Parallel Graph Algorithms

Algorithmic Graph Theory Study Report

By

Henry Xiao

Queen's University

School of Computing

Kingston, Ontario, Canada

November 2003

1 Introduction

Backing to 1736, when Leonhard Euler published his famous paper [3] asking whether or not it is possible to stroll around Konigsberg (later called Kaliningrad) crossing each of its bridges across the Pregel (later called the Pregolya) exactly once, which was recognized as the earliest paper on graph theory later, he probably did not expect there would come such a powerful mathematical tool so called computer later in this world to extend this theory started by his paper to the next generation. At recent two decades, much has happened in graph theory no less than elsewhere: deep new theorems have been found, seemingly disparate methods and results have become interrelated, entire new branches have arisen as stated at the beginning of book [2]. The interesting thing to observe theoretically is that how graph theory working at computer science area, which of course is our course's topic - algorithmic graph theory. From computer science perspective, graph theory has been extensive studied with related problems. Furthermore, various graph algorithms have been developed along with our increasing computing power. If we look at this from other side, graph problems and algorithms always accompany with relative high time complexity in terms of computer science solution. So, it is predictable that some structural change of computer architecture to enhance our computing power will eventually reshape our algorithmic graph theory. In this sense, parallel computing, one of the most popular new computing architecture nowadays will no doubt give us some fascinating results in graph theory.

Actually, developing parallel graph algorithm is not new anymore. McHuge included a chapter in his graph theory book [4] to talk about parallel graph algorithms, and the book was published in 1990. However, since the parallel algorithm has not been as well studied as sequential algorithm, and various parallel computing models involved, people did not really design algorithms in terms of graph theory, instead, many basic computing problems including some graph problems have been studied. So, in general sense, parallel algorithmic graph theory is still not here yet. As a person with a great enthusiasm about knowing different algorithms, I would like to pick up some parallel algorithms from various publishes, and try to arrange them in the way that we can get some idea about how parallel computing makes the difference in algorithmic graph theory field.

This report is organized to serve the above topics. Some personal understanding about how we can relate graph theory with parallel computing as well as how to utilize the additional power provided by multiprocessors in graph algorithms are discussed at Section 2. In order to look at those parallel algorithms, some notations and operations have to be introduced, and Section 3 is there for this purpose. I consider a better way to get a good taste or understanding of parallel algorithm with graph theory is through studying some classic problems. So I present some classic graph problems in parallel algorithm world that I collected at Section 4 as the beginning of our adventure. Section 5 is to look at some possible “practical” uses of parallel graph algorithms. It is only a little piece of what have been done and what is going on in this field. Finally, I summarize this report at

Section 6, pointing out some interesting things I have learned so far. My generally hope is that this report may help you get to know and enjoy this amazing new parallel graph world.

2 Graph Theory and Parallel Computing

Parallel computing is different from sequential computing most with its various computing models, in other words, the multiprocessor structure. From some point of view, the algorithm design in parallel is getting more flexible companied with the increases of algorithm complexity caused by the model uncertainty. Of course, the ultimate idea in parallel is somehow coming from taking the advantages of those multiprocessor models. And more important, I guess we already make the connection between graph theory and parallel computing since the models, which can be simplified as organization of processors without losing too much of generality, is nothing new to be represented as graphs. As a matter of fact, many attributes of certain graphs like tree, star, hypercube, etc have been wildly used in parallel computing models. And the advantages exposed by graph theory of a certain structure are also the intuitions for researchers to think about algorithms in parallel fashion. In general, I think it is fair to say parallel computing was born with graph theory related. At the following works in this section, I would like to explore this relationship with some popular parallel computing models as well as their applications.

Binary tree, one of most extensive studied graph structure in our graph theory is no wonder a good start example here. Figure 1 is an example of a binary tree interconnected network with 16 nodes (processors in terms of parallel computing model) from Alk's book [1]. In general, it is easy to see that this kind of complete binary tree organization has *log(n)* levels, where *n* is the total number of

nodes. Now, in order to see the power of this simple structure, let's examine a problem where it is asked to find out the maximal or minimal value from a given list (assume the size n of the list is 2^k). Sequentially, we can do this simply by comparing two value and keeping the maximum or minimum. Of course, the traversal will take us at least linear time, which is $O(n)$, and which is optimal in this case. However, we will see that this binary tree parallel model can solve this problem in logarithmic time, which is faster than linear. Actually, it is not hard to see this algorithm from Figure 1 at all. We simply feed two values to the leaf processors. Each processor then compares the two values, and sends the larger or smaller one to its parent processor. Finally, by repeating this process at each level, we will eventually get the two values to the root processor, and the last comparison by the root processor will be able to tell us the maximal or minimal value. Clearly, we only need logarithmic time since the total number of steps is equal to the height of the tree.

