UNIVERSITY OF BERGEN

A Comparison of Vertex and Edge Partitioning Approaches for Parallel Maximal Matching

by

Alexander N. Sørnes

Thesis for the degree Master of Science
December 2013

in the
Faculty of Mathematics and Natural Sciences
Department of Informatics

Abstract

This thesis will compare two ways of distributing data for parallel graph algorithms: vertex and edge partitioning, using a distributed memory system.

Previous studies on the parallelization of graphs has often been focused on a vertex partitioning, where each processor is assigned a set V' $\subseteq$ V where G = (V,E), yielding a one-dimensional partitioning. It has been shown, however, that an edge partitioning (or 2D partitioning), where each processor is assigned a set E' $\subseteq$ E, may yield a benefit in terms of a lower communication volume.

The performance and scalability of vertex and edge partitionings are compared by implementing the Karp-Sipser matching set algorithm for both partitioning schemes. A matching set is a set E' $\subseteq$ E of independent edges such that each vertex in V occurs at most once in E'.

We find that while the vertex partitioned algorithm gives a significantly higher speedup, the increased performance of the edge partitioned algorithm on more dense graphs suggests that if the graph framework is improved further, it could lead to the implementation of an edge partitioned matching algorithm that offers better scalability and comparable matching quality to a vertex partitioned matching algorithm.

An edge partitioning requires a rigorous framework for handling the communication resulting when edges owned by multiple processors are incident on the same vertex. Hopefully, the framework developed for representing an edge partitioned graph facilitates the implementation of other parallel graph algorithms using an edge partitioning approach.

Acknowledgements

I would like to thank my supervisor, Prof. Dr. Fredrik Manne, for providing wise counsel and encouragement throughout my work with this thesis.

Furthermore, I would like to thank my parents for teaching me the value of knowledge.

(On a slightly less serious note, I would finally like to thank the inventor of the microwave oven, without which I would surely have starved while confined to the study hall.)

Contents

Chapter 1

Introduction

Suppose that you are in Berlin, and you need to know the cheapest route by train to Warsaw. There is one rail link via Prague, and another one that goes through Dresden, and Dresden and Prague are directly linked as well. This may be expressed as a graph where the cities are *vertices* and the rail links are *edges*, as shown in Figure 1.1. These edges may have *weights*, which in this case would be the ticket prices, but as this thesis is concerned with *unweighted graphs*, let all ticket prices be equal.

Having expressed the problem in graph terms, it may now be solved using a shortest path *algorithm.* Although this particular problem is trivial, the same method could be used if there were scores of stations and hundreds of rail links - or indeed millions, as is the case of one considers a network of e.g. all the airports in the world, with one edge per direct flight connection.

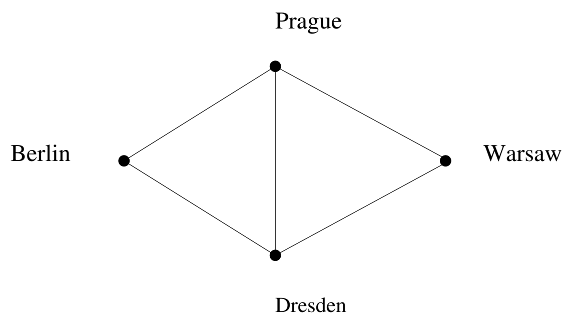

Figure 1.1: Graph of the train network; vertices are shown as circles and edges as lines

Indeed, graphs and the algorithms that may be run on them are so powerful and versatile that they may be used to express relations in such large quantities of information that it would be impossible to handle by humans. An example of this is Facebook: with one vertex per user - of which there are more than one billion - and an edge between any two friends, the resulting *friend graph* may be used to extract interesting information, such as identifying *cliques*, large groups of people in which everyone knows each other.

The same technique may be used for web pages: with one vertex per site, and edges between any two linked sites, analyzing the resulting graph may yield some interesting information. A website mostly linking to press agencies may be automatically identified as a news site, whereas one may deduce that a website linking mostly to sites about animal welfare, would be suitable for pet lovers.

What do these examples have in common? The average number of direct plane routes per airport, compared to the number of airports; or the average friend count per individual, compared to the number of human beings; or the average link count for a website, compared to the total number of such sites on the web - in graph terms, the average *edge count* per *vertex*, or average *vertex degree*, is quite small. Such graphs are called *sparse*, and it is their prevalence in real world problems that make them so interesting.

Another graph problem, and indeed the main topic of this thesis, is that of *graph matching*. A *matching set* is a subset of a graph's edges such that no two edges are adjacent, i.e. each endpoint in the matching set occurs at most once. Say that you have been put in charge of a group of high school drop-outs in order to assign them to new schools for the next year, but the number of students that may be accomodated in each school is limited. How should one express this as a graph problem?

This may be formulated as a graph with one vertex per student, and one vertex per vacant school position, thus there may be several vertices per school. Now, place an edge between each student and the schools offering study programmes he or she is qualified for. The answer to the question "what is the student-school assignment(s) that will fill as many positions as possible?" is that of a *maximum matching* in the graph that was constructed. A *maximal* matching, on the other hand, is a matching that cannot be extended by adding another edge from the graph. Thus a maximum matching is also a maximal matching, but a maximal matching is not necessarily a maximum matching.

The graph used to describe the student-school matching problem actually belongs to a special class of graphs known as *bipartite graphs*, and the corresponding matching problem is known as *bipartite matching*. In such a case, the graph vertices may be divided into two sets - in our case students and schools - and the only edges that occur have an endpoint in each of the sets.

Bipartite graphs are not limited to assigning students to schools, or prospective parents for adopted children, for that matter. In the field of computer vision, a central problem is that of object recognition - deciding whether an object perceived by a set of sensors matches a corresponding object in a pre-computed database. If each object is described by a graph, the problem of recognizing objects becomes that of *graph isomorphism* (sometimes called *graph matching*); deciding whether two graph are identical. Shokoufandeh et al [11] describes a method in which the graph of an observed object is compared to that of a database object by computing a matching of the bipartite graph obtained by combining the two.

In real-world scenarios, obtaining the best possible solution is not necessarily the only requirement; another might be that the solution has to be computable within a reasonable amount of time. A *heuristic* might guarantee a much lower running time than an exact algorithm, but at the expence that the solution it finds is not necessarily optimal. The Karp-Sipser algorithm is a heuristic procedure for the *maximal* matching problem discussed earlier. It works in a series of steps, where in each step one edge is added based on simple rules and randomization.

How does one solve the problem even faster? A tempting answer is to simply run the implemented algorithm on a faster computer; however, the speed of today's processors touches a barrier, a fundamental limitation in our current understanding of physics: the speed at which information can travel. A modern processor will have a clock frequency in the order of a few GHz, which translates into 10^9 operations per second, i.e. each operation takes about $10^{-9}s$. Most of these operations (adding numbers, multiplication etc.) requires input in the form of numbers stored in system memory, or RAM.

Higher processor speeds become irrelevant if a processor spends most of its time waiting for information. Instead of relying on doing computations faster on a single machine, another approach is to go *parallel* - dividing a problem into several smaller parts, and solving them simultaneously with multiple processors.

Parallelization raises another question: how does one divide the problem? This thesis examines two approaches for graph distribution, by parallelizing the Karp-Sipser matching algorithm: the graph may be divided by assigning each process a set of the graph's *vertices*, or by assigning a set of the graph's *edges*, sometimes referred to as a 1D or 2D partitioning, respectively.

The rest of this thesis is structured as follows: in Chapter 2, the matching problem is defined formally and discussed in further detail, by designing an intuitive algorithm for the special case of *bipartite matching*; Chapter 3 presents a heuristic for *maximal matching* in general graph, and details the Karp-Sipser algorithm's advantages in terms of solution *quality*; Chapter 4 begins by presenting some extra motivation for why parallelization is important, before presenting concepts and terminology that are necessary for the discussion and analysis of parallel algorithms; while Chapter 5 examines previous work regarding parallelization of matching algorithms.

The first attempt at parallelizing the Karp-Sipser algorithm is presented in Chapter 6, where a *vertex* partitioning is used. This is contrasted with the implementation of the Karp-Sipser heuristic using *edge* partitioning in Chapter 7.

Finally, Chapter 8 provides experimental results and a comparison between the two approaches.

Chapter 2

Matching

Before discussing the Karp-Sipser matching algorithm, the matching problem will be examined more closely. First, the problem will be defined formally, as well as related terminology and notation. A special case of matching in bipartite graphs will be used to present a first matching algorithm, before the case for general graphs will be discussed.

2.1 Definitions

An unweighted graph G is defined by its vertex and edge sets, $G = (V, E)$. The vertices of a particular graph G are referred to as $V(G)$, and the edges as $E(G)$. An edge, say e, is defined by its two endpoints, $e = (u, v)$, where u and v are vertices, $u, v \in V(G)$. Unless otherwise stated, any graph in question is an *undirected graph*: given vertices u and v, (u, v) and (v, u) refer to the same edge.

A matching M of a graph G is a set $M \subseteq E(G)$ such that for any edge $e = (u, v) \in M$, no edge in $M \setminus \{e\}$ contains u nor v as an endpoint, or more formally, for any $e = (u, v) \in M$ and for any $x \in V(G), (u, x), (v, x) \notin \{M \setminus \{e\}\}$.

Recall that a maximal matching was defined as a matching that could not be extended by adding a new edge. Formally, a matching M is maximal if for any $(u, v) \in E(G) \setminus M$ there exists some $x \in V(G)$, such that either $(u, x) \in M$ or $(v, x) \in M$.

Two edges are said to be *adjacent* if they share a common endpoint; thus the edges $(u, v), (v, w)$ are adjacent, whereas $(u, v), (w, x)$ are not. A *path* is an ordered set of adjacent edges such that no edge occurs more than once, with distinct start and end points. A *cycle* is an ordered set of adjacent edges, with no edge occurring more than once, without a unique start and end point. Thus $(u, v), (v, x), (x, u)$ is a cycle, whereas $(u, v), (v, x)$ is a path.

2.1.1 Bipartite graphs

In a biparitte graph, the vertices may be divided into two sets, where there are no edges between the vertices of each set. The set of graph vertices may thus be described as a union of the sets A and B, $V(G) = A \cup B, A \cap B = \emptyset$, with the property that for any $(u, v) \in E(G)$, either $u \in A, v \in B$ or $u \in B, v \in A$. An example is shown in Figure 2.1.

The *bipartite matching* problem is a special case of the mathcing problem, where the input graph is bipartite.

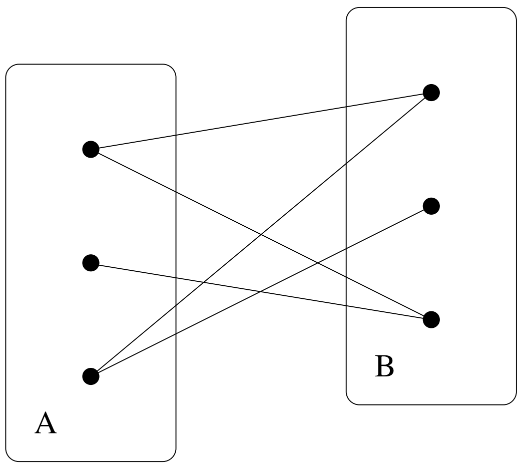

Figure 2.1: A bipartite graph with $V(G) = A \cup B$. Notice that all edges have one endpoint in A and one in B.

2.2 Bipartite matching

Central to the bipartite matching problem are the definitions of an *alternating path* and *augmenting path*. Given a graph $G = (V, E)$ and a matching M, an *alternating path* is an ordered set of adjacent edges such that every edge not in M is followed by an edge in M, and vice versa. An *augmenting path* is an alternating path that starts and ends with an unmatched edge; an example is shown in Figure 2.2.

Looking at the figure, it is obvious that the matching can be extended by removing the blue (matched) edges from it and replacing them with the unmatched (red) edges. In fact, *Berge's lemma* [12] states that a matching in a graph is maximum if and only if there is no such augmenting path. Based on this lemma, a simple algorithm for finding maximum bipartite matchings is to repeatedly find an augmenting path, remove its even edges from the matching and replace them with the path's odd edges, until there are no augmenting paths left (Algorithm 1).

Algorithm 1 A simple algorithm for a finding maximum matching in a bipartite graph

```
procedure BIPARTITE MATCHING(Graph $G$)
    $M := \emptyset$
    while $G$ has an augmenting path $P = e_1 \cup e_2 ... \cup e_n$ do
        $M := M \setminus \{e_2 \cup e_4 ... \cup e_{n-1}\}$
        $M := M \cup \{e_1 \cup e_3 ... \cup e_n\}$
    end while
end procedure
```

How many iterations of the while loop will be executed? Let $|V|$ denote the number of vertices and $|E|$ the number of edges in the graph. A vertex can be included in a matching at most once, and since each edge is represented by two vertices, a maximum matching contains at most $|V|/2$ edges. This is the case if all vertices are matched, and is referred to as a *perfect matching*. As each iteration of the loop increases the matching size by one edge, it will be run at most $|V|/2$ times. Finding an augmenting

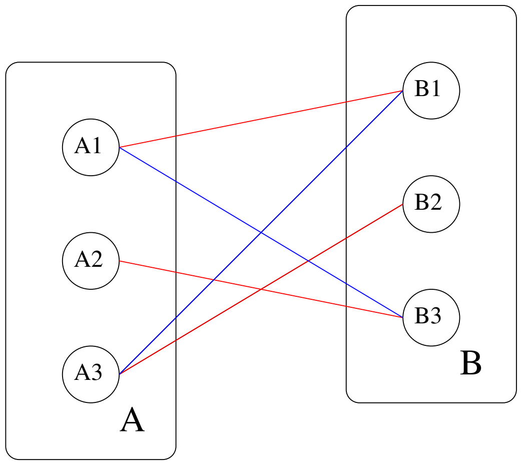

Figure 2.2: An augmenting path starting at A2 and ending at B2. Edges in M are shown in blue, unmatched edges in red.

path may be done in O($|E|$) time [12], giving a total running time of O($|V||E|$). The procedure for finding an augmenting path in a graph is as follows:

Let M_1 be the current matching and M_2 be another matching. Construct a graph G_1 consisting of the edges that belong to M_1 or M_2, but not both. G_1 will now consist of connected components, each of which is either a cycle of even length or a path, where every edge in M_1 is followed by an edge in M_2, and vice versa. If a path has odd length and begins with a vertex in M_2, then it is an augmenting path.

Hopcroft and Karp's algorithm [13] devised an improved bipartite matching algorithm, with a bound of O($\sqrt{|V|}|E|$) by exploiting the fact that Berge's lemma allows for the finding of several augmenting paths in each step, as shown in the procedure above. They start with a matching $M_0 = \emptyset$. In each step, they then find a set of j vertex-disjoint augmenting paths of the smallest possible, and equal, size. In a step $i > 0$, M_i is then formed by augmenting M_{i-1} with the j shortest augmenting paths found. By showing that the sequence $M_0, M_1, M_2...$ has at most $2\sqrt{s} + 2$ elements, where $s \leq |V|/2$ is the size of a maximum matching for the given graph, the improved bound is obtained.

In the paper they also suggest that a similar bound could be extended to general graphs; as will be discussed further in the following section, this is indeed true. The algorithm by Hoproft and Karp for biparitte matching was the fastest known (for sparse graphs) until Mucha et al.'s work [9] on randomized algorithms using fast matrix multiplication. As will be explained in the next section, however, this improved bound does not necessarily translate into faster running times in practice. For dense bipartite graphs, Alt et al. [20] showed how a version of the Hopcroft-Karp algorithm could be implemented to run in time O($|V|^{1.5}\sqrt{|E|/log|V|}$).

Yet another approach for a bipartite matching algorithm is to look at the problem as a special case of *network flow.* Given an undirected graph $G = (A \cup B, E)$, create a directed graph G' as follows: add a *source* vertex s with directed edges towards all vertices in A, add a *sink* vertex t with a directed edge from every vertex of B to t. Finally, for every edge in $E(G)$, let $E(G')$ contain a corresponding edge directed from a vertex in A to a vertex in B. Assign a unit capacity to each vertex.

