We observed 82%-86% efficiency for the MPI version and 89% efficiency for the OpenMP version for sufficiently large inputs to the problem. Additionally, we discuss both the student and faculty member's reflections about the experience.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer science education.

General Terms
Experimentation.

Keywords
Parallel computing, education, performance, knapsack problem.

1. INTRODUCTION
The 0-1 knapsack problem is an optimization problem with the goal of selecting items with weights and values in order to maximize the value of the items selected while keeping the total weight of the items below a set value [1]. In contrast to the bounded and unbounded variants of the knapsack problem that allow multiple copies of an item to be placed in the knapsack, in the 0-1 version of the knapsack problem, an item is either put in the knapsack or not [1]. The 0-1 knapsack problem is an NP-complete problem [1].

In an undergraduate parallel computing course, students learned to develop parallel programs with OpenMP and MPI. Because the course's instructor has found that NP-complete problems can help illustrate a number of concepts that can be useful when studying parallel computing, students were required to choose an NP-complete problem to have their programs solve. Students developed their programs using a multi-core server on campus and on their own systems and lab systems using the Bootable Cluster CD software [2]. Once the students had debugged their programs, they were able to run them on the Ranger supercomputer at the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. Ranger, which was just recently decommissioned after this project was completed, contained 3,936 distinct compute nodes with 16 general-purpose CPU cores each, for a total of 62,976 cores [3]. Running on Ranger allowed the students to compare the performance of the sequential version of their programs with the performance of parallel programs with OpenMP using 16 cores and using MPI with 16, 32, and 48 cores.

It is important to keep in mind that we were not attempting to devise a better algorithm than existing ones, but were focused on using the problem as a way of learning to use OpenMP and MPI and conduct some performance testing. Therefore, our results are not the important contribution of this paper. What's important is the learning that this project facilitated and our reflections about it. The first author, Michael Crawford, was a student in the second author's undergraduate course. This paper is written primarily from the student's perspective and describes the student's experience completing the course's final project.

2. RELATED WORK
A significant amount of research has been done on the 0-1 knapsack problem, which has numerous applications in business. Balas and Zemel developed an algorithm, as did Fayard and Plateau, and Martello and Toth [4, 5, 6]. Pisinger also developed algorithms, as have others [7, 8]. More recently, parallel algorithms have been discussed by a number of people. Loots and Smith developed a variation of a branch-and-bound algorithm to solve the problem for large numbers of objects [9]. Chen and Jang also developed parallel algorithms, as have others [10]. Even more recently, Pospichal et. al. have developed a parallel genetic algorithm to solve the 0-1 knapsack problem using GPUs [11].

† Undergraduate student

3. METHODOLOGY

3.1 Sequential Algorithm

Since the goal of the project was to become comfortable with OpenMP and MPI and to conduct some performance testing, rather than to implement an optimal algorithm, we chose to implement a brute force algorithm to solve the problem. The brute force algorithm could be parallelized with OpenMP and MPI with only minor changes and was easier to parallelize than a more complicated algorithm. The algorithm generates each possible permutation using the C++ standard library's next_permutation function. As each permutation is generated, the items in the permutation are placed in the knapsack from left to right, as long as there is space. Once an item in a given permutation is encountered that does not fit in the knapsack, no further work is done with that permutation and it receives the score of the items that did fit in the knapsack. If the score of the permutation is the largest one so far, the permutation and its score are saved as the current maximum. Then the next permutation is generated and scored until all permutations have been generated and scored and the permutation with the maximum score has been determined. To ensure reliable and reproducible results, the item sets were pre-generated and stored in a text file which was used by all three versions of the program. This allowed for easy testing of the sequential algorithm to prove it finds the optimal set of items, and subsequently easier verification of both parallel versions.

3.2 Parallel Algorithm

We created a parallel version of the program with OpenMP and then another one with MPI. Like the sequential algorithm, both the OpenMP and MPI algorithms were brute force. The parallel algorithms divided up the different permutations to test amongst the available CPU cores, with each core running a thread in the OpenMP version and each core running an MPI process in the MPI version. Each thread or MPI process tested an equal share of the permutations. Our sequential brute force, lexicographic permutation-dependent algorithm is nontrivial to parallelize, as each permutation is determined by the permutation before it. In order to divide the set of distinct orderings into a subset for each thread or MPI process, the factorial number system, or factoradic, was used [12, 13].

Factoradic allows one to calculate a specific lexicographic permutation of a set of numbers without having to generate each permutation between the first permutation and the one you are trying to generate. Thus, using the OpenMP thread ID number or the MPI rank and how many permutations per thread or MPI process will be computed, it is possible to determine the lexicographic permutation that any given thread or MPI process begins on.