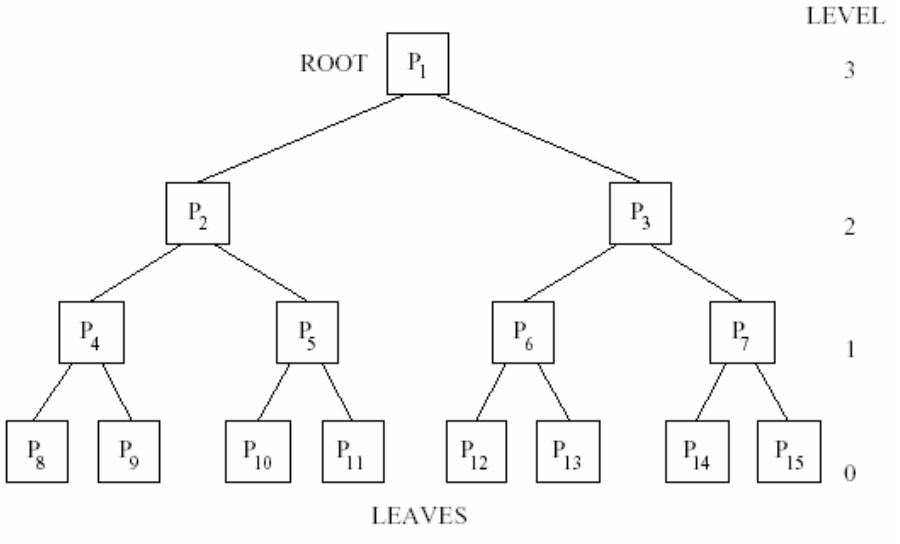
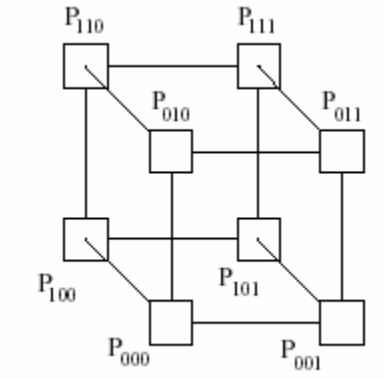


Figure 1: A tree interconnection network. [1]

A more sophisticated model is the hypercube like Figure 2 which is a 3 dimensional case. The beauty of hypercube structure is well-known as its logarithmic structure. The degree which is defined as the number of neighbors of a processor in a given network topology [1], and the diameter which is defined as the longest shortest path counting on number of links from any P_i to P_j [1] (i.e. the diameter of Figure 2 is 3 since the longest shortest path from one processor to another is 3.) are both *log(N)* where *N* is the number of processors in the hypercube. Interesting enough is that the dimension of the hypercube can be naturally expressed by binary labeling. Like at the Figure 2 case, where we have 8 processors which forms a 3 dimensional hypercube (*log8 = 3*). We can label each processor based on the way that each adjacent pair has hamming distance of one. The labeling basically explores the wonderful partition ability with the hypercube structure as we will see at the later example using hypercube. Furthermore, as Figure 3 shows, the hypercube can be transformed to different presented graphs. In 3 dimensional case, we can even get a planar graph representation of the hypercube.



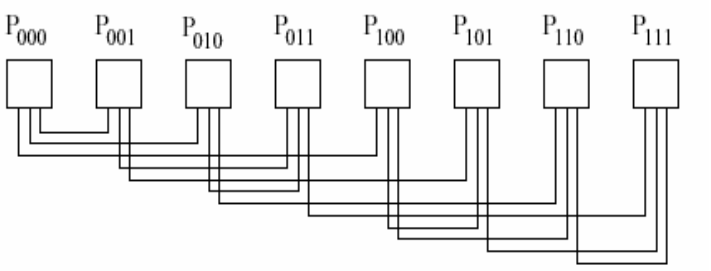


Figure 3: Alternative of figure 2. [1]

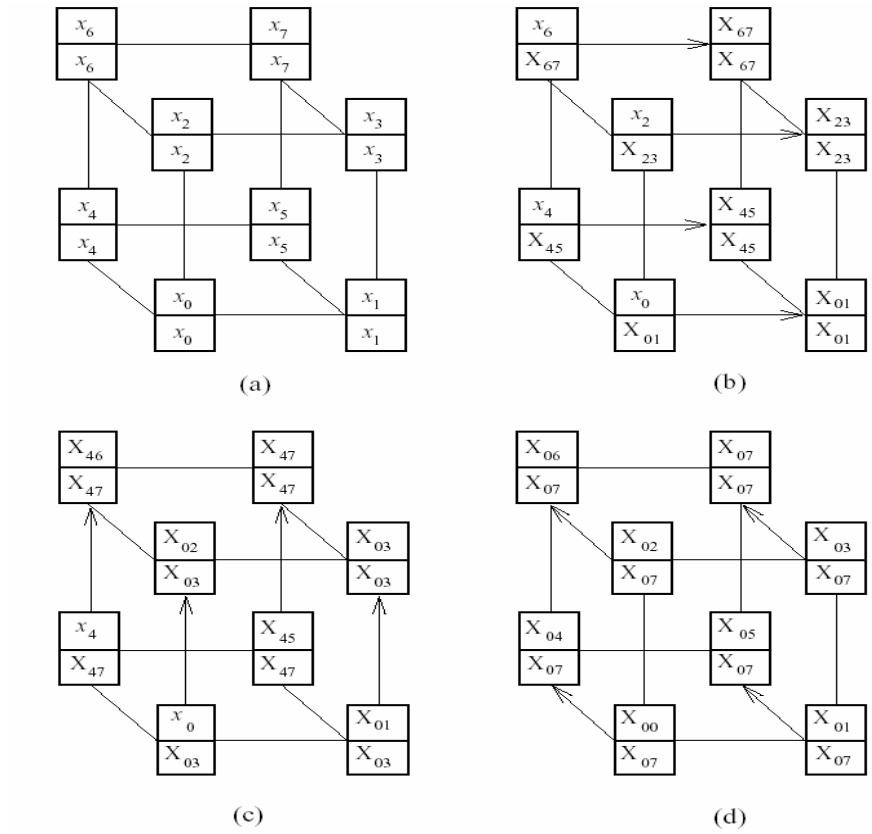
Figure 2: A hypercube interconnection network. [1]