Expressing bipartite matching as a special case of network flow is especially interesting as it allows for the employment of algorithms that were designed with parallelism in mind. The algorithm of Goldberg and Tarjan [24] is such an algorithm, running in time $O(|V||E|log(|V|^2/|E|))$. In the algorithm, the incoming flow for a vertex is allowed to exceed the outgoing one; excess flow is then pushed along shortest paths towards the sink. The flow is pushed in a series of step, where in each step the vertices are relabeled to reflect the new flow distribution, giving rise to the term *push-relabel* algorithm.

2.3 Matching in general graphs

The first polynomial-time algorithm for finding a maximum matching in general graphs was proposed by Edmonds in 1965 [14]. While not defining an algorithm in the strictest sense, it rather put forward a set of ideas that could be used to develop further algorithms, such as that of Hopcroft and Karp [13] for bipartite graphs.

The idea is, as with the bipartite case, to iteratively increase a matching using augmenting paths. Without the structural constraints of a bipartite graph, however, the algorithm becomes more complex, and its discussion would fall outside the topic of this thesis.

Using the ideas of Edmond's algorithm, Silvio et al. [10] were able to formulate an $O(\sqrt{|V|}|E|)$ algorithm for maximum matching in general graphs, thereby matching the running time of Hopcroft and Karp's algorithm [13] for bipartite graphs.

Thus far, all the matching algorithms presented have relied on augmenting paths for constructing a maximum matching. Another, more recent work by Mucha et al. [9] uses randomization and fast matrix multiplication, running in $O(n^\omega)$ randomized time, where $O(n^\omega)$ is the complexity of the fastest known matrix multiplication algorithm.

As of 2013, the fastest published algorithm for matrix multiplication is that of Coppersmith and Winograd [15], with $\omega = 2.376$. A more recent, but as of yet unpublished work by Williams [16] gives an algorithm with $\omega \leq 2.373$. In contrast, the bound of $O(\sqrt{|V|}|E|)$ obtained by Silvio et al. translates into $O(n^{2.5})$ for complete graphs, but closer to $O(n^{1.5})$ for sparse graphs.

A common characteristic of these matrix multiplication algorithms is that they rely on large hidden constants to achieve their bounds for the running time. In the words of Chris Umans of the California institute of Technology, although the algorithms "*perform better than Strassen aymptotically, the input matrices must be astronomically large for the difference in time to be apparent*" [17]. Strassen's algorithm [18] gives $\omega \approx 2.806$, resulting in a running time that is significantly slower than that of Silvio et al.

2.4 Heuristics

In real-world scenarios, obtaining the best possible solution is not necessarily the only requirement; another might be that the solution has to be computable within a reasonable amount of time. Consider a graph with a hundred million vertices and an average vertex degree of 10. A running time of $O(\sqrt{|V|}|E|)$, which, as was shown in the previous section, is the fastest known matching algorithm *in practice*, would then imply that the number of computations performed could be in the order of $\sqrt{10^8} \times 10^9 = 10^{13}$, or 10 trillion.

A*heuristic* might guarantee a much lower running time than an exact algorithm, but at the expense that the solution it finds is not necessarily optimal. Recall that a *maximal* matching was introduced as a matching not necessarily of the largest possible size, but where no further edges could be added. In other words, given the choices that have already been made, there is no "quick fix" to improve the matching; in order for it to be extended, edges already part of the matching must be removed.

The Karp-Sipser algorithm is such a heuristic procedure, and will be the topic of the next chapter. It works in a series of steps, where in each step one edge is added to the matching, and its vertices are removed. This guarantees that the number of steps is proportional to the number of vertices in the graph, but at the expense that the choice of a particular edge might not lead to a *maximum* matching.

2.5 Summary

This chapter introduced the matching problem, with a special focus on bipartite graphs as their structure is well-suited for presenting algorithmic ideas. Exact matching algorithms for general graphs were covered briefly; their relative complexity serves as a justification for developing heuristic procedures, with one such algorithm, *Karp-Sipser*, being the topic of the next chapter.

Chapter 3

The Karp-Sipser algorithm

This chapter will first present a simple greedy algorithm for the maximal matching problem, in order to show how utilizing simple observations may yield powerful results in terms of solution size. The Karp-Sipser algorithm itself will be presented in Seciton 3.2, while towards the end of the chapter, an analysis of the Karp-Sipser algorithm's running time will be given (Section 3.2.1), along with a presentation of the data structures used for its implementation (Section 3.2.2).

3.1 A simple greedy heuristic

Recall that for a matching to be valid, no vertex may be included as an endpoint more than once. As such, when adding an edge $e = (u, v)$ to a matching, u and v should not be considered for inclusion again, unless e is later removed from the matching. By permitting edges to be examined at most once it is ensured that no more than $|E|$ such operations are performed, thus providing an upper bound of the running time of the form (number of examined edges) $\times$ (time taken to examine an edge).

This leads to an idea for a simple, *greedy* heuristic: repeatedly add edges to the matching and remove their endpoints, as well as any incident edges, until there are none left. The greedy algorithm is given in Algorithm 2.

Algorithm 2 A simple greedy matching heuristic

```
procedure GREEDY MATCHING(Graph $G$)
    $M := \emptyset$
    while $G$ has edges left do
        Select an edge $e = (u, v)$ at random
        $M := M \cup \{e\}$
        remove $u$ and $v$ from $G$
    end while
end procedure
```

Two vertices are removed in each iteration of the while loop, guaranteeing that it will be run at most $|V|/2$ times.

What about the solution size? Let the *quality* of a matching M be the size of M relative to the maximum matching size for that graph:

$$\text{Matching Quality}(G, M) = \frac{|M|}{\text{Size of maximum matching}} \tag{3.1}$$

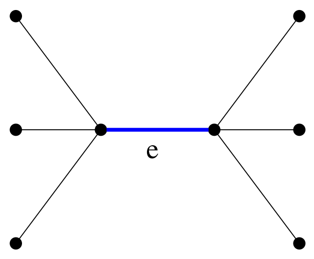

Figure 3.1: A graph with maximum matching size 2. If edge e (blue) is chosen to be part of a maximal matching, no further edges may be added and $|M| = 1$.

An example run of Algorithm 2 is shown in Figure 3.1, with a resulting matching quality of 0.5. In this particular case, edge e was selected as the first edge; if any other edge had been selected initially, the resulting matching quality could be 1.

In fact, the greedy algorithm will always produce a matching quality greater or equal to 0.5 [19]. Intuitively, this may be explained as follows: consider an edge, such as e in Figure 3.1. As no vertex may occur as an endpoint in a matching more than once, selecting e prevents any adjacent edges from being part of the matching. However, exactly because no endpoint may occur more than once, the selection of e prevents at most two optimal edges from being part of the solution.

Here, an optimal edge means an edge that would have been part of a maximum matching. Thus selecting e disqualifies at most two other edges, but it *does* increase the matching size by one. Hence the greedy algorithm will always produce a matching with quality ≥ 0.5. For a formal proof, see [19].

3.2 The sequential Karp-Sipser algorithm

The Karp-Sipser algorithm [1] (Algorithm 3) prevents the scenario shown in Figure 3.1 by giving preference to matching vertices of degree 1, hereafter referred to as *singletons*. If there is more than one singleton vertex in a given step, one is chosen uniformly at random. If there are no singletons, a random edge is selected, similar to the greedy heuristic presented in Section 3.1.

Karp and Sipser refer to the algorithm as working in two phases, where the first phase extends from the start of the algorithm until the first time the set of singletons is empty. It will now be shown, with the arguments from [1], that the choices made in the first phase are optimal, i.e. if the algorithm is terminated after the first phase, the resulting (partial) matching could be extended to a *maximum* matching.

A vertex is said to be *lost* with respect to an edge set E if it does not occur in E as an endpoint. Consequently, for a graph with no vertices of degree 0, the number of lost vertices, $l(M)$, resulting from the choice of the edge set M as a matching, is described by

$$|V| = 2|M| + l(M) \tag{3.2}$$

Consider the matching in Figure 3.1: the number of lost vertices, $l(M) = 6$. If e is not present in the matching, one may select any one edge to the left of e and any one edge to the right, doubling the matching size and resulting in only four lost vertices.

Karp and Sipser show that in general, the number of lost vertices after a run of the algorithm on a graph G may be described as

$$l(E(G)) = l_1 + l(E(H_1)) \tag{3.3}$$

where H_1 is the remaining graph after the first phase and l_1 represents the number of vertices lost in that phase. They show that while $l(E(H_1))$ depends on the random choices made by the algorithm, l_1 depends only on G.

Another, perhaps more intuitive proof, may be sketched by following the same argument as was given in Section 3.1 for proving that the greedy algorithm guaranteed a matching quality of ≥ 0.5. Consider a singleton vertex, v, and the corresponding edge $e = (u, v)$ for some u. From Section 3.1, the argument is that if e is not part of some maximum matching, selecting it could prevent at most two optimal edges from being part of the matching.

However, since v has degree 1, selecting (u, v) for the matching would only prevent the selection of at most one optimal edge that has u as an endpoint. In other words, it would prevent at most one optimal edge from being part of the matching, but the matching size is increased by one, i.e. the choice of (u, v) is also optimal.

The optimality of Phase 1 has lead to the Karp-Sipser algorithm being suitable for computing an initial matching that is later extended by an algorithm for maximum matching. This is particularly relevant due to the fact that almost all maximum matching algorithms operate by iteratively increasing a partial matching [21].

Others, such as Aronson et al. [22], have studied the average performance of Karp-Sipser for *sparse random graphs*, in terms of matching quality, and found that it produces a maximum matching with high probability when the average vertex degree is smaller than e, the base of the natural logarithm. If the average degree is greater than e, the matching calculated is within $|V|^{1/5+o(1)}$ of the maximum matching, with high probability.

Algorithm 3 Sequential Karp-Sipser algorithm

```
procedure SEQUENTIAL KARP-SIPSER ALGORITHM(Graph $G$)
    $M := \emptyset$
    while $G$ has remaining edges do
        if $G$ has at least one singleton vertex $x$ then
            $e :=$ The edge between $x$ and its neighbour
        else
            $e :=$ a random edge
        end if
        Remove $e$'s endpoints from $G$, as well as all incident edges
        $M := M \cup \{e\}$
    end while
end procedure
```

The remaining sections will present an analysis of the Karp-Sipser algorithm, including required data structures. In order to simplify the implementation, a slight modification will be made to the Karp-Sipser algorithm (Algorithm 3): instead of selecting edges uniformly at random, a vertex of degree > 0 will first be selected uniformly at random, before one of its adjacent vertices is chosen in the same manner. This increases the probability of selecting edges whose endpoints have lower degree, but as it occurs in Phase two it does not affect the optimality of Phase one.

3.2.1 Analysis

The running time analysis of the algorithm is quite straight-forward. Note from the previous section that due to its nature of deleting at least two vertices in each step, the algorithm will terminate after at most $|V|/2$ steps. In order to express the total running time, the operations performed in each step will be examined. The running time of these operations will in turn depend on the underlying data structures, whose characteristics will be briefly sketched here and outlined further in the next section.

Singletons and sets

The algorithm maintains a set of *singleton* vertices; the set implementation needs to support four operations: to add or remove members, to get a random element, and a method to get the size of the set itself. This set is implemented using two arrays, one to store the actual set *members* contiguously and one to support lookup of individual vertex indices; each array has size $|V|$. Using this implementation, which is explained further in the ensuing *Data structures* section, all four operations may be performed in constant time.

Selecting random edges

To allow for the random selection of edges, it is necessary to also maintain a set of vertices with degree > 0, which is the set of all vertices with edges incident on them; the set implementation is the same as for the set of singletons. When selecting a random edge, a vertex with degree > 0 is selected first, and then one of its incident edges, both operations uniformly at random. This has the side-effect of increasing the probability for selecting edges whose endpoints have lower degrees, as opposed to selecting edges uniformly at random.

Removing vertices

When removing a vertex v it is necessary to iterate over its $d(v)$ neighbours in order to update the sets of singletons and vertices with nonzero degree. This gives an upper bound of $\mathrm{O}(|V|)$ for the number of steps required to remove a vertex. However, one may instead consider the *average* time taken to delete neighbours, which depends on the average degree of a vertex, $d_{avg} = |E|/|V|$. The average time to delete a vertex may now be expressed as $\mathrm{O}(d_{avg})$.

Total running time

The algorithm runs at most $\mathrm{O}(|V|)$ steps. On average, each step takes $\mathrm{O}(d_{avg})$ time, giving a total of $\mathrm{O}(|V| * d_{avg}) = \mathrm{O}(|V| * |E|/|V|) = \mathrm{O}(|E|)$. Finally, it may be noted that for sparse graphs, one could have $\mathrm{O}(|E|) = \mathrm{O}(|V|)$.

3.2.2 Data structures

Vertex sets

The matching algorithms rely on vertex sets when e.g. selecting a random vertex of degree 1 or determining whether a vertex is active. Using one array for looking up vertex IDs (L, of size $|V|$), i.e. determining whether a particular vertex is part of the set, and another array for storing the set elements contiguously (S, of size $|V|$), the following operations may be performed in constant time:

1. Determining whether an element is part of the set
2. Removing an element from the set
3. Adding an element to the set
4. Selecting a random element

Upon initialization, the set accepts a value, *maxSize*, to determine the size of the underlying data structure, while a variable *size* determines how many elements are currently part of the set, initially $size = 0$. Providing a $maxSize$ of $|V|$ lets one add elements 0 through $|V| - 1$. An example is shown in Figure 3.2, where $maxSize = 7$, so elements 0 through 6 may be added.

The lookup array, L, stores -1 if an element is not part of the set, or a non-negative index otherwise; this index is the position at which the element is stored in the contiguous set members (S) array, and is used when removing a set element.

When adding an element i to the set, i is put at position $size$ of S, and $L[i]$ is updated to store the value of $size$. Finally, $size$ is incremented by 1. If the element 1 was added to the set in Figure 3.2, it would be stored at $S[4]$ and $L[1]$ would thus equal 4.

Removing a set element i is done by swapping its position in S with that of the last element in S, before updating L. Finally, $size$ is decremented by 1. Consider the set in Figure 3.2: if element 6 was removed, element 5 would be moved to $S[0]$, $L[6]$ would be set to -1 and $L[5]$ would be set to 0, to reflect that 5 was moved.

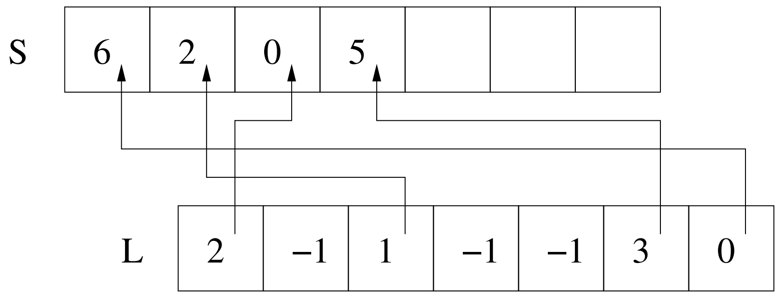

Figure 3.2: A vertex set containing elements 0, 2, 5 and 6

Graph representation

The graph is represented using the *adjacency list* format. An array, *edgeList*, of size $|V|$ stores pointers to a list of neighbours for each vertex. These lists are stored consecutively in an array, *edgeData*, of size $2|E|$. An edge $e = (u, v)$ is thus represented twice: u occurs in the adjacency list of v, and vice versa. The number of edges incident on a particular vertex is stored in an array, *edgecCount*, of size $|V|$. Thus, the total memory usage of the graph representation is $O(|V| + |E|)$. An illustration of the data structure is shown in Figure 3.3.

To avoid the use of an extra array for storing edge counts per vertex, the edge count of a vertex with ID i might have been calculated as $edgeList[i+1] - edgeList[i]$. This would, however, fail to provide a method for determining the edge count of a vertex if edges are allowed to be removed from the graph, which is the case for the implementation discussed in this thesis.