The algorithm for OpenMP used factoradic to determine the starting permutation of each thread. Using the thread ID number and how many permutations per thread needed to be computed, each thread determined its own starting permutation and worked until it finished its assigned chunk of permutations to compute. Each thread kept its own best combination. After calculating the best combination from all of its permutations, in a critical section, each thread would compare its maximum value to the current global maximum value and update the global maximum if its value was greater than the current global maximum. The algorithm for MPI was identical to the OpenMP one, with the exception that at the end of their calculations, each MPI process sent its best permutation and corresponding score to the master process, which, starting with its own best as the default compared all of them to find the most optimal knapsack items. At the end of the calculation, the master node printed the result.

4. TECHNICAL RESULTS

When there were fewer than 11 items that could be selected to put in the knapsack, the sequential version of the program took less than 1 second. While one might think there is nothing to be learned from running a parallel version of the program for that few items, we observed that the MPI version of the program using 48 cores took 1 second which illustrates the overhead associated with it and shows that for small input sizes, the parallelism can actually cause slowdown. With 11 or more items, the sequential version of the program began to take larger quantities of time and took almost 62 minutes to complete for 14 items. In contrast to that, the MPI version of the program took less than two minutes when running on 48 cores on the same computer. The full set of running times for the sequential, OpenMP, and MPI versions is shown in Table 1.

Because the times the programs took to run were measured in seconds, it wasn't possible to accurately calculate the speedup and efficiency when there were fewer than 12 items that could be put in the knapsack. For 12 items, we could not calculate the speedup or efficiency of the MPI program that used 48 cores. We were able to put a very coarse lower bound on the speedup and efficiency. The speedups and efficiencies for the OpenMP version and the MPI version of the program are shown in Table 2 and Table 3 respectively. For 13 and 14 items, the speedup observed with OpenMP is not quite as close to proportional to the number of cores used as one might expect for an embarrassingly parallel implementation (0.94 and 0.89 for 13 and 14 items, respectively). This could have been a result of the algorithm used, which could result in a number of computations being short-circuited by some threads while other threads might have fewer computations that are short circuited. It is also possible that computations that are short circuited by some threads run further into their process than the ones short circuited by other threads. This could result in some threads needing to do more calculations than other threads, thus resulting in a load that is not balanced perfectly, which might account for some of why the speedup was not proportional to the number of cores.

In terms of speedup, the MPI version of the program when run with 48 cores saw a speedup of almost 40 for both 13 and 14 items. With MPI, using 16, 32, and 48 cores split on 1, 2, and 3 nodes respectively, we observed 86%, 85%, and 83% efficiencies with 14 items. The lower efficiency as the number of cores increased may have been due to the additional network traffic needed by using multiple compute nodes or also due to a computation imbalance between the nodes or between the cores on the nodes. We did expect MPI to be slightly less efficient than OpenMP, however, due to the required network communication. More interesting was that for the instance where there was only 1 possible item to put in the knapsack, the MPI run using 48 cores that took longer than the sequential and OpenMP runs and the MPI runs using fewer nodes (and cores).

With a brute force algorithm in parallel, we expected to observe linear speedup. To our surprise, our program did not achieve as close to linear speedup as we expected. However, we believe that the number of objects that could be put in the knapsack was too small to produce a good test, which might have caused this. Increasing the number of items available and the knapsack's capacity might have resulted in closer to linear speedup. This was

limited due to limits on the amount of hours on the supercomputer allocated to each student. We noticed that the speedup and efficiency were better with OpenMP than with MPI, which we believe was due to the extra network communication required for the MPI version of the program.

Table 1. Runtime of the Programs (sec)

Items Available to Put in Knapsack	Sequential Version	OpenMP Version (16 Cores)	MPI Version 1 Node (16 Cores)	MPI Version 2 Nodes (32 Cores)	MPI Version 3 Nodes (48 Cores)
1	0	0	0	0	1
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	2	0	0	0	0
12	23	3	2	1	0
13	316	21	23	12	8
14	3716	260	271	137	93

Table 2. Speedup of the Parallel Versions

Items Available to Put in Knapsack	OpenMP Version Using 16 Cores	MPI Version Using 1 Node (16 Cores)	MPI Version Using 2 Node (32 Cores)	MPI Version Using 3 Nodes (48 Cores)
12	7.67	11.50	23.00	> 23
13	15.05	13.74	26.33	39.50
14	14.29	13.71	27.12	39.96

Table 3. Efficiency of the Parallel Versions

Items Available to Put in Knapsack	OpenMP Version Using 16 Cores	MPI Version Using 1 Node (16 Cores)	MPI Version Using 2 Node (32 Cores)	MPI Version Using 3 Nodes (48 Cores)
12	0.48	0.72	0.72	>0.48
13	0.94	0.86	0.82	0.82
14	0.89	0.86	0.85	0.83

5. FUTURE WORK

There are several ways this work can be extended in the future. The first task is to create an OpenMP/MPI hybrid version of the program and conduct performance testing to see how that compares to the other versions of the program. We note that since Ranger has been decommissioned, we will need to redo the performance testing on a new supercomputer, such as Stampede, which replaced Ranger. The second task that should be done is to port the algorithm to run on CUDA-enabled GPUs and compare the performance of that version of the program to the results from the Stampede performance tests. We note that all the measurements in the future should be done in msec instead of

seconds, so more accurate comparisons can be made with small numbers of items. We would also like to implement different sequential and parallel algorithms such as branch-and-bound and dynamic programming algorithms to solve the same problem so we can see parallel performance comparisons of those algorithms in comparison to the brute force algorithm.