A very common computing problem that has been efficiently solved by hypercube model is the prefix sum calculation. The prefix sum problem gives an

array of { $x_0, x_1, x_2, \ldots, x_n$ }, and asks to calculate the set of sums { $S_0, S_1, S_2, \ldots, S_n$ }, where $S_i = \sum x_k$ where (k from 0 to i). Akl's book [1] has presented a simple parallel algorithm assuming that each hypercube processor P_i has two registers A_i and B_i, where $0 \le i \le n-1$. The algorithm is stated as following:

```
for j = 0 to log(n) – 1 do
        for all i < i^(j) do in parallel
                (1) A_i^(j) ← A_i^(j) + B_i
                (2) B_i^(j) ← B_i^(j) + B_i
                (3) B_i ← B_i^(j)
        end for
end for.
```

The algorithm is illustrated in Figure 4 for $n = 8$, where A_i and B_i are represented as the top and bottom registers of P_i respectively, and X_{ij} is used to denote $x_i + x_i + \ldots + x_j$.



(a) Initially;

(b) j = 0;

(c) j = 1;

(d) j = 2;

Figure 4: Computing the prefix sums on a hypercube. [1]

Again, the algorithm manages to complete the calculation within *log(n)* time, where the sequential calculation time complexity is linear since the simple traversal takes *O(n)* time. When the hypercube calculations are finished, we will have the set of prefix sums at the *A* registers. Every *B* register will have the total sum of all values.

The above idea is quite simple. However, it does explore the structural advantage of the hypercube. The idea of partition a graph structure is well practiced in this algorithm. The parallel operations are all based on the separations of dimensions. Precisely, if we take the Figure 4 case, at each round, the parallel algorithm cuts the 3 dimensional hypercube to two 2 dimensional hypercubes based on different directions. And the cut can be simply done by recognizing the most significant bit at different location. The paper [5] which states a perfect load balancing on hypercube multiprocessors algorithm even extends this approach with a simple mathematic decision function, and achieves an amazing result to balance the job load to each processor on hypercube.

Limited by the length of this report, I can not keep demonstrating more parallel computing models such as mesh, star, mesh of tree, etc. But I think from the above two examples, we can get a basic taste of the relationship between the model and our graph theory. Most models using in parallel computing today have been already carefully studied from graph theory point of view even long time ago. I also found that it was quite efficient to understand a model by studying its related graph to get its attributes and features. In this sense, as I stated at the beginning of this section, graph theory is the base of parallel computing models we are talking about now. Of course, this relation is not just at the model level. More important, it

is a fascinating area from both sides when designing some parallel algorithms to solve those hard problems in graph theory.

Graph theory itself has been studied more than 200 years, many puzzles have been solved, and many still remain unsolved. Algorithmically speaking, we do get lots of NP-hard, NP-complete problems from graph theory such as Hamiltonian path, Travel Sales Man problem, Maximum Matching problem, etc. On the other hand, parallel computing was brought to us because computer scientists recognized that this would be a neat way to provide extra computing powers. I guess it would not be a surprise that people tackle on those hard graph problems with parallel computing. Actually, I find many interesting parallel graph algorithm papers have been published or being published at present. Academically, this is one of the hottest fields in algorithm designs. I selected the paper [3] which states an algorithm about parallel I/O scheduling using the edge coloring method as a case to explore this kind of graph problems represented in parallel world.

The problem in the paper [3] is already familiar with us, which is basically focused on how to schedule the CPU with I/O so that the contention can be eliminated while maintaining an efficient use of bandwidth. The idea in this paper is to deduce the CPUs and I/Os to a bipartite graph, and solve the edge-coloring and maximum matching for this graph. Of course, this only makes sense in parallel computing, since we have multiprocessors (CPUs). The authors managed to study *the highest degree first* (HDF) heuristic, and got pretty nice performance from their algorithms. The algorithm is really nothing new in terms of edge-coloring and maximum matching problems. However, the key point here is we have

multiprocessors in parallel, and the algorithm allows each processor calculates the maximum matching at the same time, in other words, doing distribution parallelly. I would like to stop here since the relation between graph theory and parallel computing is well examined at this point for this paper. Problems like this, such as Hamiltonian path which is NP-complete have been well studied in parallel. It is also not hard to see how we can efficient calculate Hamiltonian path using multiprocessors. Intuitively, if we have enough processors, we can use one processor calculate a path simultaneously. It is easy to see we will get the result pretty fast this way. Of course, many parallel Hamiltonian path algorithms are much trickier than above. We can not only measure a parallel algorithm by time complexity. As we will see at the following section, the measurement of parallel algorithm is somehow different from traditional algorithm. Obviously, this is also the reason that we keep inventing new algorithms for those hard graph problems.

I think up to now, I can conclude this section with the strong relations between graph theory and parallel computing, that have been explored by previous paragraphs. First of all, the lower level, or the base of parallel computing, which is the computing model, is directly related with graph theory, and well funded by graph theory. Parallel algorithm is built on top of parallel model, in other words, parallel algorithm is supported by graph theory. On the other hand, the most important feature of parallel computing is extra computing power provided by multiprocessors. It becomes so natural to examine new parallel algorithms for those graph problems since there are so many unsolved or time-consuming puzzles in graph theory field. Exactly, graph theory provides parallel computing

with basic model support, and parallel computing gives us new ideas to design graph algorithms dealing with those tough problems in graph field. In the later sections, we will see more such kind of relations, which can give us more inside look at this new algorithm field.