An alternative to the edge list representation could be the adjacency matrix format, where a graph is represented as an $n \times n$ boolean matrix A, with $n = |V|$. The adjacency matrix format provides two advantages: given two vertices with ID i and j, testing for the existence of an edge (i, j) simply involves evaluating $A[i][j] \neq 0$, and removing an edge can be done by setting $A[i][j] = 0$. Listing the neighbours of a vertex, or selecting a random neighbour of a given vertex, would however require $O(|V|)$ time unless some additional data structure is used. Finally, a adjacency matrix requires $O(|V|^2)$ space, which can be a factor $O(|V|)$ larger than the adjacency list format for sparse graphs where $O(|V|) = O(|E|)$.

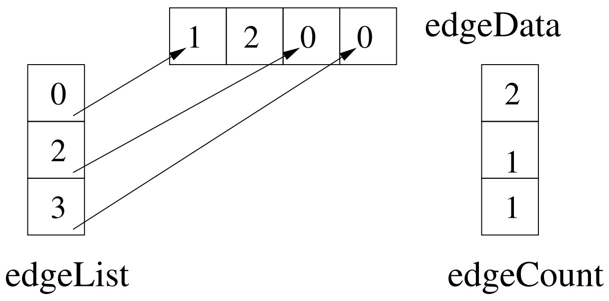

Figure 3.3: An undirected graph with $|V| = \{0, 1, 2\}$ and $|E| = \{(0, 1), (0, 2)\}$

Chapter 4

Parallelization

This chapter is concerned with two main aspects of parallelism: what a parallel computer is, and how one should divide data in other to make a sequential problem into a parallel one. These two aspects translate into two discussions: the choice of a theoretical computer model and data partitioning schemes.

This chapter starts by further examining the motivation for computation in parallel. A discussion of graph partitioning schemes is given in Section 4.3, while Section 4.2 will discuss theoretical computation models used for analyzing the partitioning schemes and the parallel algorithms. Finally, Section 4.4 will give an overview of the implementation of the parallel algorithms.

4.1 The need to go parallel

How does one solve a computational problem faster? A tempting answer is to simply run the implemented algorithm on a faster computer; however, the speed of today's processors touches a barrier, a fundamental limitation in our current understanding of physics: the speed of light, a limit for how fast information can travel. A modern processor has a clock frequency in the order of a few GHz, which translates into 10^9 operations per second, i.e. each operation takes about $10^{-9}s$. Most of these operations (adding numbers, multiplication etc.) requires input in the form of numbers stored in system memory, or RAM.

For simplicity, assume that the memory is located 3 cm from the processor. The time it takes for information to travel from the processor to memory and back, the *latency* incurred when requesting information, is then bounded from below by the speed of light:

$$\frac{\text{distance traveled}}{\text{speed}} = \frac{2 \times 3 \times 10^{-2}m}{3 \times 10^{8}ms^{-1}} = \frac{2 \times 10^{-2}m}{10^{8}ms^{-1}} = 2 \times 10^{-10}s \tag{4.1}$$

Thus this lower bound is only one order of magnitude away from the speed of current processors, and it should be noted that current memory transfer mechanisms do not necessarily achieve such speeds. To alleviate this, memory may be moved closer to the CPU; a modern processor typically includes a *CPU cache*, which is memory embedded in the processor die, used for storing frequently accessed information or CPU instructions.

Instead of relying on doing computations faster on a single machine, another approach is to go *parallel* - dividing a problem into several smaller parts, and solving them simultaneously with multiple processors.

Parallelization raises another question: how does one divide the problem between processors? This thesis will examine two approaches for graph partitioning, by parallelizing the Karp-Sipser matching algorithm: the graph may be divided by assigning each process a set of the graph's *vertices*, or by assigning a set of the graph's *edges*, sometimes referred to as a 1D or 2D partitioning, respectively.

These two data partitioning schemes will be discussed in Section 4.3; before discussing them, it is necessary to define a theoretical model for what a parallel computer really is: Section 4.2 presents two main varieties, *shared* and *distributed memory* computers.

4.2 Computation model

There are two main types of parallel computers: shared memory and distributed memory multicomputer [23]. In a computer with shared memory, the processes all have access to the same memory area, and consequently data may be shared with ease amongst processes. A challenge with such a computer is the hardware: while computers granting equal access to a common memory area for a small number of processes have been around for some time, only a decade ago there was considerable doubt [23] about whether the shared-memory approach would scale.

With the introduction of hardware such as the Intel Xeon Phi coprocessor family, systems with 57 - 61 cores, supporting up to 244 processes, can be purchased for as little as USD$ 2,000 [33]. Previously, such massive multithreading has only been available with highly specialized hardware such as the Cray XMT [34], produced in a limited number and designed for supercomputer systems.

In a distributed memory computer, each process has its own memory area, and some form of explicit communication is required when sharing information between processes. A common method is *message passing*, where the processes communicate by sending messages to each other, using a common API such as MPI [35]. While each *process* may be assigned to an individual *processor* (CPU), this is not necessarily the case; several processes might share a single CPU, as is the case with Hyper-Threading technology.

Developing parallel applications using a message passing interface, as opposed to using a shared memory model, puts an extra burden on the programmer as the sharing of information has to be done explicitly. On the other hand, benefits may be reaped in terms of avoiding time-consuming implicit synchronization mechanisms.

Another approach is to achieve parallelism by passing the same instruction set to multiple processes, each using different data, known as Single Instruction Multiple Data (SIMD) [23]. This is especially suited for situations where the results of the computations are independent, such as deciding what colour a pixel on a computer screen should be, and Graphic Processing Units (GPUs) thus often have a SIMD structure.

4.2.1 Bulk-Synchronous Processing (BSP)

The parallelization of the Karp-Sipser algorithm in this thesis, both using vertex partitioning and edge partitioning, is performed under a paradigm known as Bulk-Synchronous Processing (BSP) [5]. While BSP relies on message passing, it tries to alleviate some of the extra complications involved in programming a message passing application by structuring communication and computation.

Execution is done in a series of *supersteps*, each consisting of two phases: computation and communication. Each process is required to enter and finish the communication phase before the program can proceed to the next step, thereby ensuring a form of synchronized execution, and eliminating the need for synchronization mechanisms within a computation step.

A reference BSPlib standard [5] provides primitives that are suitable for a higher-level approach in comparison to MPI, leaving optimization to the developers of a BSPlib implementation. For the experiments in this thesis, the functionality needed for the BSP computing model is provided by the library BSPonMPI [4], which in turn is implemented using the MPI framework.

The two main primitives of BSP are *bsp_ send()* and *bsp_ sync()*. In a computation phase, a process adds messages to a communication buffer using *bsp_ send()*; these messages are stored and only sent when all processes enter the communication phase by calling *bsp_ sync()*. The primitive *bsp_ sync()* is a *blocking* operation, i.e. once a program calls it, execution is not resumed until all processes have reached the same point. A message sent from a process i to a process j is thus available to j after both processes have called *bsp_ sync()*; an example is shown in Figure 4.1.

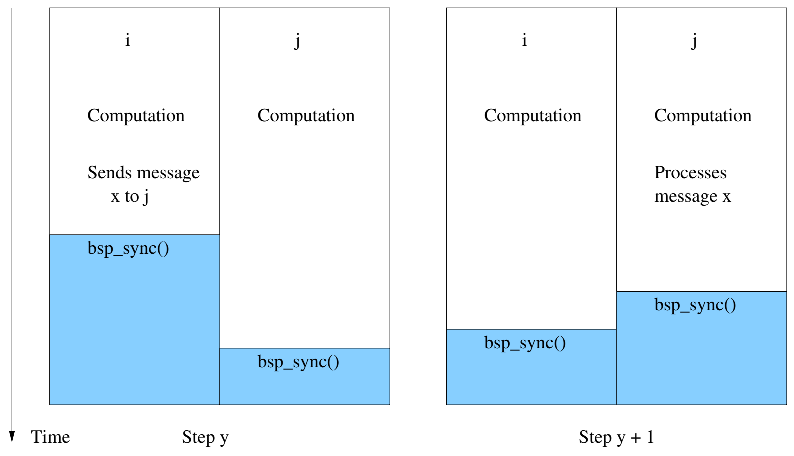

Figure 4.1: An example run of a BSP program. In step y, process i sends a message x to process j, which is processed by j in step $y + 1$. The computation phase (white) of step $y + 1$ begins when all processes have entered the communication phase (blue) of step y.

4.3 Data partitioning

This section presents two data partitioning schemes: vertex partitioning and edge partitioning. Data partitioning refers to the following problem:

Given a graph $G = (V, E)$, how does one divide G into p parts? The p parts refer to p processors, or more specifically *processes*, the difference of which was described in Section 4.2.

Any partitioning would entail that some process does not see and/or work with the whole graph at some point. However, the matching problem, as defined in Section 2.1, requires that no vertex may be included as an endpoint more than once. This is solved by introducing a notion of vertex *ownership*. The owner of a vertex will be required to keep track of whether it is part of a matching or not, and thus has the final say when including it in the result.

In vertex partitioning, a process i is assigned to a set $V_i \subseteq V(G)$, which implicitly also assigns a set of edges, $E_i \subseteq E(G)$, where an edge $(u, v) \in E(G)$ is in E_i if i owns u,

v, or both. In this case, vertex ownership becomes trivial: a process i owns a vertex u if $u \in V_i$. The assignment of E_i in turn implies the assignment of a set of *ghost vertices*, V_i^g: the set of vertices not owned by i, but adjacent to vertices owned by i. Ghost vertices are used for communication: if a process encounters an edge $e = (u, v)$ where v is owned by some other process, it can perform a query to a local data structure to retrieve the process ID of the owner of v and consequently sent a message. For an edge $(u, v) \in E_i$, there are three cases: 1) $\{u, v\} \in V_i$; 2) $u \in V_i, v \in V_i^g$; 3) $v \in V_i, u \in V_i^g$.

For the edge partitioning approach, the set $E_i \subseteq E(G)$ is assigned explicitly, which requires a separate mechanism for determining vertex ownership. In the worst case, a vertex u may be shared among $d(u)$ processes, assuming $d(u) \leq p$. The owner of a vertex is chosen to be the process owning the highest number of incident edges to it; if no process owns the plurality of incident edges, ownership is awarded to the process with the lowest ID.

The set of ghost vertices V_i^g for a process i is then the set of vertices that occur as an endpoint in E_i, excluding the set V_i. Thus a process i may not have knowledge of all edges incident on a vertex $u \in V_i$, or it may own vertices u and v but be unaware of the edge (u, v). This is solved by letting the owner of a vertex know about the set of processes owning edges incident on that vertex.

These, and other, problems that arise with the edge patitioning approach are discussed in Chapter 7, while the vertex partitioning approach is discussed further in Chapter 6.

4.3.1 Load balancing

At this point it might be relevant to question why the more complicated edge partitioning approach should be given further consideration. The answer is that of *load balancing*. As was shown in Figure 4.1, a process that finishes its computation phase earlier than some other process cannot continue execution until all processes have finished communicating. Thus while a vertex partitioning may yield an algorithm that requires less computation per process, this may be offset by an uneven load balancing, resulting in an overall longer execution time.

Consider a graph consisting of four vertices, forming a tree such that the vertex with ID 0 is connected to the remaining vertices. A division of this graph amongst three processes, using both vertex and edge partitioning, is shown in Figure 4.2. In general, graphs where some vertices have a much higher degree than the average may lead to poor load balancing, as a high-degree vertex has to be assigned to only one processor.

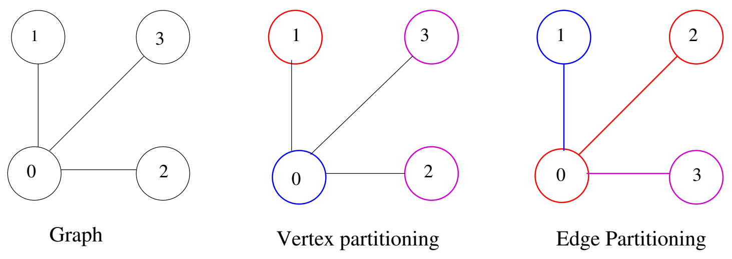

Figure 4.2: A graph and a corresponding vertex and edge partitioning among three processes (red, blue, magenta) where red has the lowest process ID.

4.4 Implementation

This section briefly covers some of the details that are common to the implementation of both parallel algorithms, discussed in Chapters 6 and 7. It covers the method used for communicating between processes, and how graph partitioning is handled.

4.4.1 Communication

The interaction between the program and BSPlib is done through a *Communicator* interface. While performing initial experiments, it was discovered that BSPlib had problems when distributing larger graphs to a larger number of processes. To alleviate this, the *Communicator* interface allows for the graph to be distributed in smaller parts.

Furthermore, using the *Communicator* interface, it is possible to record the exact number of messages sent and received, which in the following chapters will be used for comparisons with the theoretical bound on the communication volume.

On the other hand, it is aknowledged that an extra layer between the algorithm implementations and the underlying MPI library, may have a negative impact on performance. For those familiar with C++, the Communicator class is a *virtual* class with implementations for both BSPlib (which in turn is based on MPI) and MPI itself; thus the use of *inline* functions would not necessarily be optimized by the compiler. Only the BSPlib implementation of the Communicator interface was used during the experiments, and as such this situation could have been avoided, but it was the author's desire to create a framework that could be used for testing other parallel algorithms, without the BSP model being a requirement.

4.4.2 Graph partitioning

For the vertex partitioned algorithm, graph partitioning is done via a common interface, so that different partitioners may be tested with minimal effort. The interface consists of a single method, *partition*, taking two arguments: a graph, in compressed row storage (CRS) format, and the process count.

For the vertex partitioning, METIS [2] and SCOTCH [3] were utilized. Both software packages have parallel versions in order to speed up the partitioning process, but the sequential versions were chosen as the data distribution time was excluded from the experiments performed. An interface for the Zoltan [36] hypergraph partitioner could also have been developed; however, a comparison of graph partitioning software would fall outside the topic of this thesis.

For the edge partitioned algorithm, the Mondriaan partitioner [7] for sparse matrices is used. As is discussed further in Chapter 5, Patwary et al. [6] showed that the communication volume for edge partitioned parallel matching is proportional to that of sparse matrix vector multiplication, thus Mondriaan can be used without any modifications.

Chapter 5

Parallelization - related work

As was explained in Section 4.3, implementing an algorithm with an edge partitioning approach requires extra work in comparison to vertex partitioning. This chapter examines previous works related to the matching problem, and through this examination provides some justification for the edge partitioning approach. Section 5.2 examines parallel algorithms for bipartite matching, while Section 5.3 examines parallelization of algorithms designed for general graphs.

In both sections, the discussion is organized based on the two main computation models, *shared* memory and *distributed* memory.

5.1 Speedup

In order to evaluate a parallel algorithm, a notion of how to measure its performance is required. Let the *speedup* of an algorithm using p processes be given as:

$$\text{Speedup}(p) = \frac{\text{Running time of the best sequential algorithm}}{\text{Running time of the parallel algorithm using } p \text{ processes}} \tag{5.1}$$

Another, perhaps more cynical, approach would be to measure the *efficiency* of the parallel algorithm, by evaluating how often a process is active, in order to perform an analysis from a cost-benefit perspective.

5.2 Bipartite matching

5.2.1 Shared memory

The parallelization of matching algorithms for bipartite graphs requires special attention, due to the existence of *push-relabel* algorithms, as descried in Section 2.2. In their original paper, Goldberg and Tarjan give a parallel algorithm running on $O(|V|)$ processes, but without any implementation. As explained in Chapter 1, real world graphs may have millions of vertices, and as such an implementation with millions of processes might be impractical.

In 1992, Setubal [25] provided a parallel implementation of Goldberg and Tarjan's algorithm and showed that a speedup of 3.2 could be obtained with up to 12 processors on a shared-memory system on random graphs.

More recently, in 2012, Azad et al. [27] implemented parallel versions of several *maximum* matching algorithms, where parallel versions of Karp-Sipser and the greedy matching heuristic, as explained in Chapter 3, were used for providing an initial matching. Once initialized, their algorithms rely on increasing the initial matching by finding augmenting paths in parallel, using either vertex-disjoint depth-first search (DFS), DFS with lookahead, breadth-first search (BFS), or, using a modified version of the Hopcroft-Karp algorithm, a combination of both.