6. REFLECTIONS

6.1 Student Reflections

From a learning perspective, I think that this exercise was incredibly fruitful. Over the last couple of years, I have been exposed to a number of computationally intensive problems within the domains of theoretical computation and modeling and simulation. Knowledge on parallel processing is invaluable for the next generation of both computer scientists and natural scientists.

This exercise introduced me to the challenges and opportunities of high performance computing. It led me to consider the challenges of implementing parallel programs to solve problems in a way more theoretical classes cannot. While I was acquainted with NP-complete problems from my course on the theoretical foundations of computer science, this project gave me the opportunity to truly understand what it means for a problem to be NP-complete. It was enlightening to write an algorithm to "solve" an NP-complete problem; actually watching the runtime of the algorithm grow at a factorial rate with more items is humbling and seeing the limitations of a supercomputer is profound.

In retrospect, there were a number of changes I would make to both how I did the project and the course itself.

First, I wish I had paid far more attention to overhead as I coded my project. My algorithm had the potential to achieve nearly linear speedup, but I only saw roughly 85 percent efficiency. I wonder if my algorithm could have been written in such a way as to decrease the overhead and increase efficiency. In the future iterations of this class, it would be helpful to see a lesson on limiting overhead in HPC projects.

Second, we did not write programs that combined MPI with OpenMP, and I would have liked to have seen the performance of an MPI & OpenMP hybrid version of my algorithm.

Third, we did not talk about the impacts of using different compilers and optimizations during the course, so when I encountered a major performance difference of the same code running on the same hardware when it was compiled using different compilers, I was surprised. Previously, my mental model of compilers had naïvely assumed their equality for practical purposes. A brief exploration of compiler optimizations would be an interesting module for the course.

Fourth, each student was only given 3000 hours of CPU time on the supercomputer to conduct the performance testing. In the future, increasing this amount of time would provide the opportunity to test an OpenMP version of the program with 2, 4, and 8 cores in addition to with 16 cores, which might provide additional interesting results in terms of speedup.

Lastly, in order to learn the basics of parallel computing libraries like MPI and OpenMP, we were encouraged to solve our NP-complete problems using brute force algorithms. However, it would be interesting to see the difference in speedup and efficiency that other types of algorithms to solve the same problem would achieve. For example, what kind of speedup and efficiency would we see using a branch and bound versus a dynamic programming algorithm? While it would be impossible to expect students taking a three credit hour course to test three versions of each algorithm, each using a different parallel platform, it would have been a valuable juxtaposition to see different types of parallel algorithms pitted against each other.

As a student at a small liberal arts university, it was empowering to run code on one of the fastest computers in the world. The experience taught me humility in the face of computational intensity and provided me the tools to think in parallel. It will be a priceless course for students in the future.

6.2 Instructor Reflections

This course project spanned 8 weeks of a 15-week semester, making it one of the larger projects students do in our courses. The idea behind assigning such a large project was to give the students an opportunity to gain experience with not only the technical aspects of the material, but also with other important skills like time management, creating a poster, and giving a talk. The project was supposed to give students the opportunity to demonstrate mastery of the basics of OpenMP and MPI, as well as to perform some performance comparisons and give them experience running a program on a supercomputer. A side goal was to introduce the students to a variety of NP-complete problems.

Although I have taught a parallel computing course twice before, this was the first time I integrated this project and the use of a supercomputer into the course. Because of that, I learned a number of lessons, including that too many students will procrastinate if they are not given enough intermediate deadlines for a big project, as was the case with this project. A number of students were unable to complete the project, only developing an OpenMP implementation and not completing an MPI implementation or the performance analysis portion of the project. In general, the students that did not complete the project did not lack technical ability, but rather, they simply did not start the project until the last couple of weeks of the semester. Next time I teach the course, I will have parts of the projects due every 1-2 weeks.

Each student had to select an NP-complete problem for the project, but I limited the number of students who could choose the same problem to 2. This was done to force the students to produce a variety of posters and talks, so they would not all be the same and to keep things more interesting, as we as to prevent cheating. This caused some students to choose harder problems than they should have chosen. In the future, I will sacrifice the variety of the posters and talks to make the students more successful.