3 Basic Parallel Operations and Measurements

Before we can look at some exciting parallel graph algorithms, it is worth our time to settle down some basic parallel computing operations as well as some measurements. Like the sequential algorithm world, parallel algorithm does come with some physical limitations from the computer structures. Furthermore, parallel computing models are various as we have seen couple of them at the previous section. And it is not appropriate to refer a parallel algorithm without specify the computing model. This is where parallel algorithm design differs from sequential algorithm design the most. So, at the following, we will look at some very general assumptions that have been taken when designing parallel algorithms. Of course, the measurements of parallel algorithm are somewhat different from sequential algorithm as we can imagine for the same reasons.

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. However, the PRAM models are not all uniformed. With different memory access methods, PRAM models actually vary in four preliminary divisions. Firstly, we have basic two types of instructions as reading and writing. So for the reading, we will subdivide into *Exclusive Read* (ER) which provides processors to access memory with a

one-to-one fashion, and *Concurrent Read* (CR) which allows processors to read from a memory location simultaneously. Of course, for the writing, we have two types respectively, which are *Exclusive Write* (EW) and *Concurrent Write* (CW). Because of the purpose of this report, I will not get into more details about further divisions such as *Priority CW*, *Common CW*, etc. For your interests, please look at book [1] and [7] for comprehensive information. If you think back our examples at the Section 2, clearly, they are not just taking PRAM as the model, but more complex structures. The book [7] treats them as processor connected model algorithms which are different from PRAM model algorithms. As demonstrated from previous problems, taking those models such like tree and hypercube would give us many benefits in terms of algorithm design. One thing we have to keep in mind is that the communication between processors does take time, and sometimes, it can not be assumed as a constant. The memory access in some situation suffers the same problem that simply taking constant time can lead to a wrong time complexity analysis.

As an algorithm designer, the most important thing is to analyze the algorithm and understand the algorithm efficiency or time complexity. We have used the asymptotic system well-known as big “O” and big “Ω” for year. In parallel algorithm world, we keep using it to measure our time bound. However, we always like to think about optimal for problems such as sorting, we know the best we can do is *O(nlogn)* proved with sequential computing algorithms. But if we reverse our first example algorithm, and feed the values that we want to sort from the root processor, then divide into two part along the binary tree, we will get the sorted

array from the leaves in *O(logn)* time. Of course, the speedup in this case is *O(n)* from using this parallel algorithm. And the question is whether this is optimal in our parallel case since it already beats the sequential lower bound. This motivates us to look at new measurements at parallel world. The obvious measurement solution is the cost. The cost is defined as the time complexity multiples by the number of processors used. In the above case, the tree structure has $nx(n-1)/2$ which is $O(n^2)$ processors. So the cost is $O(logn)xO(n^2) = O(n^2logn)$, and not optimal at all. It is quite reasonable this way since we can not just add processors for nothing, and the number of processors is another important issue to consider with. For many graph problems, it is even critical to control the number of processors as graphs can get eventually very large and complicated. There are also other measurements like works, etc. A parallel algorithm analysis may be done in quite a different way. Sometimes the traditional way of analysis results a very inaccurate estimation. Many efforts have been put on analysis a parallel algorithm. The report will not cover those interesting topics from other measurement methods since the algorithms provided here are quite elegant for analyzing. I think that the best way we can explore new analysis ideas about parallel algorithms would be through reading those papers from various parallel algorithm publishes nowadays. It is no doubt the truth that analysis of an algorithm becomes harder and harder since the newly developed algorithms especially in parallel world become more and more complicated. The topic itself is a big area being studied.

We now have examined some basic operations and measurements which we need to consider for understanding our parallel algorithms. As I mentioned

above, those are very preliminary ones. More advanced ones are needed for different parallel algorithms which I do not include in this report. The essential two things we have to keep in mind from this section are the model dependency of a parallel algorithm, and new analysis measurements involved with parallel algorithms.

4 Parallel Classic Graph Algorithms

At the beginning of this report, I have already explored some parallel algorithms related with graph theory. Those algorithms are pretty elementary like sorting using tree model, prefix sum from hypercube model, etc. However, really graph problems have also be extensively studied in parallel, and those parallel graph algorithms are not as simple as above ones. More issues like partition, representation, etc have to be considered careful. In this section, we will examine some classic graph problems in parallel.

It is really easy to pick up some parallel algorithms from many publishes. However, I feel many of them are more concentrated on their parallel part, which is to say that the articles and papers are really focused on parallel techniques while using the graph problems as their media. However, it makes more sense to find something really related with graph theory more here. So I guess this is the motivation that I choose the shortest path problem from book [7]. The book is no doubt tackling on graph problems as it is a graph theory book. And the shortest path problem (SP) is also a well-known graph problem captured so many big names like Dijkstra, Floyd-Warshall, Bellman-Ford, and so on. For all those reasons, I think this problem would be a perfect choice to put in this section. Two traditional SP algorithms as Dijkstra and Floyd will be discussed in parallel at the following part. We will also look at the analysis of these two parallel algorithms.