They tested their algorithms on both systems with a few dozen processes (1-2 processes per core) and a more special system with 16 384 processes using 128 processes per core, the *Cray XMT*. They find that the Karp-Sipser algorithm as an initialization mechanism produces better results when using dynamic scheduling, as used for the DFS based algorithms, whereas the greedy heuristic performs best for static scheduling, as used in the BFS based algorithms.

Using these initalizaiton mechanisms, they were able to obtain good speedups for the maximum matching problem on systems with a few dozen processes, where the DFS- based algorithms performed best on average, and for the massively multithreaded system, where the BFS based algorithms offered the highest speedup.

More recent still, in October 2013, Dufossé et al. [28] developed a maximal matching heuristic that uses a parallel version of the Karp-Sipser algorithm as a subroutine. Their algorithm works by scaling the corresponding adjacency matrix of a graph to doubly stochastic form, from which they are able to select subgraphs with properties that allows their parallel version of Karp-Sipser to find a maximum matching. They show that their algorithm guarantees a matching quality of at least 0.866, and give an implementation that achieves a speedup ranging from 9 to 12 when using 16 processes.

5.2.2 Distributed memory

A recent attempt at parallelizing the push-relabel algorithm was performed by Langguth et al. [26] in 2011. The shared-memory algorithm of Setubal is adapted to a distributed-memory environment by limiting the number of communication operations performed in each push-relabelling step. In addition, they test the usage of the Karp-Sipser algorithm as an initializing method for speeding up computation; while they find that the use of Karp-Sipser does not provide a predictable speedup, it does pay off on average. Their algorithm uses an edge-partitioning approach, which is based on the findings of Patwary et al. [6] described in the following section.

5.3 Matching for general graphs

5.3.1 Shared memory

With the onset of GPU programming, a new possibility for running shared-memory parallel algorithms emerged. Auer et al. [8] found that the memory bandwidth of modern graphics cards, which can be tenfold that of CPU-RAM bandwidth, made their matching algorithm implemented for the GPU significantly faster than the same algorithm implemented on a system with eight physical cores. They implemented their algorithm in two versions: maximal matching for unweighted graphs, and an approximation algorithm for *weighted matching*, where the edges have weight, and the goal is to maximize the total weight of the edges included in the matching.

A GPU based algorithm is limited by the number of cores available on a graphics card; a system of interconnected GPUs would potentially defeat their benefit of a large memory bandwidth.

5.3.2 Distributed memory

This section is divided into two sections: parallelization of approximation algorithms for *weighted* matching and maximal cardinality (unweighted) matching.

Weighted matching

Using the MPI framework for message passing, Catalyurek et al. [30] developed a parallel 0.5 approximation algorithm for the weighted matching problem for distributed memory computers. Their algorithm is based on finding locally heaviest edges on each processor, and while this is shown to only guarantee a solution that has at least half the weight of the optimal one, they experimentally obtain matchings that deviate less than 1 % from the optimal. They obtain speedup using up to 4,096 processors using the IBM Blue Gene/P supercomputer on sparse graphs.

For a non-supercomputer system, Manne and Bisseling [31] devised a parallel algorithm based on finding local heaviest edges. Their algorithm operates in phases, where local matching operations is followed by communication for vertices that have neighbours owned by other processes, leading to a BSP style algorithm. The algorithm is shown to give good speedup using up to 32 processors for both sparse and complete graphs.

Maximal cardinality matching

The idea of comparing vertex and edge partitioning for graph algorithms is a novel one, with the first work of Patwary et al. [6] published in 2010 for the maximal matching problem for sparse graphs. The idea was inspired by the field of sparse matrix-vector multiplication; a symmetric $n \times n$ matrix A, where all entries on the diagonal are zero, may be viewed as an undirected graph with $|V| = n$ vertices, where there is an edge between vertices i and j if $A[i][j] \neq 0$.

A vertex partitioning thus corresponds to a partitioning of *rows* or *columns* in matrix terms, whereas an edge partitioning corresponds to a partitioning of *non-zero* elements. By establishing that the communication volume for an edge partition based Karp-Sipser matching algorithm is proportional to that of sparse matrix-vector multiplication, Patwary et al. indicated that partitioners developed for that purpose, such as Mondriaan [7], may be used.

They tested their findings in an experiment where a one-dimensional partitioning was *simulated* by constraining the matrix partitioner to assigning all nonzero elements up to the diagonal in a given row to the same processor. Their experiments *indicated* that a two-dimensional partitioning would yield better results both in terms of speedup and matching quality, when compared to a one-dimensional partitioning. While that was indeed the case for their experiments, an edge partitioning requires a more complex data structure and message handling than a vertex partitioning.

In Chapter 6 and Chapter 7 their hypothesis will be tested further by implementing a purely vertex based and a purely edge based algorithm, respectively.

Chapter 6

Karp-Sipser using vertex partitioning

This chapter presents a parallel version of the Karp-Sipser algorithm where the graph is partitioned by assigning each process a set of vertices; the task of determining these sets is performed by dedicated graph partitioning software, as was explained in Section 4.4.

An overview of the algorithm, as well as terminology and definitions, is given in the following section. In Section 6.2, the algorithm is explained in detail, before an analysis is presented in Section 6.3, and the data structures explained in Section 6.4. Experimental results are given in Chapter 8.

6.1 Overview

6.1.1 Terminology

In the parallel version of the Karp-Sipser algorithm, vertices are divided into three sets, that may overlap: *singletons*, as seen in the sequential algorithm, are vertices of degree one; vertices that have at least one adjacent vertex owned by the current process; and vertices of nonzero degree. Furthermore, an edge is said to be *shared* if its endpoints belong to different processes, and *local* otherwise.

Given a vertex u, the process owning u will be denoted as p_u.

6.1.2 The algorithm at a glance

The master process, hereafter referred to as p_0, takes care of reading an input graph, partitioning it and thereafter the distribution of data to the other processes. Once a process has received the data, it proceeds to set up an internal representation of its part of the graph; the data structures are further explained in Section 6.4.

Each superstep of the algorithm has three phases: message processing, queue processing and examination of the local graph. Message processing handles the information sent by other processes in the previous superstep, queue processing takes care of removing or reactivating vertices involved in matching operations with another process, and examination of the local graph performs local matching operations and sends requests to other processes.

For the examination of the local graph, each process attempts to perform a number of matching operations per round; this number is denoted by MPR. Preference is given to singleton vertices; if none are available, random edges for which the process owns both endpoints are matched. If the number of singleton and local matching operations in a given round is less than MPR, the process picks a vertex it owns and sends a matching request to a random neighbour.

Thus the only case where a process does not perform MPR matching operations in a given round, is if it has less than MPR unmatched vertices available. The value for MPR is a question of load balance and matching quality: a lower value will mean that the processes synchronize more often, which may reduce the time any process stays idle waiting for the other processes to call *bsp_ sync()*, and the more frequent communication will mean that fewer random matching requests are performed before it is discovered that a vertex is a singleton, thus improving the quality of the solution. On the other hand, more frequent synchronization results in a higher accumulated startup cost, as each *bsp_ sync()* operation requires synchronization messages and incures a *latency*, and if each process has an even amount of work, frequent synchronization will not improve load balancing.

If a process p_u encounters a *shared* edge $e = (u, v)$ during examination of the local graph, it sends a *matching request* to p_v. At this point, u is temporarily removed from the graph, and u is added to the *queue* pending an answer from p_v. As was explained in Section 4.2.1, it takes two supersteps before p_u receives an answer, if any. During the queue processing stage in round i, a process examines the vertices queued in round $i - 2$. It checks whether each queued vertex u has been permanently removed from the graph or whether it should be added back into the local graph for future matching operations.

At the end of a computation step, each process checks whether it has performed any work in that step. If no messages were received, no vertices were queued and no local matching operations were performed, a message of no activity is sent to the master process. This information allows the master process to determine when the algorithm may be terminated; at that point, it notifies all other process that their (partial) results should be sent to the master process and execution halted. Each process continues execution until the result is requested by the master process, as a call to *bsp_ sync()* is required from each process until a superstep can be completed.

An overview of the algorithm is given in Algorithm 4. The following section explains the algorithm in more detail, with a dedicated section for each of its phases.

6.2 The vertex partitioned algorithm

In this section, the different phases of the parallel algorithm are presented in detail; for an overview, refer to the preceding section. The layout is as follows: Section 6.2.1 describes how the vertex queue is handled and how vertices are temporarily removed from the graph when part of the queue; Section 6.2.2 gives details as to how the local matching operations are performed; while Section 6.2.3 details which messages are sent between processes, and how they are handled.

6.2.1 Queue processing

If a process p_v wants to use a shared edge (u, v) for matching, it sends a matching request to p_u. At this point, v is temporarily removed from the graph, pending an answer; this is done by removing it from the vertex sets that vertices are drawn from when performing matching operations. However, a reply is only sent if the matching request succeeds or if u is temporarily unavailable due to being part in another matching request. If no reply is received, it therefore means that the request has failed because u is part of the matching already.

However, v may have several incident edges, and in that case it has to be added back into the graph to be available for future matching operations. A matching request

Algorithm 4 Overview of parallel Karp-Sipser algorithm with vertex partitioning

```
procedure PARALLEL KARP-SIPSER WITH VERTEX PARTITIONING(Graph $G$)
    $M_{local} := \emptyset$
    while termination not requested by master do
        for $i := 1 \to |messages|$ do
            handle message
            if message[i].type == master requests result then
                send $M_{local}$ to master and terminate
            end if
            if message[i].type == match ok then
                add edge to $M_{local}$
            end if
        end for
        handle queue
        for $i := 1 \to MPR$ do
            if All locally owned vertices are exhausted then
                break
            end if
            Pick an edge $e$ from local graph
            if $e$ is local then
                $M_{local} := M_{local} \cup \{e\}$
            end if
            if $e$ is shared then
                Send match request and add our endpoint to queue
            end if
        end for
        if no activity then
            notify master
        end if
        if master notified of no activity from all processes then
            master process requests result
        end if
        bsp_sync()
    end while
end procedure
```

sent by p_v in round i is processed by the receiver in round $i+1$, and the reply (if any) is processed by p_v in round $i+2$. This is handled by having two queues, as shown in Figure 6.1.

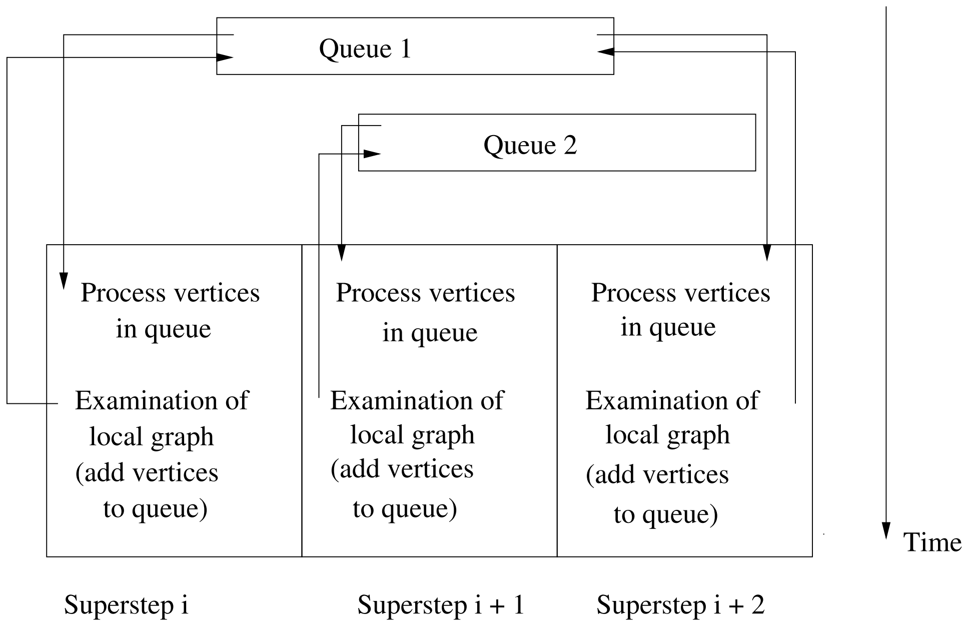
Queue 1
Queue 2
Process vertices in queue
Examination of local graph (add vertices to queue)
Process vertices in queue
Examination of local graph (add vertices to queue)
Process vertices in queue
Examination of local graph (add vertices to queue)
Time
Superstep i
Superstep i + 1
Superstep i + 2

Figure 6.1: Handling of queued vertices. The vertices in Queue 1 are read in superstep i; afterward Queue 1 is overwritten by new queued vertices. Queue 2 is used for reading and writing in superstep $i+1$; the vertices queued in superstep i are read from Queue 1 in step $i+2$.

Thus when p_v processes a queue entry for an edge (u, v) where no reply has been received for two rounds, there are two options: 1) v is a singleton, i.e. u was the only possible match, so v may be permanently removed from the graph; 2) v has degree $>$ 1, so u may be removed from its adjacency list and v added back into the graph.

The vertex is added back into the graph by including it in the relevant vertex sets: vertices of degree 1 are added to the set of *singletons*, and vertices of degree $>$ 0 are added to the set of vertices with nonzero degree. Note that a vertex with local neighbours will never be queued.

6.2.2 Examination of the local graph

In each superstep, every process tries to perform at least MPR matching operations, either by directly matching local edges or by sending matching requests. As in the sequential algorithm, preference is given to matching singletons, regardless of whether the corresponding edge is local or shared. If the corresponding edge is local, the edge is added to the matching and the vertices are permanently removed from the graph. If it is shared, a matching request is sent to the other process, and the locally owned vertex is temporarily removed from the graph and added to the queue, pending an answer, as was described in Section 6.2.1.

If all singleton vertices are exhausted, random matching is performed on vertices with local neighbours, which may be added into the local matching immediately. Both endpoints are then permanently removed from the graph, along with any edges incident on the endpoints; note that this may lead to new singleton vertices. If neither singletons nor local edges are available, random matching is performed using the set of vertices with nonzero degree. The procedure is summarized in Algorithm 5.

Algorithm 5 Examination of local graph

```
procedure LOCAL MATCHING(Graph $G$, Queue $Q$)
    matched := 0
    Q := $\emptyset$
    while matched < $MPR$ do
        if |singletons| > 0 then
            Pick a random singleton vertex $u$ with neighbour $v$
            if $v$ is owned by the current process then
                $M_{\text{local}} := M_{\text{local}} \cup \{(u, v)\}$
                Permanently remove $u$ and $v$ from $G$, and all incident edges
            end if
            if $v$ is owned by another process then
                Temporarily remove $u$ from graph, add $u$ to $Q$
                Send a singleton match request to $p_v$
            end if
            matched := matched +1
            continue
        end if
        if |vertices with local neighbours| > 0 then
            Pick a random vertex $u$ with locally owned neighbour $v$
            $M_{\text{local}} := M_{\text{local}} \cup \{(u, v)\}$
            Permanently remove $u$ and $v$ from $G$
            matched := matched +1
            continue
        end if
        if |vertices of nonzero degree| > 0 then
            Pick a random vertex $u$ of nonzero degree with neighbour $v$
            Temporarily remove $u$ from $G$, add $u$ to $Q$
            Send a singleton match request to $p_v$
            matched := matched +1
            continue
        end if
        break
    end while
end procedure
```

In addition to testing different values for MPR, two strategies for examination of the local graph are tested, and the results presented in Chapter 8; in Algorithm 5, only Strategy 1 is presented. The two strategies are as follows:

Strategy 1: Perform MPR matches per round

In this strategy, exactly MPR matching operations are performed, unless the number of vertices available for matching by the current process is less than MPR. Priority is given to performing matching operations on singleton vertices; let the number of such operations be denoted by SM. If there are not enough singleton vertices available, i.e.

$SM < MPR$, matching is perform on vertices with at least one local neighbour; let the number of such operations be denoted by LM. If neither singletons nor verices with local neighbours are available, i.e. $SM + LM < MPR$, random matching requests are performed until the number of matching operations in the computation step equals MPR, or no vertices are available.