Students were supposed to develop a sequential program to solve their problem for small instances and then run the program to see how large an instance their program could solve in 24 hours. This instance of the problem was then supposed to be used as the baseline for their performance comparisons. Because the instance was supposed to take close to 24 hours to solve, I did not tell the students to do their timing in milliseconds because that granularity should not have been required for the performance comparison. However, some students benchmarked their code on systems other than the supercomputer and that led to them running their code for instances of the problems that were too small for the timing granularity of seconds to be as useful as needed. Because many students didn't run their program on the supercomputer until right before the project was due, they did not have time to test the programs with larger instances if the problem instances they used were too small. In the future, I will make sure that the students do their timing in msec instead of seconds.

The students were told that they each had 3000 hours to use on the supercomputer for the projects. At the end of the semester, a couple students said they wished they could have had more time on the supercomputer. After the projects were completed, those student were told they could use the remaining time from the educational grant that had not been used by others, but that was during finals week and the students didn't get around to using the extra time. Because students were supposed to have a program that ran for almost 24 hours sequentially and Ranger had 16 cores per node, doing that would have used 384 of the students 3000 hours. Since the students were supposed to run the program with OpenMP on one node and MPI on 1, 2, and 3 nodes, if they had no speedup, they would have consumed 3072 hours. I expected they would get significant speedup, but that there would also be problems and students would need to run their programs more than once to complete the project. Therefore, 3000 hours per student seemed reasonable. In the future, I want to rethink that.

6.3 Suggestions For Instructors

Our recommendations to a faculty member who adopts this project for their course are:

1. Have the entire class do the same NP-complete problem or half of the class do one and the other half of the class do another. We recommend the 0-1 knapsack problem and the traveling salesman problem.
2. Have the students who did one problems compare their speedups and efficiencies for their implementations and discuss the reasons for any differences.
3. Have prebuilt implementations of the problem(s) with data set(s) that students can run their algorithm on and compare their algorithm's solutions to the known correct solution.
4. Do not bother with presentations, which took time that might be better spent on other tasks. In particular, teaching the basics of CUDA would be helpful. However, having the students create a poster might still be useful.
5. Ensure that the compiler used on the supercomputer is available to the students on the local computer where they develop their solutions to ensure consistency. Discuss the different optimization levels for the compiler and ensure all the students use the same level.
6. Ensure students conduct timing in msec rather than seconds.
7. Have intermediate deadlines so that students do not get behind and are all ready to run their code on the supercomputer before the end of the course.
8. Request twice as much time on the supercomputer as you think the students will need to conduct the performance comparisons to ensure that even with mistakes, students have adequate time to test their implementations.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation through XSEDE resources with grant ASC120039: "Introducing Computer Science Students to Supercomputing in a Parallel Computing Course" and by the Texas Advanced Computing Center (TACC), where the supercomputer we used was located. We wish to thank XSEDE and TACC for their support.

8. REFERENCES

[1] Knapsack Problems: Algorithms and Computer Implementations, Silvano Martello, Paolo Toth, 1990. J. Wiley & Sons.

[2] BCCD | Bootable Cluster CD. http://bccd.net/.

[3] Texas Advanced Computing Center – Ranger-User-Guide. http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide. Updated 10/30/12.

[4] Balas, E. and Zemel, E. An algorithm for large zero-one knapsack problems. *Operations Research*. 28 (1980).

[5] Fayard, D. and Plateau, G. An algorithm for the solution of the 0-1 knapsack problem. *Computing*. 28 (1982), 269-287.

[6] Martello, S. and Toth, P. A new algorithm for the 0-1 knapsack problem. *Management Science*. 34 (1988).

[7] Pisinger, D. A minimal algorithm for the 0-1 Knapsack Problem. *Operations Research*. 45 (1994), 758-767.

[8] Pisinger, D. An expanding-core algorithm for the exact 0-1 Knapsack Problem. *European Journal of Operations Research*. 87 (1993), 175-187.

[9] Loots, W. and Smith, T. H. C. A parallel algorithm for the 0-1 knapsack problem. *International Journal of Parallel Programming*. 21:5 (Oct. 1992), 349-362.

[10] Chen, G. and Jang, J. An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing*. 18:7 (July, 1992), 811-821. DOI=http://dx.doi.org/10.1016/0167-8191(92)90047-B.

[11] Pospichal, P., Schwarz J., and Jaros, J. Parallel Gneetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. *Proceedings of the 16th International Conference on Soft Computing MENDEL* (Brno, Czech Republic, June 23-25, 2010), 64–70.

[12] Irene's coding blog: Factorial base numbers and permutations. http://irenes-coding-blog.blogspot.com/2012/07/factorial-base-numbers-and-permutations.html. Updated July 22, 2012.

[13] Factorial number system - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Factorial_number_system.