Parallel Algorithms Studying Report

Course work and Paper reading

By

Henry Xiao

Queen's University

School of Computing

Kingston, Ontario, Canada

November 2003

1 Introduction

As stated in the first chapter of our textbook [1], parallel algorithms are the most important aspect of parallel computation. A parallel algorithm defines how a given problem can be solved on the given parallel computer. The essential idea of a parallel algorithm is to divide a problem into subproblems which can be solved parallelly by multi-processors. In this way, the speedup which is the critical measurement of the quality of an algorithm is presented. However, the speedup itself is not a single measurement for parallel algorithms. Unlike sequential algorithms, parallel algorithms involves many new considerations, such as load balancing, processor communication, locality, etc. which ask for extra thinking and care when designing and analyzing parallel algorithms. So, in parallel world, we do have measurements like cost, work, etc to help doing the calculations and comparisons.

Because many new elements or issues are added to parallel algorithms, the papers that we can find about this topic are really various. It is interesting to see different papers tackle on the same field from different perspectives and yield different improvements. On the other hand, it is hard to set up a barrier of field for the parallel algorithm world since parallel algorithm is not "independent" from some point of view. Hardware structure is somehow involved when taking an algorithm to parallel. Furthermore, sometimes, data communication cost is not "free" and has to be considered into the time complexity. And some algorithms just directly developed to serve certain structure or model of a kind. Generally, the diversity of the parallel world does company with more complexity when dealing with parallel

algorithms.

Among those 8 papers that I picked and presented, each of them has its own targeted field. Different design and analysis techniques are applied. It is not surprising that most of them still discussing problems at the “bottom” level of computation. We never gave a thought or never had those problems in today’s sequential world. However, we have to face them with parallel algorithms. In this new field, many disciplines have to be established from the beginning. Those new ideas are quite a challenging for me. This report is organized as a summary of different things I learnt from the course and paper studies. Most of them are very basic understanding of many parallel algorithms as well as how to analyze them. I point out those important aspects when switching from sequential to parallel at Section 2. At Section 3 and 4, I discuss the algorithms among those 8 papers I presented. Section 5 is a little attempt to extending the algorithms from theoretical point to implementations since I did present a paper regarding parallel language implementation. A short summary at Section 6 is presented to summarize the knowledge from those paper studies.

2 Sequential to Parallel

The very first paper [8] I presented is about improving parallel merge sort with load balancing. At the time I picked up this paper, there were two things attracted my attention. First of all, merge sort is a well known sequential sorting algorithm which runs O(nlogn), and which is also one of the optimal sorting algorithms sequentially. More important, it does contain lots of parallel possibilities since the algorithm subdivides the whole problem into small subproblems which can be run parallelly. From this point, it would be an ideal algorithm to run in parallel. Second, the "load balancing" was a new technique to me at that time for improving a sorting algorithm. The way that the paper states its improvement is not very difficult. However, it does yield some significant speedup not only to the sequential algorithm but also to the parallel algorithm without load balancing. Here, it becomes clear why we want to use parallel algorithm and what things needed to be considered when designing a parallel algorithm. In this case, because the data set to be sorted is too large to fit in a uniprocessor's cache, sorting this data set in a sequential manner can take entirely too long that is limited by the memory access bottleneck. On the other hand, doing the sort in parallel can yield a significant speedup. Of course, this is only the motivation for this paper, but it is obviously an example of taking advantage of using parallel algorithm to deal with some traditional computational problems. Furthermore, load balancing is definitely a new terminology involved in parallel algorithms. Again, this noticed me that the price of obtaining some significant speedup from using parallel algorithm is the increasing complexity when designing the algorithm. In this paper, the load balancing takes

special care of utilizing all processors after each merge stage. The motivation for the authors to make this paper is clear that they realized the complex issues involved in parallel algorithm design. I think all those things above made this paper a great start for me to understand the parallel algorithm world.