Strategy 2: Perform as much local work as possible, but at least MPR operations

In this strategy, each process tries to perform as much local work as possible. Preference is given to matching singletons, regardless of whether their adjacent vertices are owned by the current process or not. The number of singleton matching operations is not limited by MPR; singleton matching operations are performed until the set of available singletons is empty. When the set of singletons has been exhausted, vertices with local neighbours are matched. Each process performs as many local matches as possible; this may generate new singleton vertices, which are also matched. If, when the sets of singletons and vertices with local neighbours are both empty, the number of matching operations performed is less than MPR, random matching operations are performed until the number of matching operations equals MPR, or no vertices are available.

The hypothesis is that by increasing the value of MPR, or by using strategy 2, more work is done in each round, and consequently the number of rounds should be reduced, leading to fewer *bsp_sync()* operations being performed. On the other hand, this less frequent synchronization is of less importance if several processes run out of work waiting for replies to matching requests. In addition, fewer synchronizations should imply that singleton vertices are discovered less frequently, and as such the matching quality may be reduced.

6.2.3 Message processing

Since the graph is partitioned by assigning sets of vertices to different processes, the endpoints of an edge might have different owners. If a process, say p_i, encounters such an edge, it has to communicate with the owner of the other endpoint, say p_j, to ensure that it has not already been matched, or whether it is *busy*, because p_j has already sent a request to match it with another vertex. A message includes the ID of the sending process, a message ID, vertex IDs (denoted by u and v), the size of the message and, for **M_RESULT**, any additional data. Some of the message parameters may be redundant; if a process p_u receives a request to match u with v, the ID of the sender can be obtained by querying the local graph for the owner of v, instead of reading the message parameter. However, having a set of standard message parameters simplifies the debugging of the implementation, and, more importantly, allows for the graph framework that was developed to be used for implementing other algorithms.

In total there are nine message types (message information in parentheses):

M_MATCH_SINGLETON (u,v) - The sender, p_v wishes to match the singleton vertex v with u. If u is already part of an unresolved matching request, p_u sends an **M_MATCH_BUSY** message; as a special case, if p_u has already sent a request to p_v to match u with v, **M_MATCH_OK** is sent. Otherwise, u has already been matched to some other vertex, and no reply is sent.

M_MATCH(u,v) - The sender, p_v wishes to match u with v. The message handling is the same as for **M_MATCH_SINGLETON**.

M_MATCH_OK(u,v) - Sent by p_u to inform p_v that the request to match u with v succeeded. The edge (u, v) is then added to the local matching, and u and v are permanently removed from the graph.

M_MATCH_BUSY(u, v) - The request to match u with v failed because p_u has already sent a request to match u with another vertex, and has not received a reply yet. Upon receiving this message, p_v removes v from the queue and adds v back into the graph.

M_NO_ACTIVITY - Sent to the master process, this message indicates that the sender did not have any work to do during the previous communication step.

M_REQUEST_RESULT - Sent by the master process if it received messages of no activity from all processes for two consecutive rounds, and is itself idle. Upon receiving this message, a process sends its results to the master process and ends execution after the next *bsp_sync()*.

M_RESULT(sender, resultSize, data) - Sends matching result as pairs of integers.

M_REMOVE_VERTEX(u, v) - Sent by p_u to p_v to inform that u has been removed from the graph, and as such should be removed from v's adjacency list. This allows for the detection of singleton vertices at an early stage. If p_v owns several vertices adjacent to u, it will receive one message for each of its adjacent vertices. Alternatively, this could have been handled by letting the owner of u maintain a set *nonOwners(u)* containing the process IDs of all processes owning vertices adjacent to u. This would in turn require each process to maintain an edge list for its ghost vertices, and notifing the owner of a ghost vertex when its local count is reduced to zero. This is the approach taken for the edge partitioned algorithm, and is discussed further in Chapter 7.

6.3 Analysis

The choice of BSP as the computation model allows for a more accurate analysis of the parallel algorithm's running time; it will depend on the sequential work performed in a computation step for of the process with the most work, the number and cost of *bsp_sync()* operations performed, which in turn depends on the communication volume, the *latency* incurred for a sync operation, and the number of rounds.

To simplify the analysis, only three aspects will be considered: the communication volume, the number of *bsp_sync()* operations, i.e. the number of supersteps before the algorithm is terminated, and the local work performed by a given process. The communication volume is examined in Section 6.3.1, the number of supersteps in 6.3.2, and the sequential workload in Section **??**.

6.3.1 Communication volume

Let each message sent be of unit length, such that the communication volume is expressed in terms of the number of messages sent. Now consider a vertex with ID i that is owned by p_i. From Algorithm 5, observe that there are three cases, which will be examined in turn.

1) i is a *singleton* vertex, thus j is the only available match for i. If $p_i = p_j$, the edge is matched locally and no communication occurs. Otherwise, a matching request is sent to p_j. If the matching request fails, no message is sent, and no further matching attempts for i are carried out. If the request succeeds, p_j sends a confirmation message to p_i. The upper bound for singletons is then $t = 2$ messages.

2) i is a vertex with at least one locally owned incident edge. In this case, all matching operations are done locally. p_i still needs to send messages to any processes owning edges incident on i to inform them that i has been removed. Assuming that

the other $d(i) - 1$ adjacent vertices are owned by other processes, this gives a bound of $t = d(i) - 1$.

3) i is a vertex whose *d(i)* neighbours are all owned by other processes. The worst case is observed when the first $d(i) - 1$ matching requests fail, resulting in $d(i) - 1$ messages being sent. The final request will add one message if it fails, and two if it succeeds, giving a total volume of $t = d(i) + 1$. $d(i)$ is bounded by $|V|$, but since the algorithm is intended for sparse graphs, it may be assumed that $d(i) \ll |V|$. Letting $f(i)$ denote the number of failed matching requests for vertex i, the communication volume may be expressed as $t = f(i) + s(i)$, where $s(i) = 2$ if vertex i is matched, and zero otherwise.

Let s denote the number of *singleton* vertices, and l the number of vertices with *local* edges. Then the bound for the volume, Vol, is obtained as follows:

$$\begin{aligned} Vol &\leq \sum_{i=1}^{s}[f(i) + s(i)] + \sum_{i=s+1}^{|V|-l-s}[f(i) + s(i)] + \sum_{i=1}^{l}[d(i) - 1] \\ &\leq \sum_{i=1}^{s} 2 + \sum_{i=s+1}^{|V|-l-s}[d(i) + 1] + \sum_{i=1}^{l}[d(i) - 1] \end{aligned}$$

As s and l are not known, the upper bound is expressed in terms of $|V|$ only:

$$Vol \leq \sum_{i=1}^{|V|}[d(i) + 1] \tag{6.1}$$

In order to examine how 'good' the communication bound is, it may be compared to the oft-studied problem of sparse matrix vector multiplication, whose communication volume is given by

$$Vol_{spmv} = 2\sum_{i=1}^{|V|}[d(i) - 1] \tag{6.2}$$

$$Vol_{spmv} = 2(\sum_{i=1}^{|V|}[d(i) + 1] - \sum_{i=1}^{|V|} 2) \tag{6.3}$$

$$\frac{1}{2}Vol_{spmv} + \sum_{i=1}^{|V|} 2 = \sum_{i=1}^{|V|}[d(i) + 1] \tag{6.4}$$

$$\tag{6.5}$$

Noticing that the right-hand side now equals the upper bound for the communication volume of the matching algorithm, the following relation is obtained:

$$Vol \leq \frac{1}{2}Vol_{spmv} + \sum_{i=1}^{|V|} 2 \tag{6.6}$$

6.3.2 Number of supersteps

Let $|V_i|$ denote the number of vertices assigned to process i and l_i the number of vertices with local neighbours assigned to i. It is sufficient to consider the work on a single process, assuming it has the highest workload, as the algorithm cannot terminate before all processes have exhausted their locally-owned vertices.

Assuming that all matching requests succeed, one would expect the algorithm to terminate after $|V_i|/MPR$ supersteps, if strategy 1 from Section 6.2.2 is used. However, as MPR increases, the chance of a vertex being part of an ongoing matching request increases as well, thus reducing the chance that a random matching request will succeed. Let $E_{\text{match}}(MPR)$ be the expected number of matching requests that has to be sent before a match is successful; as MPR increases, so does E_{match}. Using the number of local edges, l_i, the expected number of supersteps of the algorithm using strategy 1 may be described as.

$$|\text{Supsersteps}(1)| = \frac{l_i + (|V_i| - l_i)E_{\text{match}}(MPR)}{MPR} \tag{6.7}$$

Using strategy 2, all local edges are matched in the first superstep, thus $\frac{l_i}{MPR}$ may be replaced with 1, giving the expected number of supersteps as

$$|\text{Supersteps}(2)| = \frac{(|V_i| - l_i)E(MPR)}{MPR} + 1 \tag{6.8}$$

6.3.3 Sequential work

The sequential work in a computation work consists of three elements: message processing, queue processing and examination of the local graph. First consider the message processing: a maximum of $MPR \times p$ matching requests may be sent in a given superstep; on average a process should expect to receive at most MPR matching requests. If a process p_u receives a matching request for the vertex u, it first has to check whether u is part of an ongoing matching request; this may be done in constant time, as is described in Section 6.4. If u is available, it is removed from the graph; the cost of removing a vertex is $\mathrm{O}(d_{avg})$, where d_{avg} is the average vertex degree, as described in Section 3.2.1.

Examining the vertex queue is done by iterating over the queued elements, of which there may be at most MPR, unless matching strategy 2 from Section 6.2.2 is used, and the number of singleton vertices is significant; assume that it is not. Finally, examination of the local graph is similar to that of the sequential algorithm (Section 3.2.1), taking at most $\mathrm{O}(d_{avg})$ time for each matching operations, of which MPR are performed.

This gives the cost of a computation step as $\mathrm{O}(MPR \times d_{avg})$.

6.4 Data structures

6.4.1 Graph representation

Vertex IDs

Let V_i be the set of vertices assigned to process i, and $n_i = |V_i|$. For the vertex-partitioned graph, each process needs to store information that allows it to communicate with the owners of vertices adjacent to those in V_i. When referencing vertices internally, each process will use a local ID, in the range 0 through $n_i - 1$, while a global ID in the range 0 through $|V| - 1$ is used when communicating with other processes.

For converting IDs, each process has an array *localIdToGlobalID* of size n_i, and *globalIDToLocalID* of size $|V|$. The owner of each vertex is stored in an array *owners*, of size $|V|$. With a total of p processes, this requires $\mathrm{O}(|V|p)$ space. An alternative is for communication to be done using local vertex IDs, and storing the local ID (for the owner process) of ghost vertices.

Shared and local edges

The storage of edges is akin to that of the sequential graph, as described in Section 3.2.2, however the edges are grouped as *local*, where the process owns both endpoints, and *global*. The distinction is made for two reasons; mainly, it allows for the selection of a random locally-owned edge in O(1) time.

Furthermore, grouping edges as *local* or *global* simplifies the implementation: global edges may be temporarily removed from the graph, pending an answer for a matching requests, whereas local edges are only removed if they are included as part of the matching.

The data structure does not keep track of the local degree of vertices that are not owned by the current process. Thus, if a process owns two vertices with IDs i and j that are both adjacent to a third vertex k owned by some other process, p_k has to sent two **M_REMOVE_VERTEX** messages if k is removed from the graph. This leads to extra communication, but it frees p_k of maintaining a set of the processes owning vertices adjacent to k, and p_i from maintaining a local edge list for k.

6.4.2 Vertex sets and states

The implementation uses three vertex sets, as described in Section 3.2.2: *singletons* is the set of vertices of degree 1; *locals* is the set of vertices that have at least one locally owned neighbour, and is used when there are no singleton vertices left in a computation step; *nonzeros* is the set of all vertices of nonzero degree, and is used when neither singletons nor locally-owned edges are available.

When a vertex is temporarily removed from the graph, pending a matching request to another process, the vertex is removed from all three vertex sets, so that it is not selected when the graph is queried for a random vertex. As a vertex is only temporarily removed from the graph if it is either a singleton with a global neighbour or a vertex with only global neighbours, it may be added back into the graph, and to the set of singleton or nonzero vertices, by using its global edge count.

When permanently removing a vertex, its owner ID is set to -1, and it is referred to as *purged*.

6.4.3 Vertex queue

In order to add temporarily removed vertices back into the grpah, and to check whether a vertex is part of an ongoing matching request or not, the implementation uses a vertex queue that supports the following operations:

1) To determine whether a given vertex is part of the queue, in constant time.

2) To add or remove a vertex from the queue in constant time.

3) To iterate over all the vertices in the queue in time $O(|Q|)$, where $|Q|$ denotes the number of queued vertices.

4) If a given vertex is queued, perform a query to determine which vertex the corresponding matching request is for. Thus if two processes try to match the same vertex pair, the request can be accepted instead of sending an **M_MATCH_BUSY** message.

Operations 1, 2, and 3 are supported using a data structure analogous to the vertex set that was described in Section 3.2.2. Thus one array, L is used for looking up a vertex ID, while another array, Q, stores the queued vertices consecutively. The vertices in Q are stored as pairs of integers (i, j), where i is the local ID of the queued vertex owned by the current process, p_i, and j is the global ID of the vertex that p_i wants to match i with.

As was explained in Section 6.2.1, it is necessary to maintain two queues: the data structure therefore consists of three arrays: Q_1, Q_2, and L. Note that there is no need for two L arrays, as a vertex cannot be part of more than one matching request.

Chapter 7

Karp-Sipser using edge partitioning

This chapter details the Karp-Sipser algorithm parallelized with an edge partitioning approach. As was explained in Section 4.3, an algorithm based on edge partitioning is more complex than its vertex-partitioned counterpart, as multiple processors may own edges incident on the same vertex.

Section 7.1 gives an overview of the algorithm, while Section 7.2 describes its various parts in further detail. An analysis of the algorithm is given in Section 7.3. Section 7.4 describes the data structures used for handling the partitioned graph.

7.1 Overview

The algorithm works in three phases: each superstep consists of handling incoming messages, examining queued matching requests and examination of the local graph. Although these are essentially the same steps as for the algorithm described in Chapter 6, the message handling and queue processing is more involved, as several processors will need to share information about a single vertex. An overview of the algorithm is given in Algorithm 6.

As was explained in Section 4.3, the owner of a vertex may not be aware of all its adjacent vertices. In order for a vertex owner to determine at the earliest possible stage that a vertex has become a *singleton*, it will instead maintain a set of the processes that own edges incident on a vertex it owns, called *ghost processes*, and whether they own zero, one or many such edges.

Once the local edge count for a non-owned vertex v becomes one, a process thus sends an local singleton message to the owner of the vertex, p_v; when the local count becomes zero, an unsubscribe message is sent, and p_v removes the sender from its list of processes owning edges incident on v. If p_v detects that v has become a singleton vertex, and p_v does not own the only remaining edge, a global singleton message is sent to the owner of the remaining edge, p_u. Upon receiving the message, p_u matches v with its only remaining neighbour, u, if it has not been matched already.

7.2 The edge partitioned algorithm

As with the vertex-partitioned algorithm, it is divided into three phases: message processing, queue processing and local matching operations. As such, only the new elements are discussed in this section.

Algorithm 6 Overview of parallel Karp-Sipser algorithm with vertex partitioning

```
procedure PARALLEL KARP-SIPSER WITH EDGE PARTITIONING(Graph $G$)
    Current process: $p_u$
    $M_{local} := \emptyset$
    while termination not requested by master do
        for $i := 1 \to |messages|$ do
            handle message
            if messages[$i$].type $==$ master requests result then
                send $M_{local}$ to master and terminate
            end if
            if messages[$i$].type $==$ match ok then
                add edge to $M_{local}$
            end if
            if messages[$i$].type $==$ local singleton $u$ then
                Check if $u$ is a global singleton, notify sender if it is
            end if
            if messages[$i$].type $==$ unsubscribe from $u$ then
                Check if $u$ is a global singleton, hand over if necessary
            end if
        end for
        handle queue
        for $i := 1 \to MPR$ do
            if All locally owned vertices are exhausted then
                break
            end if
            Pick an edge $e = (u, v)$ from the local graph
            if $p_v = p_u$ then
                $M_{local} := M_{local} \cup \{e\}$
            end if
            if $p_v \neq p_u$ then
                Send a matching request to $p_v$, remove $u$ from local graph
                Add $u$ to queue
            end if
        end for
        if no activity then
            notify master
        end if
        if master notified by all processes of no activity then
            master process requests result
        end if
        bsp_sync()
    end while
end procedure
```

7.2.1 Vertex ownership

With several processes possibly owning edges incident on the same vertex, a mechanism of vertex *ownership* is employed for communication. Vertices are referred to as *pure* or *shared*, and *shared* vertices are further classified as *owned* or *ghost* vertices. A ghost vertex, as described in Section 4.3, is a vertex for which the current process owns one or more incident edges, but does not own the vertex:

1) *pure* Pure vertices are vertices for which only one process owns incident edges. As such, the owner may remove them from the graph without any communication taking place.