Dijkstra's SP algorithm is probably one of most widely used algorithms not only in graph theory, but also some other practical fields like networking, database,

and so on. Dijkstra's algorithm is essentially a greedy algorithm which always chooses the lightest or closest vertex in the given graph. And the sequential version runs $O(|V|^2)$ time generally speaking, where $|V|$ is the number of vertices. The parallel version given by book [7] introduces an $O(|V| \log |P|)$ algorithm where P is the processors. This parallel algorithm takes PRAM EREW model. In order to see the parallel algorithm, we can first find out the performance bottlenecks in sequential version algorithm. Clearly, identifying the next vertex to include in the shortest path tree, where the vertex is the search tree vertex with the smallest estimated distance from the start vertex, and which is not yet in the shortest path tree, costs most time of the algorithm. And of course, after identifying the next vertex, we have to update each vertex distance to others in the whole graph. It is not hard to find out the possible parallel operations here. With the divide and conquer idea in mind, we can organize the required computations in the fashion of a binary tree. At the leaf level of the tree, we group the values of the distances from one vertex to another into $|V|/2$ pairs, and find the minimum of each of the pairs using $|V|/2$ processors which is constant time operation. Then group the resulting $|V|/2$ minima into $|V|/4$ new pairs and repeat the process again. Without loss of generality, we can assume $|V|$ is some power of 2, then after $\log (|P|)$ stages, we have found the minimum distance. Now, next step is to update the shortest distances for each pairs. We can again use the binary tree to broadcast the distance from one processor to others. The process works that one processor tells another processor, and then they each tell two other processors, and so on. Obviously, this task can be done linear time respected to the size of the graph

which is $|V|$. So the total algorithm will take $O(|V|log|V|)$ combining the two dominant steps mentioned above. The book [7] also mentions the situation where we do not have enough processors. At this situation $|P| < |V|/2$, and we can put $|V|/|P|$ vertices into each processor in stead of just a pair. So the minimum distance part will take $O(|V|/|P| + log|P|)$, and the broadcasting will take $O(|V|/|P| + log|P|)$, which are some kind of input sensitive. The EREW provided by the model assures that when calculating the distance concurrently as well as the updating the distances, each vertex from the graph is only accessed by one processor at each stage. This is very important in terms of the correctness of this algorithm. The cost for this parallel algorithm is $O(|V|^2log|V|)$ which is no better than the $O(|V|^2)$ time algorithm sequentially. The standard trick is to use $O(|P| / log|V|)$ processors to hit the sequential bound if it is measured by the cost of parallel algorithm. Because the additional data that a processor gets in this case will be relative small comparing with the whole process, the sequential manner additional calculation in each processor will have minor effect towards time complexity, or asymptotically no increase of the total run time. So, eventually, we can make the cost of this algorithm $O(|V|^2)$ to match the sequential time complexity.

Another well-known algorithm for SP problem is the Floyd SP algorithm. One of its parallel versions is also presented by book [7]. I include it here as an example comparing with above Parallel Dijkstra's SP algorithm. The Floyd SP algorithm is based on the distance matrix. Through manipulating the matrix, the algorithm identifies the paths for each pair vertices. We will denote the *shortest distance matrix* as SD at the following description. Actually, matrix manipulations

are natural to be thought in parallel. So it is relatively easy to bring Floyd SP algorithm to parallel. Notice that every entry in SD that can change at stage k depends only on its current value and the values of a pair of components in the k^{th} row and column. The components of SD may be updated in parallel because each entry at a stage is sort of independent to others. We only need to make sure only one processor can write to certain memory location at each stage. EW as we specified as the model of this algorithm ensures the one to one writing at each stage. And CR is needed, because the components of the k^{th} row and column have to be read concurrently in order to utilize the multiprocessors. Each stage now only needs constant time to be calculated since each entry is updated concurrently. Finally, the algorithm needs to run through all vertices which takes $O(|V|)$ time. In this case, we ask for $|P| = |V|^2$, if we do not have enough processors, we can apply the same technique as mentioned by the end of previous algorithm. However, if the number of processors is significant less than what we need, the sequential process in each processor can not be ignored asymptotically. The time complexity of the parallel algorithm will be estimated as $O(|V|^3/|P|)$.

Parallel Dijkstra's SP algorithm and parallel Floyd SP algorithm are two parallel algorithms tackle on the SP problem in graph theory brought from their sequential versions. The idea is similar to examine the sequential algorithms and figure out where in the algorithms can be paralleled, in other words, the jobs can be executed simultaneously without disturbing each other. In parallel Dijkstra's algorithm, the shortest distances from one vertex to others at a stage can be calculated independently, and in parallel Floyd algorithm, the SD entries can also

be updated concurrently with exclusive write control. The speedup of the parallel algorithm is indeed brought to us by those parallel abilities. Essentially, these two algorithms are easy to understand because the minor changes from their sequential versions. However, this is not always the case, even with classic graph problems. A good example to look at is the Hamiltonian problem. Many parallel algorithms have been presented dealing with Hamiltonian problem. If we only look at the time complexity, we can check a graph in constant time assuming we have enough processors one for each vertex. The most naïve algorithm is almost the same as sequential and speedup comes purely from the multiprocessors plugged in. Of course, as we have seen from previous sections, the parallel algorithm has many unique techniques with parallel models. So it is not hard to find many improvements from papers presenting parallel Hamiltonian algorithms. Unfortunately, Hamiltonian problem is still NP-complete even in parallel world. It is generally believed that if we can find a parallel algorithm with a polynomial cost, then it is possible to find a sequential algorithm with a polynomial time complexity dealing with the same problem by simulating the parallel one. In this sense, many classic NP-complete graph problems stay unsolved in parallel computing. Of course, for those problems, parallel algorithms vary a lot with sequential algorithms. I decide not to give such example in this report simply because describing one instance using totally new approach will take another entire report. But as we have noticed, it is obviously true that classic graph problems in parallel are generally much harder to be solved than the two examples we have seen above.

5 Parallel Graph Algorithm Applications

The parallel graph algorithm application is really an implicit concept to me. Graph theory as a widely used field has been applied to so many areas. Even some very theoretic areas like computational geometry as one good example, can be called as an “application” of graph theory. Of course, for the sake of the meaning of application, a theoretic area can not be application rigidly speaking. So, the two problems I prepared for this section which are one geometry problem and one marriage problem are somehow unfortunately fall into the theoretic “application”. However, for the faith of algorithmic interest, I still would like to include them, because I found them rather interesting to me.