Along the later papers that were presented in our class, quite a few algorithms were brought from their sequential versions. New ideas and techniques were "plugged" into those parallel version algorithms which made them even more efficient than I thought they could be. The O(1) time 3D Euclidean distance transform algorithm [9] is one example which actually achieves constant time bound. Also, many NP-complete and NP-hard problems from the sequential algorithms are presented to parallel since NP problems are accomplished with very high time complexities. Intuitively, people would hope to take advantages of parallel algorithm to calculate such problems more efficiently. We could find such kind of problems almost every week of our presentations like Hamiltonian path, Travel Salesman problem, multiprocessor scheduling problem, etc. The fifth paper [2] I presented by Aggarwal, Motwani, and Zhu from Stanford University is one of those papers which contains NP problems. The "classic" approach that most researchers followed to construct parallel solution for this kind of problems in their papers looks like explaining the sequential or parallel algorithms for the related NP problems first. Then throw out their idea which is either a new method or an improvement of existence. Finally, the related analysis is preformed to give time complexity or run time complexity for the presented algorithm, and the comparison is also done with the analysis.

Concluding above, it is clear that problems that we concentrate on with parallel algorithms are most times those hard problems like NP-complete, NP-hard in sequential manner. Intuitively, it makes perfect sense to study them because the motivation of using parallel algorithm based on parallel models is to give us more computational power. However, I guess another important question to be discussed in this section is how to design parallel algorithm over those hard problems from sequential algorithm.

One of the most important things that I learned from this serial of paper studies is the complexity involved in designing parallel algorithm as I stated in the introduction section already. Actually, it is almost impossible to just say transforming a sequential algorithm to parallel. Jeon and Kim's paper [8] discussing parallel merge sort with load balancing is probably the closest example we could find that contains some sort of "transformation" from sequential to parallel. However, the load balancing is introduced there. And we could clearly see those difficulties involved in utilizing multiprocessors. As stating at the next section, the structure or model of multiprocessors will make our parallel algorithm design much trickier than sequential algorithm design. Of course, many new problems studied in those papers are just totally new in parallel. Like Fu's fault-tolerant cycle embedding in the hypercube [7] paper, the study is directly tackled on a parallel model which may never be thought without parallel computational models. In Durand, Jain, and Tseytlin's paper [5], they studied the parallel I/O scheduling. Again, traditional edge coloring problem is used to help understanding the new problem. However, the situation is that we are dealing with distributed system now.

In this case, communication between processors has to be taken into consideration. Other papers presented in class introduced many new concerns from different perspectives such as preprocessing the data in order to use some efficient algorithm or take some advantage of a parallel model, etc. Generally, I collected my thoughts about issues that needed extra care in parallel world into two main aspects. Firstly, how the additional computational power provided by parallel computational models can be utilized. Load balancing, data preprocessing, etc. all belong to this kind of efforts. Then, how can effective manipulations be achieved when executing the algorithm. Specifically, a good parallel algorithm should be good at subdividing tasks such that each task can be executed with less information from others at each stage or iteration. We had many good algorithms such as prefix sum, sorting, etc based on different computational models in class which all have strong performances in this field. From an algorithm designer perspective, I think the above two considerations are two key things when we want to make an achievement algorithmically from sequential world to parallel world.

3 Parallel Models

Parallel model is no doubt the base of our parallel algorithms. Almost every paper that we found through the duration of this course was specified with certain parallel model at the beginning by the authors. Compared with sequential algorithm which is based on Von Neumann structure (basically), we can see the diversity of choices for parallel algorithms. However, I think we can still divide the complicated models into two basic cases. I will try to conclude the two cases about parallel models from algorithm design perspective at the following of this section.

The most common choice for parallel algorithm designers is obviously the *Parallel Random Access Machine* (PRAM). Just like we study sequential algorithms, in the *Random Access Machine* (RAM) model we can think preprocessing and communication only take constant time. The beauty of this is well-known as easy to analyze an algorithm. And the memory is assumed to be shared for the same reason of eliminating those hardware effects. We can see from the parallel merge sort paper [8] and 3D Euclidean distance transform paper [9], those algorithms designed based on PRAM model are quite general in terms of usability. Actually, if we review a little of these two papers, the way that the algorithms were presented is pretty familiar to us. The model “side-effect” was eliminated and we did not need to concern those specific hardware structures like mesh, star, hypercube, etc. Personally, I would like to consider them better than algorithms designed for specific structure or just serve for a model like the perfect load balancing for hypercube multiprocessors presented at paper [6]. However, as early as the second chapter from our textbook [1], we saw the great power from