2) *owned* If several processes own edges incident on the same vertex, the processor owning the most such edges is designated as the owner. If no single process owns the most incident edges, ownership is assigned to the processor with the lowest ID. The owner of a vertex v records the process ID(s) of all processes having v as a ghost vertex, and keeps track of whether their local edge count for v is nil, one or many. These processes are refered to as *ghost processes.*

3) *ghost* The vertices for which a process owns incident edges, but it is not the owner. For ghost vertices, a process records the owner's process ID and notifies the owner when the local edge count is one or nil, so that it may be detected at the earliest point whether or not a vertex is a *singleton.*

7.2.2 Singleton vertices

For a *pure* vertex, determining whether it is singleton is trivial. For a *shared* vertex, the owner needs to know about the local edge counts on the ghost processes. A shared vertex is a singleton vertex if: 1) the number of ghost processes is zero and the local edge count is one; 2) or the number of ghost processes is one and the ghost process' local edge count for that vertex is one.

If a process p_u detects that u has become a singleton vertex, and its local edge count for u is zero, it sends a global singleton message to the last remaining process owning an edge incident on u, p_v. Upon receiving the message, p_v adds u to its set of singleton vertices. If v is not involved in an ongoing matching operation, p_v matches u with v the next time it examines its local graph.

7.2.3 Message types

The messages may be divided into two parts: the ones sent by the main algorithm, and those generated by the graph framework in order to maintain a synchronized graph state, such as detecting global singleton vertices. Message parameters are shown in parentheses:

Algorithm messages

EPM_IDLE (sender) Sent to the master process, indicating that *sender* did not perform any work in the previous superstep.

EPM_MATCH (u, v) Sent by p_v to p_u, requesting that u and v should be matched. A reply of EPM_MATCH_OK is sent if the matching succeeds. If u is involved in another matching request, EPM_MATCH_BUSY is sent, unless the pending request is to match u with v; if no reply is sent, the matching failed because u is already part of the matching.

EPM_MATCH_BUSY (u, v) Indicates that v was not available for matching with u because it was involved in another matching operation.

EPM_MATCH_NONOWNED (u) Sent by p_u to a random ghost process owning edges incident on u, requesting that u be matched with any adjacent vertex.

Sent if p_u ran out of edges incident on u. If the receiver does not own any vertex adjacent to u, the message is forwarded. An example where this message is needed is discussed in Section 7.2.4.

EPM_MATCH_FORWARDED (u, v, p_v) If a process $p_{(u,v)}$ owning the edge (u, v) receives an EPM_MATCH_NONOWNED message from p_v for the vertex u, but $p_{(u,v)}$ does not own the vertex u, it forwards the request to p_u. As p_u might not know about the vertex v and its owner, the information is sent along with the matching request. Upon receiving an EPM_MATCH_FORWARDED message, p_u handles it similarly to an EPM_MATCH message. An example where this message is needed is discussed in Section 7.2.4.

EPM_MATCH_OK (u, v) Indicates that v was available for matching, and that u and v are now matched and should be removed from the graph.

EPM_REQUEST_RESULT Sent by the master process to indicate that all processes should send their result and terminate execution in the following superstep.

EPM_RESULT (sender, size, matching) The (partial) matching sent to the master process.

Framework messages

EPM_GLOBAL_SINGLETON (u) Sent by p_u to the last process owning edges incident on u, i.e. if p_u's edge count for u is zero, the ghost process count is one, and an EPM_LOCAL_SINGLETON message has been received from the last ghost process. The receiving process immediately matches u with its only adjacent vertex, v; if v is part of an ongoing matching request, u is added to the vertex queue and matched if v becomes available; if not, u is permanently removed from the graph.

EPM_LOCAL_SINGLETON (sender, u) Sent to p_u, indicating that *sender*'s local edge count for u has been reduced to one.

EPM_REMOVE_VERTEX (u) Sent by p_u to all processes owning edges incident on u to notify them that u has been permanently removed from the graph.

EPM_UNSUBSCRIBE (sender, u) Notifies p_u that *sender* no longer owns any edges incident on u. Upon receiving the message, p_u removes sender from the list of ghost processes for u.

7.2.4 A special case

This section describes a special case where the need for the **EPM_MATCH_NONOWNED** and **EPM_MATCH_FORWARDED** messages is demonstrated.

Consider the graph shown in Figure 7.1. As neither u, v, nor x is a singleton, and neither p_u, p_v, nor p_x owns any edges incident on these vertices, a message of **EPM_MATCH_NONOWNED** is sent to a process owning incident edges, in this case either p_{blue} or p_{red}. A message of **EPM_MATCH_FORWARDED** is then sent to a vertex owner, which in turns sends a reply to the process originally sending the **EPM_MATCH_NONOWNED** message.

Given that the graph is sufficiently large compared to the number of processes, the need for **EPM_MATCH_NONOWNED** and **EPM_MATCH_ FORWARDED** messages should be limited. As such, a process only sends an **EPM_MATCH_ NONOWNED** message if it was not able to perform any matching operations in a given step, in order to avoid the extra message volume incurred when an intermediary process has to forward the matching request.

When sending an **EPM_MATCH_NONOWNED** message, a process may not receive a reply for four supersteps, while the normal queue data structure only allows vertices to be queued for two rounds. This is solved by having a second queue, *delayQueue*. It stores pairs of integers, as for the queue described in Section 6.4: the first integer, used for indexing, is the local ID of the queued vertex, while the second

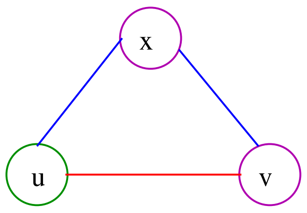

Figure 7.1: A graph consisting of a single cycle $(u, v)(v, x)(x, u)$, where the edges $(u, x)(v, x)$ are owned by p_{blue}, the edge (u, v) by p_{red}, vertices v and x are owned by p_{magenta} and u is owned by p_{green}, demonstrating the need for an **EPM_MATCH_NONOWNED** message.

integer is the number of the round in which the vertex should be added back into the local graph.

7.3 Analysis

The analysis of the edge partitioned algorithm consists of the same factors as that of the vertex partitioned algorithm, described in Section 6.3: the sequential work performed in a computation step, the communication volume and the number of supersteps. The expression for the number of supersteps is the same as for the vertex partitioned algorithm; however, as described in Section 4.3, the vertex partitioned algorithm should allow for a more even load balancing, which should decrease the number of supersteps.

7.3.1 Communication volume

An analysis of the communication volume for edge partitioned parallel matching was performed by Patwary et al. [6] for their implementation, where they obtained an upper bound for the communication volume, Vol, in terms of the communication volume for sparse matrix-vector multiplication:

$$Vol \leq \frac{3}{2} Vol_{SpMV} \tag{7.1}$$

The algorithm presented in this thesis differs in the communication volume in two ways:

1) When a process p_u accepts a matching request from p_v to match v with u, p_u will send an **EPM_REMOVE_VERTEX** message for u to p_v in addition to a message signalling that the match succeeded.
2) When a process p_u accepts a matching request from p_v to match v with u, p_u will send an **EPM_UNSUBSCRIBE** message for v to p_v in addition to a message signalling that the match succeeded.

This extra communication volume is redundant; however, it is the intention that the graph framework that was developed should be suitable for implementing other parallel algorithm, and adding special handling for matching operations would entail a loss of generality. It should be noted that in both cases described above, the extra messages are sent in the same round as their non-redundant counterparts. Assuming that the cost of a *bsp_sync()* operation is significant compared to the cost of sending a single message, the extra message volume should not impair the running time of the algorithm to a significant extent.

Given that a maximum matching has size at most $|V|/2$, at most $|V|/2$ redundant messages are sent by a vertex owner to a ghost process when accepting a matching request, and at most $|V|/2$ redundant messages are sent by a ghost process to a vertex owner when accepting a matching request. This gives the following communication bound:

$$Vol \leq \frac{3}{2} Vol_{SpMV} + |V| \tag{7.2}$$

7.3.2 Sequential work

The work performed in a computation step is greater for the edge partitioned algorithm than the vertex partitioned one, mainly for two reasons:

1) Each process maintains an adjacency list for its ghost vertices. Thus when a process p_u removes a vertex u that it owns from the graph, adjacent to a ghost vertex v owned by p_v, p_u needs to check whether its local edge count for v has been reduced to one or zero, and notify p_v accordingly.

2) The graph representation does not distinguish between *local* and *shared* edges; thus when a process p_u wants to match a vertex u with a random neighbour that it owns, it needs to iterate over u's adjacency list, costing $O(d_{avg})$ time instead of $O(1)$ as for the vertex partitioned algorithm. This is discussed further in Section 7.4.

7.4 Data structures

7.4.1 Graph representation

The data structures for the edge partitioned graph is based on the *EP_ Graph* framework developed by Martin Tofteberg [29] under the supervision of Prof. Dr. Fredrik Manne. As a work in progress, the data structures had to be modified in order to support the following operations:

1) Removing edges and vertices.

2) Changing vertex ownership, limited to the special case of changing the owner of a singleton vertex and permanently removing a vertex.

3) Removing ghost processes.

In the original format, the edges were stored using two arrays: one for the storing the edge endpoints, of size $2m_i$, where $m_i = |E_i|$ is the number of edges assigned to process i, and another array of size $n_i + 1$, where $n_i = |V_i|$. To accommodate 1), a new array, *currentDegree*, of size n_i was introduced to store the current degree of each vertex, as for the sequential Karp-Sipser implementation described in Section 3.2.2.

To accommodate 2), an array *newOwnerPid*, of size equal to the number of *pure + owned* vertices, was added.

For accommodating 3), the list of ghost processes was modified by adding a new array for storing the current ghost process count, analogous to the changes made for accommodating 1).

7.4.2 Vertex sets

The algorithm uses five vertex sets: the sets of singletons, vertices with local neighbours and vertices with nonzero local degree are used in a similar manner to the vertex partitioned algorithm. The fourth set is a set of *active* vertices, i.e. the vertices that have not been queued due to them being part of an ongoing matching request. The final set is the set of *non-purged* vertices; the set of vertices that have not been permanently removed from the graph.

This set is used in order to send *EPM_ MATCH_ NONOWNED* messages; these are sent when a process was not able to perform any local matching operations in a given round, for vertices that have local degree 0 but that are not global singletons.

Chapter 8

Experimental results

This chapter presents the experimental results of both parallel algorithms, as well as results of the sequential Karp-Sipser algorithm for comparison. Section 8.1 gives a description of the setup that was used to perform the experiments, while Section 8.2 gives an overview of the graphs that the algorithms were tested on. The experimental results follow: Section 8.3 presents results for the sequential algorithm, that are used when calculating the speedup of the parallel implementations; results from the vertex partitioned algorithm are given in Section 8.4; while results from the parallel algorithm with edge partitioning are given in Section 8.5.

8.1 Experimental setup

All three algorithm were implemented in C++ and compiled with the GNU GCC compiler version 4.8.0 using the -O2 optimization option. For the parallel versions, communication was provided by the BSPOnMPI library version 0.3, with the addition of thread safety and global memory extensions by Peter Krusche, as available from the BSPOnMPI SourceForge page [4]. BSPOnMPI was linked and run with the OpenMPI library, version 1.6.1.

The tests were executed on a system with 40 Intel Xeon E7 4850 cores with a clock frequency of 2.0 GHz, running 64 bit Linux (CentOS, kernel version 2.6.32) with 126 GB of total memory.

For the vertex partitioned algorithm, the input graphs were partitioned using the METIS partitioning software [2]. The SCOTCH graph partitioner [3] was tested in early experiments; however, the resulting differences in matching quality and running time were not found to be sufficient to merit a separate set of test runs. For the edge partitioned algorithm, the Mondriaan matrix partitioner [7] was utilized.

In the following tables, the number of matches performed in each round is designated by MPR. Negative values, such as an MPR of -500, indicates that the strategy of performing as many local matches as possible (described in Section 6.2.2) was used.

For each set of test parameters the algorithm was run twice, using the same random seed, and the run with the lowest execution time was used. This was done to alleviate the effects of sudden load changes that may have occurred due to system services and other background jobs running.

To compute the matching quality, a *maximum* matching for each graph was computed using the LEMON C++ library [37].

8.2 Data set

The data set (Table 8.1) was chosen to represent a diverse set of applications. *netherlands_ osm*, from the DIMACS10 implementation challenge, is a graph representation of a street network, and as such has a very low average vertex degree. *audikw_ 1* is a structural problem with a larger number of edges per vertex. *as-Skitter* is an Internet topology graph from Stanford Network Analysis Platform (SNAP), generated by running traceroute. These data sets were obtained from the University Florida Matrix Collection [32]

The data set also includes a random graph, previously generated by Fredrik Manne usin the GTGraph package [38]. Although the graph was symmetric, it was stored in a directed graph format, and was converted for the purpose of these experiments by discarding any edges with vertex IDs (i, j) where $i < j$.

Graph	Group	$\|V\|$	$\|E\|$	Avg. degree	$\|M_{\text{maximum}}\|$
netherlands_osm	DIMACS10	2,216,688	2,441,238	2.20	1,090,219
audikw_1	GHS_psdef	943,695	39,297,771	83.28	471,847
as-Skitter	SNAP	1,696,415	11,095,298	13.08	513,304
random21_16	Random	2 097 152	16 776 643	16.00	1,048,575

Table 8.1: The graphs used for performing the experiments. $|M_{\text{maximum}}|$ is the size of the maximum matching for the graphs, as calculated by LEMON.

8.3 Sequential algorithm

The results from the sequential algorithm are shown in Table 8.2. As can be seen from the matching quality, the algorithm produces a nearly maximum matching for all the graphs that were tested.

The running time is seen to increase with the number of edges in the graph; the graph *audikw_ 1* has approximately 20 times the number of edges as the graph *netherlands_ osm*, and the difference in running time is approximately 20.

Graph	Time	Matching size	Matching quality (%)
netherlands_osm	0.496921	1,086,107	99.623
audikw_1	11.497590	469,388	99.479
as-Skitter	2.631596	506,354	98.646
random21_16	4.831498	1,048,567	99.999

Table 8.2: Results for the sequential implementation.

8.4 Vertex partitioning

The results from the parallel Karp-Sipser algorithm using vertex partitioning are given in Tables 8.3, 8.4, 8.5, and 8.6 for p = 10, 20, 30, and 40 processes.

The tables show the total running time, the number of messages sent, the message volume in bytes, the communication time, the number of supersteps (*Rounds*) is used in the table for limiting column width), and finally the matching size.

This section focuses on describing the speedup and the matching quality, and how they are affected by the values of MPR and the number of processes. The number of rounds and communication time/volume are discussed in Section 8.6.