The first problem is a graph problem with some geometric meaning, which was given by Professor David Rapapport (Faculty of School of Computing, Queen’s University), and brought to parallel by Professor Selim Akl (Faculty of School of Computing, Queen’s University). It is a very elegant example to demonstrate the power of parallel algorithms giving us an amazing computational speed up. The problem simply asks for a transformation from the (a) in Figure 6 to (b). The rule for transformation is that at each stage, we can move and add edges while maintaining a triangulation of the rectangle inside, as we can see the initial graph (a) is triangulated inside. Geometrically, the final goal of the transformation is to reverse the triangulation direction putting “0 fan” marked in (a) from upward to downward, and “1 fan” from downward to upward. In other sense, this can also be considered as moving each upward triangle to one downward triangle location. So,

if we have a sequential algorithm to do this, the algorithm will need to move each triangle *n/2* steps forward or backward, and half triangles have to be move to achieve the transform which is *n/2*, where *n* is the total number of triangles we have. Clearly, we need roughly $n^2/4$ steps. In the Figure 6 case, $n = 6$, we need $6^2/4 = 9$ steps.

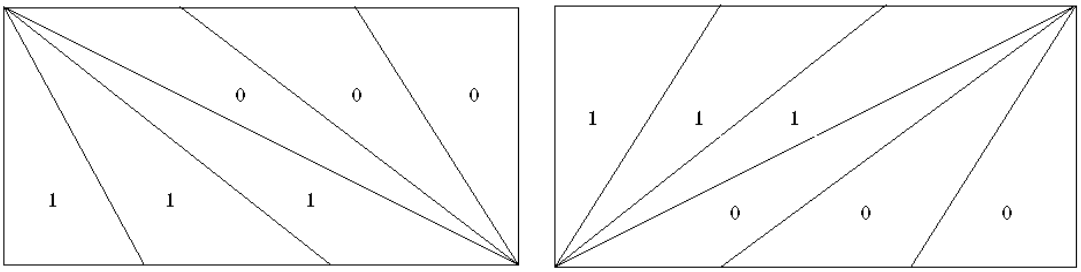


Figure 6: question demonstration. (a) Initial stage. (b) Finial stage.

Figure 7 below shows the sequential algorithm to process this transform. The first step is certain that we have to move the diagonal and add an edge like (a) in Figure 7 represented by a dot line. Then (b) part demonstrates the step two to five which move the remaining edges of two sides and replace with those dot edges. Step six from part (c) has to remove the central dot line and replace it with a forward dash line. The seventh to eighth steps are used to establish the two sides' triangles. Final step is just to replace the central dash line with a diagonal line to finish the transform. Exactly, there are nine steps as we stated at above paragraph. If we count triangles as input, we will have a time complexity of $O(n^2)$ in this case with sequential algorithm. Furthermore, there is no way to speedup at sequential manner, since we have to move and replace an edge a time without losing any triangle, in other words, half triangles have to be moved step by step with the total

number of at least *n/2* each.

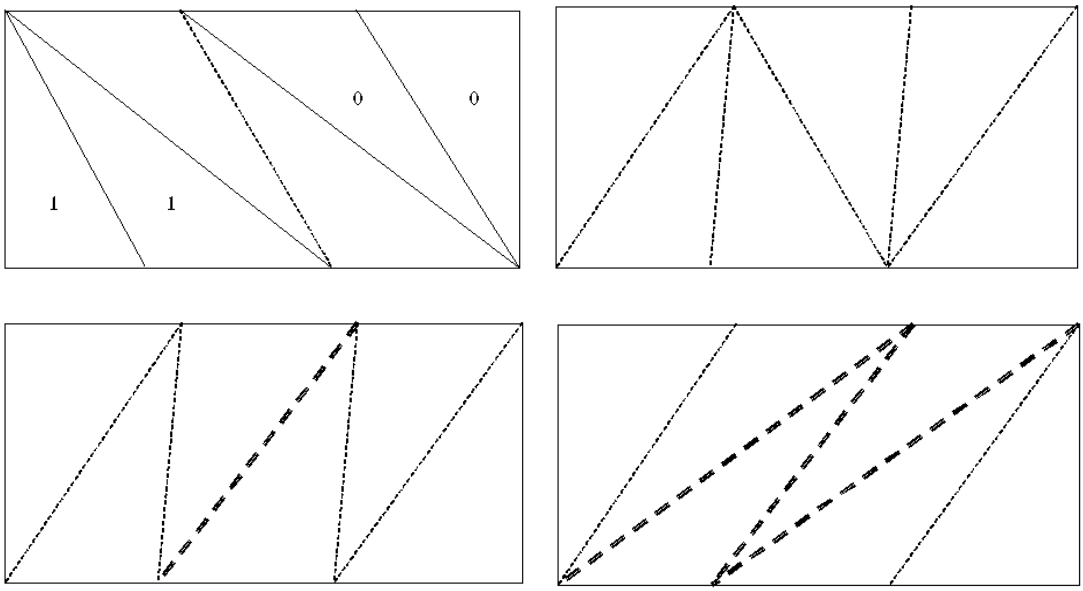


Figure 7: transformation steps. (a) First replace step. (b) Second to fifth steps (c) Sixth step. (d) Seventh and eighth steps.