using certain model for certain problems. It is unfair to measure a parallel algorithm just by the model it uses. And it is pretty impossible to talk about which model is better than another as we discussed in class. Even for the PRAM model, we know that there are a number of different ways for processors to gain access to memory. Exactly, the paper [9] specified the CRCW which is concurrent read and concurrent write at the title. And the O(1) time is the benefit from taking CRCW assumption. Here, it may be possible to state that general algorithm just taking PRAM model is rarely to be developed nowadays. Of course, this is not because parallel algorithms all have to be model-dependent, but PRAM based algorithms have been extensively studied. We can find many good algorithms from our textbook [1] almost all based on PRAM which cover many aspects of our computing problems. On the other side, papers we studied are all no later than half a year old. We can clearly see the open problems in parallel algorithm are most time designated for a model. In other sense, PRAM is actual not a practical model. It is not bad to design an algorithm for a more realistic model to make theory staying on its ground. Of course, as an algorithm designer, or a person who like to think about algorithm, the beauty of simplicity of PRAM is still irresistible. Just like in sequential world, RAM makes our discovery of algorithm comes so naturally.

It becomes obvious now what the second part I want to conclude here about parallel model related with algorithm. The other six papers I picked were all about algorithms served for different specific models. It is also not hard to get the most popular model out from those papers, which is the hypercube. The paper [6] states an interesting way to balance the job load among all processors in the hypercube

structure. The simple math function actually made a “perfect” result for loading jobs among all processors. Then the paper [2] looked at this problem from another side, the author formed a question and also gave possible solution. The question is how to keep balancing job load at the run time, which they called “load rebalancing problem”. As we could imagine, the solution would rather be complicated and not traditional at all. The paper [7] considered a really different problem as I could think of at that stage. The result was a little surprising to me that we could get around faulty nodes (processors) with 2n – 4 faulty nodes at most, where n is the dimension of the hypercube. The fault-tolerant feature really forced me to re-think lots of things about hypercube model. Hypercube model holds advantages against models like mesh and star as it is somehow a 3D structure which makes a great various number of ways for processors to communicate. Problems involving sorting, matrix manipulation, etc are naturally easy to be decomposed into 3D and deal with each dimension calculation concurrently. It makes even more sense when we are going to design those algorithms. Unfortunately, I wasn’t lucky to find some paper using other models. So the comparison will be somewhat incomplete from papers I presented. For the sake of completeness, I would like to take some papers I heard in class as examples of other models. I noticed that many scientific parallel computing problems presented in class were based on master-slave model, which I considered to be a star like model. Those problems are essentially taking the advantages of center-control provided by the “root” node. Also, there are high demands of computing power from those problems. Most cases, the testing can be done by computer network which is kind of distributed simulation of the star

topology. From some point of view, those algorithms are the middle stage transforming from sequential to parallel. If we take hypercube structure, there is no way to use computer network to test an algorithm based on hypercube, instead, the special hypercube model parallel computer has to be presented.

As I mentioned at the last paragraph, those algorithms from papers that targeted different models are more practical, and we can always see some implementation of the algorithm with results. The paper [2] which stated the load rebalancing problem does give specific algorithm that leads to implementation about solving the problem. Even the merge sort paper [8], which used the PRAM to design the algorithm, includes implementations on two different model computers (cannot be PRAM anymore) to get the testing results. Our textbook [1] does lots of other models and hybrid models like mesh-tree, etc. They are hardly to be found from those papers we presented. I would like to put the reason to simplicity just like using PRAM. It is true that we can not avoid model effect when designing a parallel algorithm. However, as the matter of fact, we do want to use the model as simple as possible so that the concentration can be focus on algorithm itself. In this sense, if we only need features from a mesh model, we do not bother to use a mesh-tree, or pyramid model. However, we can clear expect new powers coming from those models as specific problem is concerned. Like the PRAM case, once those basic models such as mesh, star, hypercube, etc have been extensively studied, our adventure will be for sure going to more complicated models.