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	0.074868	1231	39392	0.00939353	7	1086031
10	500	0.088469	862	27584	0.022873	222	1086063
10	1500	0.08032	884	28288	0.015411	77	1086066
20	-500	0.0335591	2230	71360	0.00970085	7	1085956
20	500	0.046943	1567	50144	0.01737	112	1085923
20	1500	0.039102	1604	51328	0.010439	41	1086009
30	-500	0.0211871	3339	106848	0.00625397	7	1085956
30	500	0.035186	2307	73824	0.017872	77	1085963
30	1500	0.028056	2352	75264	0.011196	29	1085999
40	-500	0.0186779	3886	124352	0.0125031	7	1085847
40	500	0.03404	2852	91264	0.021554	60	1085980
40	1500	0.027183	2867	91744	0.014905	24	1085987

Table 8.3: Results with vertex partitioning for the graph netherlands_osm, $|V| = 2216688$, average degree = 2.20

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	0.775224	1917849	61371168	0.134255	83	469003
10	500	0.793072	1022717	32726944	0.158397	181	469287
10	1500	0.737932	1043861	33403552	0.113889	113	469288
20	-500	0.380827	3099897	99196704	0.05752	95	468917
20	500	0.354897	1633877	52284064	0.040089	112	469188
20	1500	0.354737	1701625	54452000	0.041058	79	469118
30	-500	0.320152	3983117	127459744	0.072481	77	468761
30	500	0.28961	2109107	67491424	0.052098	113	469139
30	1500	0.286617	2235301	71529632	0.050767	122	469007
40	-500	0.333343	4690341	150090912	0.111416	95	468645
40	500	0.305244	2509072	80290304	0.09358	101	469015
40	1500	0.314437	2708064	86658048	0.10447	113	468869

Table 8.4: Results with vertex partitioning for the graph audikw_1, $|V| = 943,695$, average degree = 82.28

8.4.1 Speedup

To get a clearer picture of the speedup and matching quality, the results are presented graphically. The speedup is shown in Figures 8.1, 8.2, 8.3, and 8.4 for different values of MPR. In Chapter 6, the hypothesis is that a higher value of MPR, or the use of strategy 2 from Section 6.2.2, could reduce the number of rounds, but at the potential cost of improper load balancing, leading to an overall longer running time.

This is indeed the observed behaviour of the algorithm. For all graphs except netherlands_osm (Figure 8.1), a higher value of MPR, or strategy 2, results in a lower speedup. netherlands_osm has a low average vertex degree compared to the other graphs, and as such the number of edges shared between processes is expected to be small, thus each process can do a large amount of local matching operations without any communication, a scenario in which matching strategy 2 is superior. The small number of shared edges for netherlands_osm can be confirmed by looking at the message count in Table 8.3.

The speedup is lowest for the graph random21_16; in the random graph, each edge is present with a constant probability, and thus a lower speedup compared to the real

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	0.415825	2273286	72745152	0.171701	71	497086
10	500	0.294541	1392141	44548512	0.111969	175	494028
10	1500	0.328393	1422185	45509920	0.128501	89	494267
20	-500	0.331269	2582765	82648480	0.199776	85	495200
20	500	0.217734	1539823	49274336	0.124738	148	492713
20	1500	0.224628	1605775	51384800	0.128043	99	492976
30	-500	0.364507	2808411	89869152	0.268464	93	494658
30	500	0.171872	1713863	54843616	0.109733	79	492373
30	1500	0.180195	1807789	57849248	0.115922	62	492910
40	-500	0.304243	2920349	93451168	0.227879	99	493592
40	500	0.18687	1816083	58114656	0.134608	97	491715
40	1500	0.194503	1936284	61961088	0.140316	72	492673

Table 8.5: Results with vertex partitioning for the graph as-Skitter, $|V| = 1,696,415$, average degree = 13.08

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	1.84	20768220	664583040	0.432526	58	998860
10	500	1.122928	12032264	385032448	0.188238	264	1037639
10	1500	1.211263	12145872	388667904	0.227864	116	1031838
20	-500	0.882883	22298927	713565664	0.173335	59	1001702
20	500	0.538406	13042310	417353920	0.087336	154	1036352
20	1500	0.55806	13278809	424921888	0.091037	80	1026417
30	-500	0.701434	22889756	732472192	0.172437	59	1003106
30	500	0.42536	13522464	432718848	0.094026	117	1033588
30	1500	0.439827	13881008	444192256	0.099757	70	1021085
40	-500	0.698865	23227289	743273248	0.254544	57	1004302
40	500	0.452538	13834350	442699200	0.163215	100	1030970
40	1500	0.467051	14311370	457963840	0.173549	66	1016914

Table 8.6: Results with vertex partitioning for the graph random21_16, $|V| = 2,097,152$, average degree = 16

world graph suggests that the structural properties of a graph has a profound effect on the performance of the algorithm.

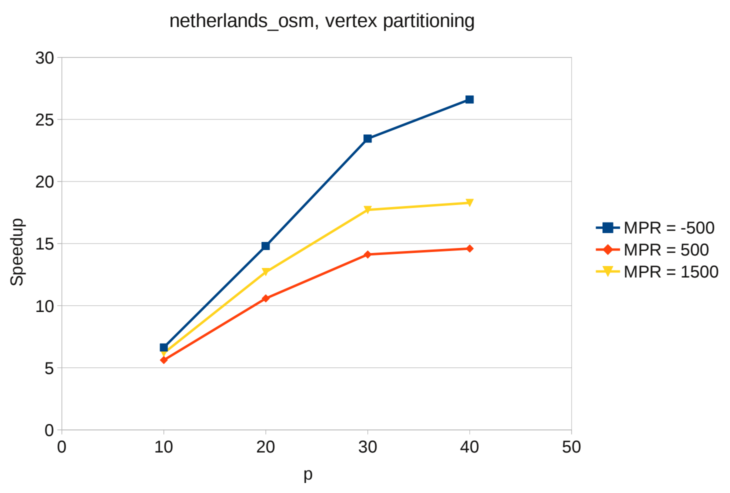

Figure 8.1: Speedup for the graph netherlands_osm with vertex partitioning

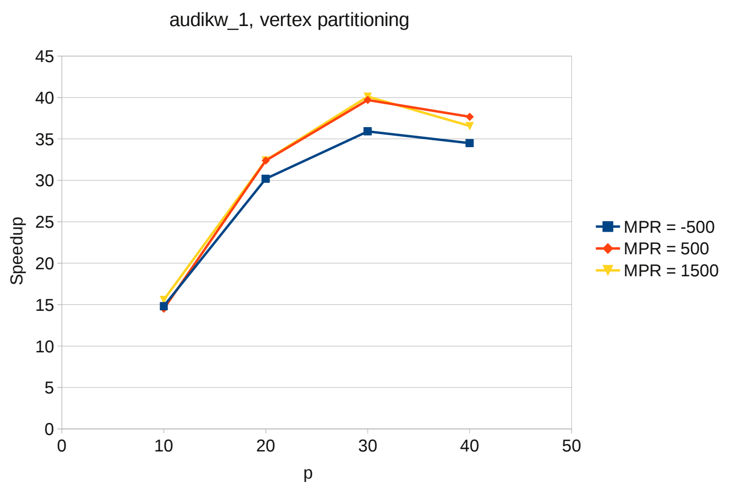

Figure 8.2: Speedup for the graph audikw_1 with vertex partitioning

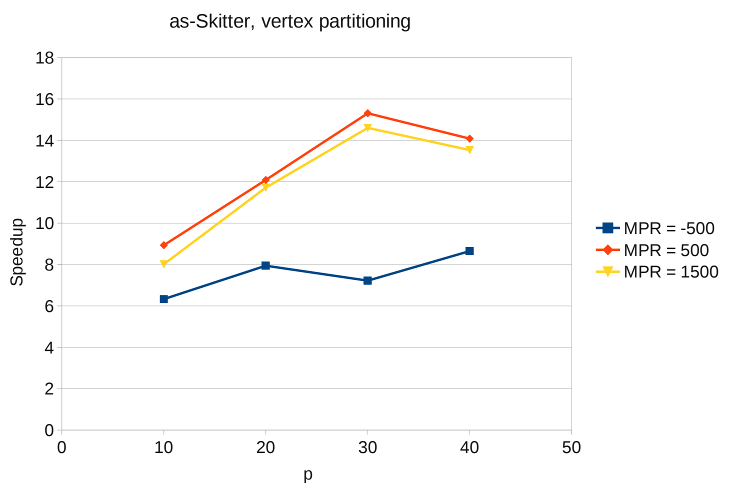

Figure 8.3: Speedup for the graph as-Skitter with vertex partitioning

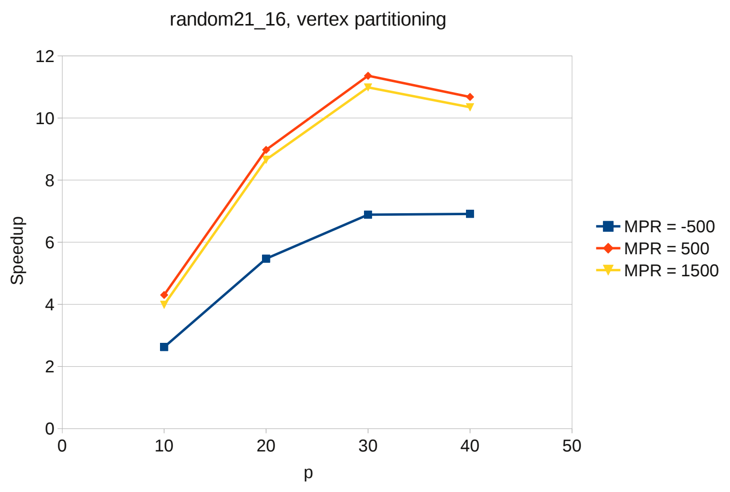

Figure 8.4: Speedup for the graph random21_16 with vertex partitioning

8.4.2 Quality

The hypothesis presented in Chapter 6 is that a higher value of MPR, or matching strategy 2 as described in Section 6.2.2, increases the number of local matching operations performed before a vertex is detected to be a singleton, thus reducing the matching quality.

The same holds for the number of processes: given that the local matching operations perform in each step can be described by $p \times MPR$, a higher process count should have a negative impact on the matching quality.

Examining the effect of MPR first, the following may be observed: for the graph netherlands_osm, there is no clear winner as for the choice of MPR and matching quality; however, all results are within 0.4 percentage points of the optimal solution, and as such any variations may be due to the random nature of the algorithm. For the graph audikw_1, a lower value of MPR clearly leads to a better matching quality; however, all results differ by at most 0.7 percentage points of the optimal solution.

The results for as-Skitter are special in that the higher value of MPR, and especially matching strategy 2, consistently produces a better matching quality. However, the difference in quality (for the values of MPR) is merely around 0.5 percentage points. The random graph produces the expected result: that a higher value of MPR, or especially matching strategy 2, produces a considerably lower matching quality (4 percentage points for $p = 10$ processes), although the difference is reduced as the number of processes increases.

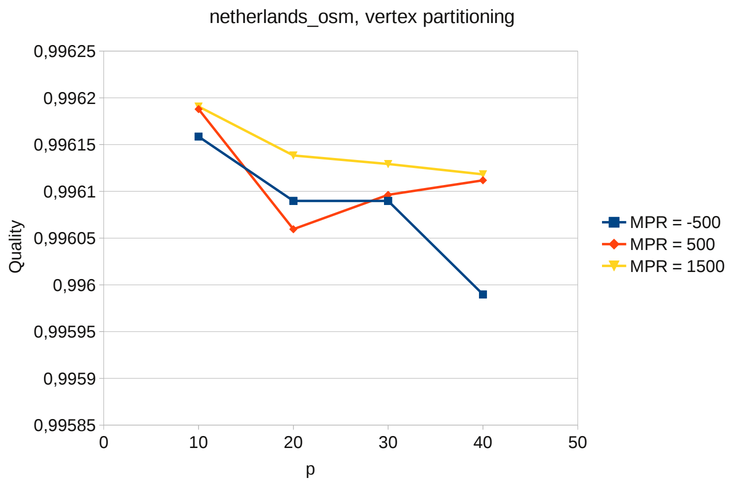

Figure 8.5: Quality for the graph netherlands_osm with vertex partitioning

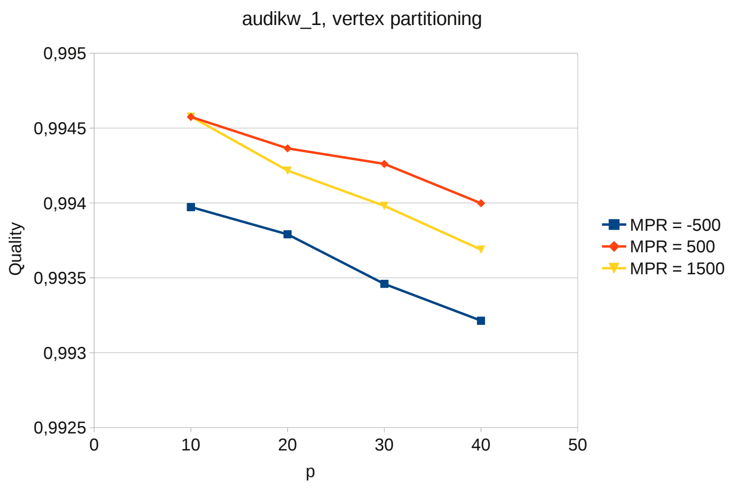

Figure 8.6: Quality for the graph audikw_1 with vertex partitioning

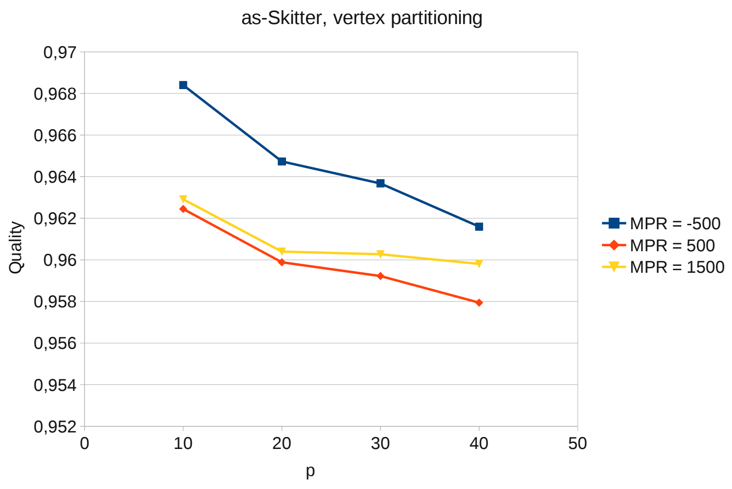

Figure 8.7: Quality for the graph as-Skitter with vertex partitioning

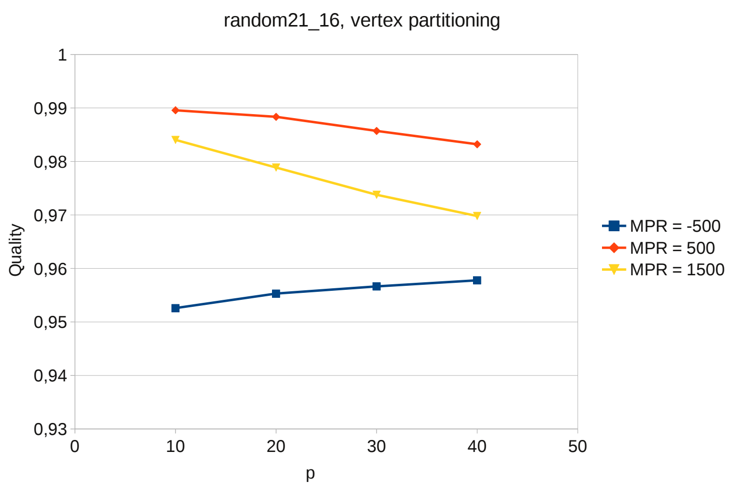

Figure 8.8: Quality for the graph random21_16 with vertex partitioning

8.5 Edge partitioning

The results for the edge partition based algorithm are give in Tables 8.7, 8.8, 8.9, and 8.10. for p = 10, 20, 30, and 40 processes.

To better illustrate the results, the speedup is shown graphically and discussed in Section 8.5.1, while a graphical representation and discussion of the matching quality is presented in Section 8.5.2.