In order to compare with sequential algorithm, we would like to see how parallel algorithm can accelerate our transform, and more important, how good it can do. In this case, the parallel algorithm leads a simple constant time algorithm. It is quite easy to imagine the algorithm just thinking that we have enough hands to grab all triangles up and restore them once. With n processors, we can put each triangle to a processor simultaneously, and restore all triangles at constant time without violating the rules. The cost here is *O(n) (c(n) = O(1)xO(n))*, and speedup is $O(n^2)$ comparing with $O(n^2)$ time complexity from sequential. It happens that this is also an example where the classic speedup theorem, which states the speedup can not go over the ratio of processors used, fails, since we only use *O(n)* processors, and

we get the speedup of $O(n^2)$. I think this is a very impressive problem where a simple geometric graph problem can be resolved efficiently with parallel algorithms.

The second example for application at this section is the “old” marriage problem in graph theory. Of course, it is also extensive studied and widely used. The parallel algorithm we will look at later is a relatively new development from paper [6]. The algorithm is not as understandable as the previous example. We have briefly looked at the problem and some solutions from previous works in class. However, for the sake of completeness, I will restate the problem here, but ignore the previous sequential solutions and studies about this problem. The problem is also called stable marriage problem, which was first introduced by Gale and Shapley. Given *n* men, *n* women, and *2n* ranking lists in which each person ranks all members of the opposite sex in order of preference, a *matching* is a set of *n* pairs of man and woman with each person in exactly one pair. A matching is *unstable* if there are two persons who are not matched with each other, and each of whom strictly prefers the other to his/her partner in the matching; otherwise, the matching is stable. Gale and Shapley showed that every instance of the stable matching problem admits at least one stable matching and such a matching can be computed in $O(n^2)$ iterations. The paper [6] propose a new approach, *parallel iterative improvement* (PII), to solving the stable matching problem. The PII algorithm basically consists of two alternating phases. The first phase is a procedure that randomly generates a matching, and the second phase consists of multiple improvement iterations. The parallelism is explored as identifying a subset

of unmatched pairs to replace matched pairs for an existing matching so that the number of unstable pairs in newly obtained matching can be reduced. The authors also managed to design this algorithm for various parallel models. They found that both phases took *O(logn)* time on completely connected multiprocessor system and array with multiple broadcasting buses, which are very naïve models, however, not practical at all, and $O(\log^2 n)$ time on both hypercube and MOT (mesh of tree). Let's look at this PII algorithm in more details at following.

Let M = { m_1, m_2, …, m_n } and W = { w_1, w_2, …, w_n } be the sets of *n* men and *n* women respectively like the paper [6] specified. Let $mL_i = \{ wr_{i,1}, wr_{i,2}, \ldots, wr_{i,n} \}$ and $wL_i = \{ mr_{i,1}, mr_{i,2}, \ldots, mr_{i,n} \}$ be the *ranking lists* for man m_i and woman w_i, where $wr_{i,j}$ (resp. $mr_{i,j}$) is the rank of woman w_j (resp. man *mj*) by man m_i (resp. woman w_i). Then, we can get a *ranking matrix* of size *n x n*. The below Example 1 comes from the paper which will make our first construction clear.

Man ranking lists:	Woman ranking lists:	Ranking matrix:
$mL_1 : \{4, 2, 3, 1\};$	$wL_1 : \{1, 4, 2, 3\};$	$4,1 \quad 2,1 \quad 3,4 \quad 1,3$
$mL_2 : \{3, 1, 2, 4\};$	$wL_2 : \{1, 2, 3, 4\};$	$3,4 \quad 1,2 \quad 2,2 \quad 4,1$
$mL_3 : \{2, 4, 1, 3\};$	$wL_3 : \{4, 2, 3, 1\};$	$2,2 \quad 4,3 \quad 1,3 \quad 3,4$
$mL_4 : \{1, 4, 3, 2\}.$	$wL_4 : \{3, 1, 4, 2\}.$	$1,3 \quad 4,4 \quad 3,1 \quad 2,2$

Example 1: an instance of ranking matrix. [6]

The two phases of the PII algorithm no doubt will be set up working with this ranking matrix to find out a stable matching. It is easy to say that a matching is stable if and only if there is no unstable pair in the selection set from the matrix. The paper [6] identifies two types of pairs in the process. A set NM_1 of *type-1 new matching pairs* (simply nm_1*-pairs*) is defined as follows. If there is no unstable pair,

NM_1 = *null*. Otherwise, for every row R_i with at least one unstable pair, select the one with the minimum left value among all unstable pairs as an *nm_1-generating* pair; for every column C_j with at least one *nm_1-generating* pair, select the one with the minimum right value as an *nm_1-pair*. Based on NM_1, a set NM_2 of *type-2 matching pairs* (simply *nm_2-pairs*) can be found by a procedure that first identifies *nm_2-generating* pairs and then identifies *nm_2-pairs* using an *nm_2-generating* graph. For any *nm_1-pair* $a_{i,j}$ in the set, pair $a_{l,k}$ with *l* being the pair at the same column and *k* being the pair at the same row is called the *nm_2-generating* pair corresponding to $a_{i,j}$. If we choose any set (i.e. any matching) from the matrix, the degree of *nm_2-generating* graph is at most 2. Then we can swap the original unstable pairs out, and replace them with *nm-pairs* as we described above. The paper [6] states that if let $NM = NM_1 \cup MN_2$, and *RM* be the set of $a_{i,j}$ which is going to be replaced, then we can get the stable matching by *(M – RM) U NM*, where *M* is the random matching set from the first phase.