My understanding about parallel model is still far from clear. Many models have showed their amazing power dealing with certain problems. What I did learn

is that one has to understand the parallel models to be a parallel algorithm designer. Without accurately mastering features of a model, we can hardly have an algorithm using its power. Definitely, many open problems will still come out related with this topic, and we will also keep finding interesting stuff here.

4 Parallel Techniques

Considering the size of this report, I have to put many great parallel things together here. Personally, I would never give a definition about parallel technique since I don't believe someone can find one. So, in this section, I am trying to explore those techniques used in the papers I picked. I have to remind myself that they are only a tiny part of what have been done in this field.

First of all, let's look at things that made most appearances among those papers. Load balancing is no wonder with the highest score. I picked three papers directly talking about this problem which consist one third of all my chosen papers. After reading those papers, it is clear to me why this problem becomes so popular at the present. Paper like merge sort with load balancing [8] gives a second thought of existing algorithm. The obvious question here is if I have such many processors, how I can utilize them, in other words, to keep them busy all the time. With the memory of sequential run time analysis still fresh in mind, we can see this is not an easy question to answer. The paper [8] is actually quite a simple idea to implement load balancing on a parallel version sorting algorithm. However, the performance, in terms of speedup over normal loaded algorithm is significant. The only extra work we need to do in this case is preprocessing the jobs among processors to achieve load balancing. After this paper, I found the perfect load balancing on hypercube paper [6] was more general to look at the loading problems. As we discussed at above section, hypercube model is probably the most popular choice today. The related load balancing problem is somehow trickier than I thought. By recognizing the level or dimension of the cube, the

authors managed to load and exchange jobs among processors quite efficiently. The extra care was taken to assure that the possible accumulating error is eliminated. Again, the whole idea is simple. The preprocessing work can also be done quickly without disturbing the job processing too much. Here, the algorithm fully uses the structure advantage from the hypercube. Those two papers are both about preprocessing jobs to achieve a balance. However, another good question is how to keep this balance at run time. It is very likely that we could lose load balancing when processors getting different sizes of jobs. In this point, paper [6]'s contribution is very limited as jobs can be different size. So, the loading rebalancing problem was pointed out by the paper [2]. This became a hard question and finally reduced to NP-complete by the authors. Indeed, we can not get perfect load balancing at run time with different size jobs. The naturally approach will be using approximation algorithms, and this is what the authors took in this paper. Intuitively, the greedy would be a very good choice as the paper stated. The 2-approximation was quite acceptable with the simple algorithm from greedy. However, because of the extra power we got from multiprocessors, the authors managed to find a smarter algorithm which is not simple as greedy, but yields a 1.5-aproximation ratio. My consideration here is that the algorithms are not as important as the idea itself. We can expect better algorithm in the near future for this problem. But the idea or questioning this question is of a deeper thought. Furthermore, we can see how hard those problems, just about to utilize the multiprocessors, can be.

If load balancing is stated right at the papers' titles, another common

technique used by many papers is hidden in the body of the papers. I found many parallel problems have been compared with sequential problems to make themselves cleared out. I rather like this way to state a problem, especially those problems for a new environment that I am not used to think with. The parallel I/O scheduling paper [5] is a very good example of this kind. The authors took the new parallel problem to compare with edge coloring problem and matching problem which are very common problem in graph theory. Then sent problem to distributed manner and provided solution specified where parallel computing could be employed to accelerate the process. Because of the deduction to a well-known problem, the new problem becomes familiar with us and quite understandable. Some paper like the fault-tolerant cycle embedding [7] also uses other well studied theories to help proving the results. The paper [7] represents the hypercube using graph structure. Many graph theory results are used as bases to construct the new theorems. Papers with new theorems are always hard to get through. But if the paper can employ some other theories, which may be sort of traditional, to help, it would be much easier for readers to follow.