Based on this idea, the algorithm first picks up a matching randomly. The basic technique uses to find a matching is through pointer jumping. A pair of processors swap their pointers with each other, and *O(logn)* time to find a matching since the length of the list is n. The second phase is to identify the *NM1* and *NM2* which have been defined above. To find *NM1* through checking each row and each column, we have to find the minimum left or right value, which takes *O(logn)* time. All rows and columns can be checked concrrently. To find *NM2*, because the *NM2* graph has a degree at most 2, parallel computing can do this in constant time. After we find out *NM1* and *NM2*, we can use *(M – RM) U NM* to calculate the new matching. Again,

it will only take *O(1)* time. Ideally, because the algorithm identifies the parallelism of calculating *NM1* and *NM2*, the total time complexity stays *O(logn)*. We have seen at Section 2 that parallel algorithms are model dependent. So the paper tries to implement this PII algorithm with different possible models. The Figure 8 is the representations of the models that have been chosen from the paper. It terms out the time complexity which has been specified at the previous paragraph is model dependent too. The array with multiple broadcasting buses is an ideal model to use here which achieves *O(logn)* bound. However, the hypercube and the MOT both increase the time complexity to $O(\log^2 n)$. The main explanation for this is that the communication or broadcasting at a hypercube or a MOT take *O(logn)* time, but only constant time through buses. Buses used here are generally treated as *O(1)* time for each read or write operation. And each processor can send or broadcast data to any processor in the array within constant time. And in this PII algorithm, at phase two, we need to find the *NM2* through checking *NM1*, which is some kind of broadcasting operation. When hypercube or MOT is taken into consideration, we can not assume finding *NM2* is an *O(1)* time operation, instead, it has to take *O(logn)* time.

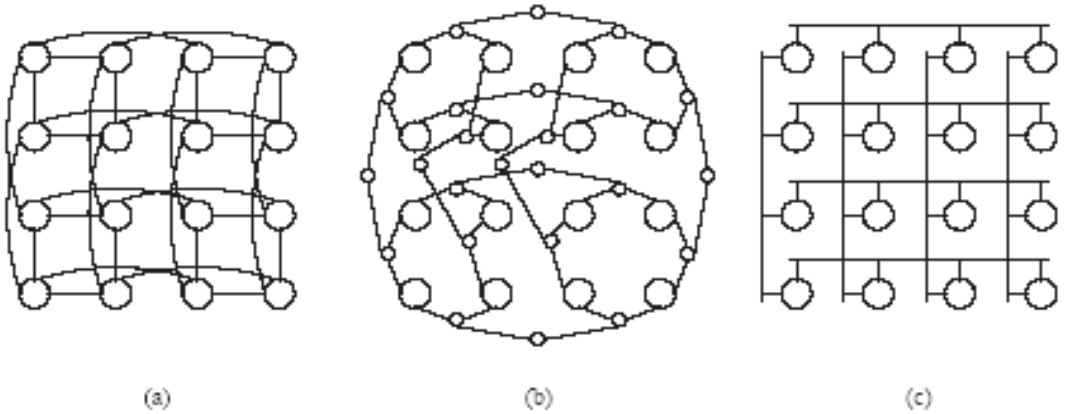


Figure 8: (a) a 16-processor hypercube (b) a 4x4 mesh of trees (c) a 4x4 array with multiple broadcasting buses.

We have looked at two recent parallel graph algorithms. Although it is only an overview, please refer back to the original publishes for details, we can still experience the difference on how parallel algorithms are to be constructed and analyzed. Many new issues have been taken into consideration. From graph theory side, we have seen the power of using parallel ideas to solve some graph problems either new or traditional. The implementations of those parallel graph algorithms also can be very useful in practice. Like the PII algorithm for marriage problem can be applied to scheduling algorithm in order to provide QoS guarantees suggested by paper [6]. Of course, parallel algorithms applied to graph problems are always complicated nowadays. It will be a very challenging field in the future I believe.

6 Summary

I feel this report still a very basic summary of the works have been done so far. Limited by the size of this report, I have to give up many interesting parallel graph algorithms. Book [1] covers very detail about parallel computing, which happens to be a good textbook to understand the parallel world. Book [7] is the only book in graph theory I found with parallel algorithms. The parallel algorithm chapter in this book is definitely a good start to get some idea about parallel graph algorithm. Of course, one may suffer from the out of date problem reading some algorithms in the books. I think this is the main reason I included two relative new examples at Section 5. Since the parallel algorithm is changing so fast, the best way to get around this would be reading papers from recent publishes. It is quite beneficial to me as I actually formed this topic from the paper reading.

My expectation with writing this paper is to explore a new way of thinking dealing with algorithmic graph problems in parallel computing. I consider the observation of a graph problem from a different angle to be more interesting than the algorithm presented here. Parallel is a very power idea, and expected to reshape our traditional computational world. In this sense, looking at our traditional graph problem from a new perspective would be rather elegant. If you feel enjoyable with some new thinking from this report, it would be the best reward for me to have it.

References

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

[2] R. Diestel, Graph Theory: Electronic Edition 2000, Springer-Verlag, New York, 1997, 2000.

[3] 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.

[4] L. Euler, Solutio problematis ad geometriam situs pertinentis, Commetarii Academiae Scientiarum Imperialis Petropolitanae 8, 128-140, 1736.

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

[6] E. Lu and S. Zheng, A Parallel Iterative Improvement Stable Matching Algorithm, Proceedings of International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science, Springer-Verlag, 2003.

[7] J. McHugh, Algorithmic Graph Theory, Prentice Hall Inc., New Jersey, 1990.

[8] J. Rourke, Computational Geometry in C: second edition, Cambridge University Press, New York, 1998.