One can sometimes use a comparison with sequential example to make the parallel problem clearer. However, the essential difference is what we have to focus on. The most common difference always mentioned at the early stage of the paper is partition or something like that. Partition varies a lot from an algorithm to another algorithm in parallel. Like load balancing problem, it is directly related with utilizing multiprocessors. However, partition is even more basic, and hence rarely studied along. Almost all parallel algorithms contain some kind of partition step. Of

course, in those eight papers, I can certainly find many examples. The 3D Euclidean distance transform paper [9] provides a partition algorithm as the first step of its calculation. The merge sort paper [8] is essentially a partition algorithm. The block independent set paper [4] makes the partition with the matrices to distribute smaller data set to different processors. It is not a hard job to list all the partition idea from those papers. Partition is the very first thing towards distributing jobs among processors. If we want to design a parallel algorithm, we have to find the way to partition the whole data set. Of course, sometimes, it is obvious how we can partition the set like the merge sort, since the sequential algorithm already gives a clear clue about how to. But, sometimes, it is not that obvious, like the block independent set algorithm from paper [4]. The improvement of that algorithm comes from the transformation from global operation to locate operation. In other words, the advantage of partition data set with more independency. Here, the goal of partition is certain that we want to get the whole data set divided into small sets so that each set is independent to others and the size of set is uniformed ideally. However, as I mentioned in the model section, many practical problems presented in class related with networking, biocomputing, etc use master slave model, which means the distributed small data sets can not be independent, and the relation is across the “root” processor. Uniformed size is also a basic assumption since many parallel algorithms take the job size as 2^n where n is a big factor. In our binary world, it is certainly the ideal case to get job of size 2^n. The paper [4] is a very interesting instance to see the effects of making data sets more independent. The problem and algorithm presented in that paper is to calculate the distributed sparse

matrices using block independent set. Previous study already figured out a parallel algorithm to divide the sparse matrix into small block and search each small block for independent set. The thing that goes wrong in the previous algorithm is that when we search the block independent set, we have to run it as a global operation which is to check all blocks instead of staying at the local block. This causes the "double node removal problem" because the processors run parallelly based its own global calculation and may get smaller size independent set than optimal set. The paper's new strategy is to set up rules through observation and cut the dependencies among blocks by limiting global operations. The consequence is the speedup gained from less global operations, and it partly limits the "double node removal problem". The price is we have to sacrifice the block independent set calculation accuracy, which means the size of the independent set is probably smaller than calculated from global operation.

Another interesting aspect to look at partition is from the model perspective. Again, the most popular partition taking from model structure is the partition related with hypercube. The 3D structure of the hypercube really gives us lots of intuitions to manage processors through those connections. The beauty of hypercube is related with the number of processors it has, which is 2^n. In the binary world, we would always give a smile if we have something as the size of some power of two. The paper [6] states perfect load balancing algorithm for hypercube, which is a good example of taking advantage of the model. By simply labeling each processor with binary sequence from 000 to 111 with the regulation that adjacent two processors have hamming distance one counting on their labels, many ways

of partition and communication can be done based on different problems. We already saw the sorting and matrix multiplication in class using hypercube. The dimension of the hypercube is a natural partition, which is a born advantage verse other 2D structures like mesh, star, etc. And also, the hypercube is probably the simplest regulated 3D structure we can use today. Of course, some structure like tree also has its advantage dealing with some problems. If we take the merge sort case, tree structure probably more suitable for the partition. The way I like hypercube is more from its ability to re-partition itself based on its labels at different phase. The hamming distance is a very useful term to partition hypercube from different directions. This makes the way of broadcasting and communication data through processors every efficient and parallel features are sort of embedded. In general, I think hypercube is probably one model we have extensive studied at these days. I believe there will be more studies on other structures which will give us more amazing results about partition.

There are definitely many more techniques even only taking those 8 papers I used. However, I want to concentrate myself to another big part of parallel algorithm design which is the analysis part at following. Parallel algorithm analysis shares lots of common things with sequential algorithm analysis. But as we can see from any paper, the difference is not hard to identify. I wrote in the introduction section that the new measurements like cost, works, etc were introduced to analyze the parallel algorithms. I guess the reason for us to take the new measurements is because of the fairness. We can hardly say that an O(1) time algorithm using $O(n^2)$ processors is better an O(n) algorithm using O(n) processors.

The costs for both two algorithms are $O(n^2)$. If we look at the paper [9], the authors did state an O(1) time algorithm. However, when comparing with other algorithms, they provided the number of processors required as well. And the authors also tried hard in their design to reduce the usage of processors. The partition algorithm was designed due to this reason in the paper. For parallel algorithm design, we have to concentrate one more task besides improving the time bound that to reduce the number of processors. Some papers, like [5], are pretty abstract with less testing issue involved, which only give possible time bound but not cost. However, papers, like [8], actually implemented their algorithms and fully tested. Their analysis is fully equipped with parallel measurements. Furthermore, because of the time bound in this case has to be measured more accurate since practical testing has been done. The communication and transmission can not be assumed constant time any more. The paper [8] did a subtle job to get a good estimation taking communication and preprocessing time as a simple linear function based on coarse-grained model. The time bound they got there is slightly above purely theoretical time bound but more close to their experiments. The final function sequence from the paper is much simpler than I thought it would be. This actually tells that it is possible to use some math model to simulate our really time situation and make a more practical time bound. Of course, it is useful for implementing some algorithm with actual machines. But as the first step of algorithm design, this has to be avoided for the truth of simplicity again.

The difficulty of producing parallel algorithm analysis I feel from reading those papers is that hard to identify which part of the algorithm is parallel and why.

The confusion always starts from the partition phase. Like in the paper [9], it took me a lot of time to understand how the authors partitioned their data set using $N^{1+\varepsilon}$ processors and achieved constant time bound. Then the ε actually equals to $1/(2^{c+1}-1)$ which gave me a headache about why using this magic number. I guess up to now, I understood how it works, but not where they got it. I recognize that is why idea is more ingenious. I would like to end this section here with the above two part of parallel techniques I learned from paper studies. I think a good approach toward designing a parallel algorithm starts from considering possible sequential solutions and examining them to explore parallel procedures like partition. Then we have to concentrate on the parallel issues like load balancing etc, in order to make algorithm parallelly efficient. Finally, many other parts have to be checked, like sometimes, we can use less number of processors to do the same job especially partition without increasing overall time bound. In general, I think parallel algorithm techniques are still hard to conclude. Unlike sequential algorithms, where we can talk about dynamical programming, greedy approach, etc, many parallel algorithms are still on going to be developed. And because of the various parallel models used, I would not consider to category such techniques.

5 Parallel Implementations

The implementation of parallel algorithm is somewhat out of the bound of this course. However, for the faith of motivation of theory works, I still decide to put a few words at the end to talk about the implementations.

There were many presentations in class about practical problem algorithms in different areas such as networking, biocomputing, and even economy. What differs those algorithms from purely theoretically designed algorithms is that those algorithms seem all based on today's common machines, thus have less parallel abilities. Of course, theory is always above practice, and we can think theoretically without worrying physical constraints. From this point, our course about parallel algorithms would give us more fun in the future. I did not really choose a paper of talking about a really problem for the fact that many really world problems are too specific and hard to be understood without really touching it. However, as a person holding strong interest with computer science, my question is how to use the parallel programming languages which support implementing those parallel algorithms. I guess we can think this as a parallel implementation in computer science.

For above reason, I did present a paper regarding a parallel language. The paper [3] introduces an extended ANSI C for processors with a multimedia extension. In the case of parallel, the physical machine which the language underlies is only SIMD (Single Instruction Multiple Data). It is very practical at the present, but not very parallel since when designing most parallel algorithms, we assume MIMD (Multiple Instruction Multiple Data) machine already. The good

thing comes out from that paper is that the language can be understandable by programmer like me used to sequential programming, and make a good first taste for me to see parallel programming difference. I think it is a debatable question whether to keep sequential programming format for easy understanding, or to make changes for taking more parallel features. I did not do enough research with parallel languages, but from the paper, nowadays, the first approach is more applicable since really parallel machines are expensive and hard to reach by most programmers. The paper [3]'s parallel language extending from ANSI C is based on quite simple idea to take the power of multiple data ability from the processor. Like the matrix manipulation, the manipulation functions can actually fetch multiple data in parallel and processing same operation on different data at the same time. Obviously, this will make the manipulation faster than fetching a single datum at a time. In order to do this, new expressions have to be introduced and the operations like summing etc have to be redefined for parallel. Practically, those redefining jobs would rather be complicated. Of course, the authors did successfully implement the language with all those parallel features. The primary question is how parallel this language could be. I consider this case quite limited since it is based on the sequential C language, and C is primarily designed for single processor computers. The machine used in this paper is still single processor equipped with multimedia extension which is sort of parallel ability extended on a single processor. So, the language can only be said to be a parallel language for parallel enhanced single processor machine. But, it will be my stop here for exploring parallel implementations.

6 Summary

To summarize this course and paper study, I would like to briefly go through all the things I talked above. Three important aspects I have learned from the course and paper studies, as I listed at section 2, 3 and 4. First of all, the sequential algorithms and parallel algorithms share a lot of things in common. So many good parallel algorithms do have sequential elements, and parallel algorithm designers also like to think with sequential algorithms, then above that. There are some common elements I noticed from those papers, partition is one of the basic steps we need to keep an eye on for most parallel algorithms. The primary question to ask when designing a parallel algorithm is how to utilize the multiprocessors. Many works have been done related with this topic. However, since this question goes with every single parallel algorithm, there will be various ways now and future to deal with different situations. Diversity of parallel algorithm also comes from different parallel models as discussed at Section 3. This diversity definitely adds complexity to parallel algorithm, which is interesting to observe, but causes lots of extra efforts from algorithm designers. This diversity also expressed itself extensively when we tried to take a closer look at those techniques from those papers at Section 4. In general, the structure of how to organize multiprocessors, in other words the model, and the way of how to distribution data and communication among processors, in other words the partition and utilization, make the parallel algorithm design involving many new aspects totally new from sequential algorithm.

At end of this report, I realize that I am still on the surface of parallel world. However, I think I should appreciate this opportunity to get my taste of a different

computing world. From the future perspective, parallel algorithm may eventually lead our computing power to the next generation. It is also very important to design parallel algorithm in theoretically general. If parallel algorithm belongs to the next generation of algorithm design, we do not know how long we will keep in this computing base, even we do not know whether we will stay with binary world. So, as I wrote at Section 5 about implementation of parallel algorithm, for the taste of power of parallel computing, we would like to see some real examples in practice to convince the abilities of parallel computing. I firmly believe this great idea to organize multiprocessors together, which now called parallel, will change they way of computing. And this course and those papers have showed me a really colorful new computing world. The essential breakthrough brought by parallel computing is that it provides us a new environment to design algorithms which can break the old physical limits, even more exciting theoretically is that we do not have a physical bound for parallel computing yet. This is definitely the driving force for professionals to keep researching here. Also, it is no wonder the motivation for me taking this course, and reading those papers to know this field.

References

[1] S.G. Akl, Parallel Computation: Models and Methods, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[2] G. Aggarwal, R.Motwani, and A. Zhu, The load rebalancing problem, SPAA '03, June 7-9, 2003.

[3] P. Bulic and V. Gustin, An extended ANSI C for processors with a multimedia extension, International Journal of Parallel Programming, Vol. 31, No. 2, April 2003.

[4] C. Chen and J. Zhang, A fully parallel block independent set algorithm for distributed sparse matrices, Parallel Computing, 29, 2003, 1685-1699.

[5] D. Durand, R. Jain, and D. Tseytlin, Parallel I/O scheduling using randomized, distributed edge coloring algorithms, Journal of Parallel and Distributed Computing, 63, 2003, 611-618.

[6] G. E. Jan and Y. Hwang, An efficient algorithm for perfect load balancing on hypercude multiprocessors, The Journal of Supercomputing, 25, 2003, 5-15.

[7] J. Fu, Fault-tolerant cycle embedding in the hypercube, Parallel Computing, 29, 2003, 821-832.

[8] M. Jeon and D. Kim, Parallel merge sort with load balancing, International Journal of Parallel Programming, Vol. 31, No. 1, February 2003.

[9] Y. Wang and S. Horng, An O(1) time algorithm for the 3D Euclidean distance transform on the CRCW PRAM model, IEEE Transactions on Parallel and Distributed Systems, VOL. 14, No. 10, October 2003.