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	0.566059	2178662	35457996	0.246382	14	958493
10	500	0.458841	2157491	37184724	0.136917	229	1083510
10	1500	0.455092	2160120	37179716	0.12109	86	1078960
20	-500	0.242866	2258779	36750540	0.087696	15	956720
20	500	0.21291	2255066	38837420	0.048398	119	1080880
20	1500	0.216439	2257656	38783948	0.041154	49	1072061
30	-500	0.17378	2289420	37231500	0.068864	13	956240
30	500	0.154285	2290065	39410292	0.039459	83	1078384
30	1500	0.15932	2294335	39351796	0.035254	35	1065844
40	-500	6.675678	2294603	37326400	6.63181	17	956170
40	500	0.156992	2300319	39561936	0.065302	66	1076046
40	1500	0.155852	2306127	39496672	0.055708	31	1059916

Table 8.7: Results with edge partitioning for the graph netherlands_osm, $|V| = 2,216,688$, average degree = 2.20

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	6.095041	158192	2531988	0.31195	9	469003
10	500	6.22226	74850	1200496	0.400557	102	469326
10	1500	6.235245	76261	1223188	0.373039	39	469313
20	-500	3.067785	266128	4259856	0.099699	9	468767
20	500	3.22782	130261	2087796	0.201737	72	469245
20	1500	3.188601	134310	2153664	0.171381	37	469116
30	-500	2.495847	333898	5344368	0.40669	12	468654
30	500	2.57654	165817	2652812	0.435869	104	469246
30	1500	2.556064	172052	2758452	0.427252	30	469003
40	-500	1.773914	397000	6353856	0.226824	12	468593
40	500	1.82948	194958	3125844	0.241285	33	469137
40	1500	1.813586	208134	3338992	0.233579	21	468821

Table 8.8: Results with edge partitioning for the graph audikw_1, $|V| = 943,695$, average degree = 82.28

8.5.1 Speedup

As described in Sections 7.3 and 7.4, maintaining a synchronized graph state for the edge partitioned graph requires extra work in each computation step, and in addition, the selection of local edges is not done in an optimal manner. However, edge partitioning should allow for better load balancing.

Although the speedups obtained are for the most part an order of magnitude

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	1.675946	1135631	18863612	0.795284	16	484433
10	500	2.16061	837435	13828468	1.231791	147	504587
10	1500	2.179157	866818	14325148	1.240654	61	504254
20	-500	0.834983	1273781	21145712	0.436326	21	482939
20	500	1.53156	979900	16161940	1.110773	130	503445
20	1500	1.577438	1025991	16967992	1.15436	45	502629
30	-500	0.932082	1391933	23081388	0.669083	19	481500
30	500	1.30441	1089274	17950124	1.022305	104	503004
30	1500	1.283835	1151492	19052472	1.00563	41	501555
40	-500	0.756817	1499303	24853940	0.554343	18	479034
40	500	0.97595	1196831	19748940	0.766329	102	502179
40	1500	0.887403	1273678	21104924	0.682353	40	499175

Table 8.9: Results with edge partitioning for the graph as-Skitter, $|V| = 1,696,415$, average degree = 13.08

p	MPR	Time	Messages	Bytes sent	Comm. time	Rounds	|Matching
10	-500	1.892022	9528723	153826796	0.444592	59	1028187
10	500	1.9819	8042404	130505276	0.415548	230	1038642
10	1500	1.857619	8108996	131592724	0.307976	103	1028194
20	-500	1.10598	12557275	203051408	0.283414	68	1029972
20	500	1.13055	11083372	179863236	0.22752	129	1032906
20	1500	1.139437	11262469	182860016	0.240786	68	1015618
30	-500	0.819532	14183380	229372568	0.201184	75	1023579
30	500	0.849227	12582519	203948736	0.178404	91	1026363
30	1500	0.842293	12899985	209407840	0.176506	54	1007486
40	-500	23.290801	15154479	245412676	20.66995	46	1019846
40	500	0.695007	13709598	222433128	0.145768	79	1021053
40	1500	0.700256	14159578	230291160	0.150923	72	1003762

Table 8.10: Results with edge partitioning for the graph random21_16, $|V| = 2,097,152$, average degree = 16

lower than those of the vertex partitioned algorithm presented in Section 8.4, they will nevertheless be described here.

For the graph netherlands_osm (Figure 8.9), a maximum speedup of 3.25 is obtained; the speedup is roughly equivalent for $p = 30$ and $p = 40$ processes. While there is little difference between MPR values of 500 and 1500, the matching strategy 2 (described in Section 6.2.2) produces a speedup of near zero for $p = 40$ processes. Looking at the Communication time in Table 8.7, it appears that all but one process stays mostly idle. As netherlands_osm has a very low average vertex degree compared to the other graphs, it does not provide for the same load balancing opportunities.

On the graph audikw_1 (Figure 8.10), the algorithm obtains a speedup of 6.5 using $p = 40$ processes; the choice of MPR has little effect on the overall running time, and the speedup is almost linear.

For as-Skitter (Figure 8.11), matching strategy 2 provides the best speedup: 3.5 using 40 processes, while an MPR value of 500 and 1500 show little difference in running time. The best speedup is obtained for the random graph (Figure 8.12): both for $MPR = 500$ and $MPR = 1500$, a speedup of 7 is obtained for 40 processes;

it should be noted that speedup is almost linear from $p = 10$ to $p = 40$ processes, indicating that the algorithm scales very well when the graph does not have any special structure.

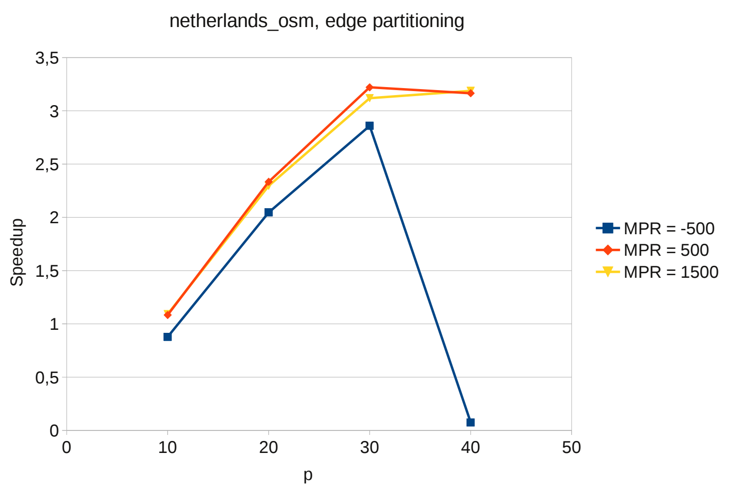

Figure 8.9: Speedup for the graph netherlands_osm with edge partitioning

8.5.2 Quality

Consider the matching quality for the graph netherlands_osm, shown in Figure 8.13. The strategy of perform as much local work as possible has a profound negative effect on the matching quality, giving a result that is around 12 percentage points below the optimal. For an MPR value of 500, the quality is within 1 ($p = 10$) to 2 ($p = 40$) percentage points of the optimal.

A quality within less than 0.6 percentage points of the optimal is obtained for the graph audikw_1 (Figure 8.14), with an MPR value of 500. For $p = 10$ processes, there is no difference in the quality between an MPR value of 500 and 1500, but for 40 processes an MPR value of 1500 produces a matching quality that is 1 percentage points lower.

For the graph as-Skitter (Figure 8.15), a matching quality of around 98 % is obtained with an MPR value of 500; the negative impact of a higher MPR value increases along with the number of processes. The matching quality obtained when the algorithm is run on the random graph is within 1 percentage point of the optimal for 10 processes, but decreases to roughly 97.5 % for $p = 40$, with an MPR value of 500.

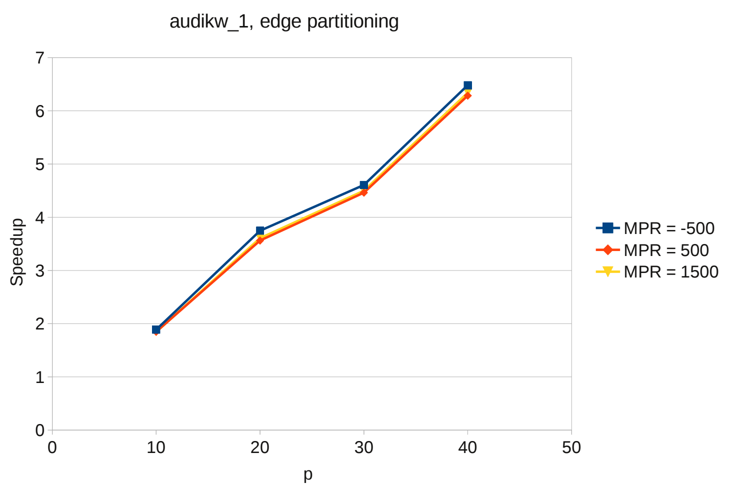

Figure 8.10: Speedup for the graph audikw_1 with edge partitioning

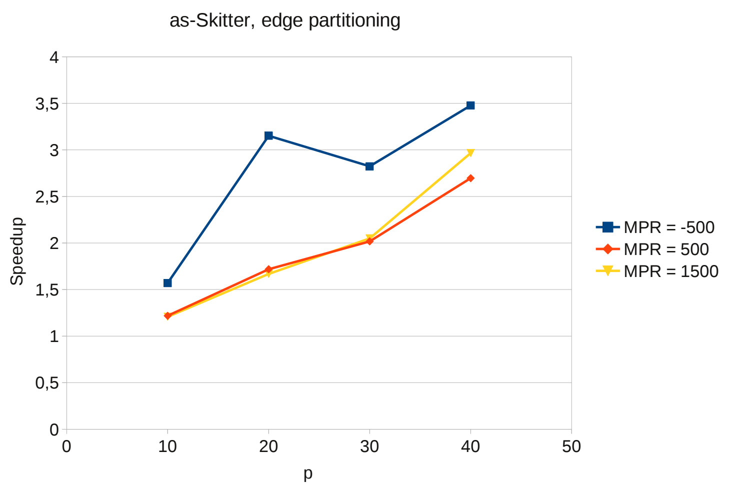

Figure 8.11: Speedup for the graph as-Skitter with edge partitioning

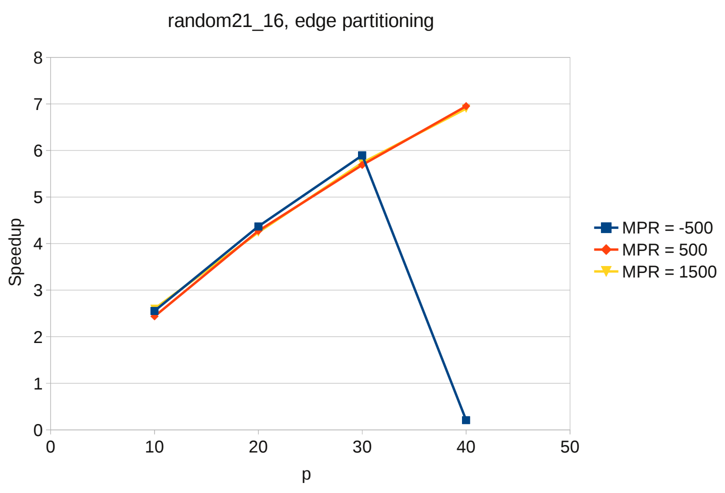

Figure 8.12: Speedup for the graph random21_16 with edge partitioning

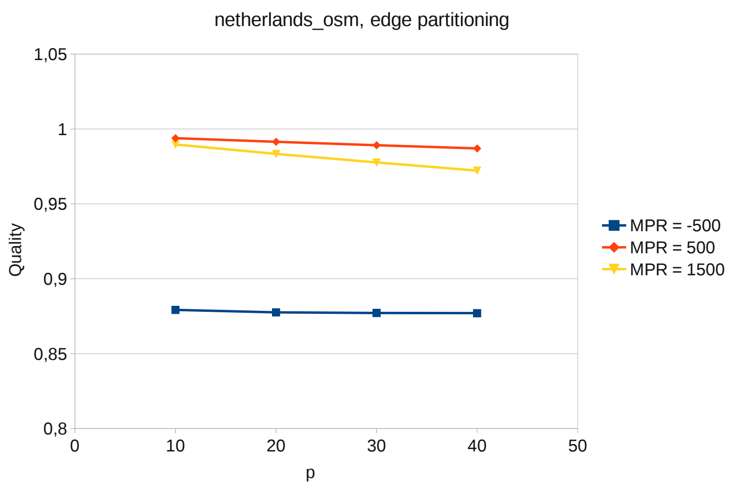

Figure 8.13: Quality for the graph netherlands_osm with edge partitioning

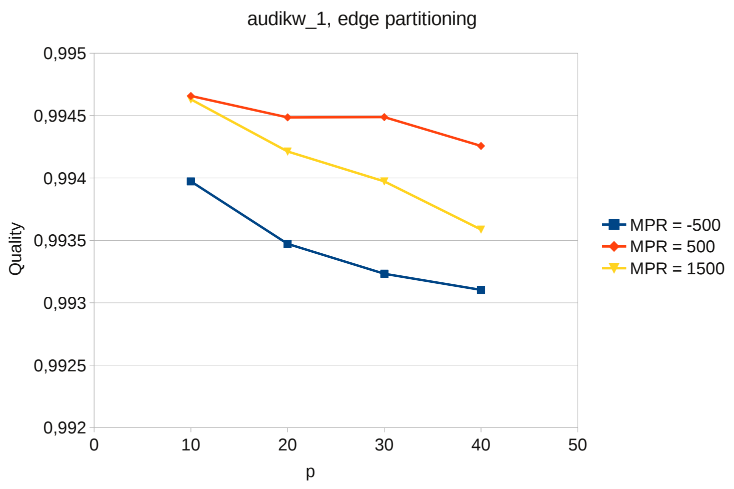

Figure 8.14: Quality for the graph audikw_1 with edge partitioning

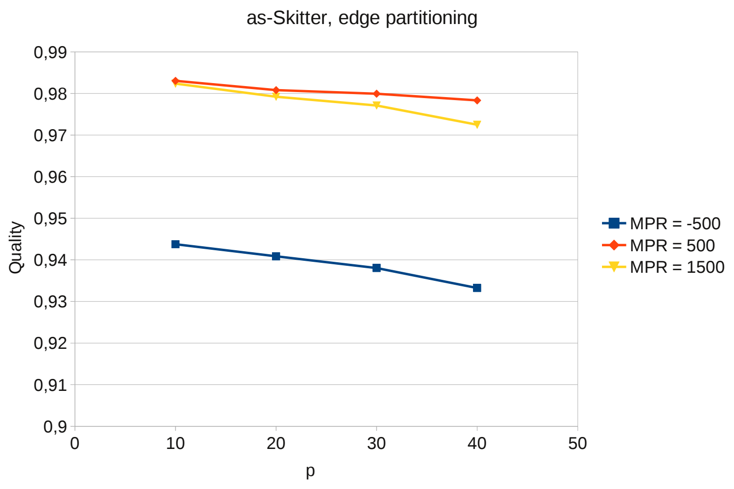

Figure 8.15: Quality for the graph as-Skitter with edge partitioning

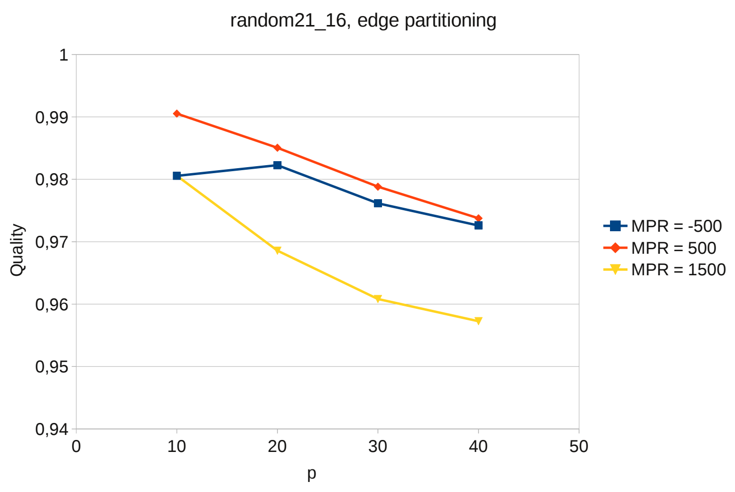

Figure 8.16: Quality for the graph random21_16 with edge partitioning

8.6 Comparison

This section compares the results for the vertex and edge partitioned algorithms. From the results presented in Section 8.4 and Section 8.5, it is obvious that the vertex partitioned algorithm is fastest, and as such no further comparison is performed. However, the lower speedup values for the edge partitioned algorithm may be due to inefficiencies in the implementation, as noted in Sections 7.3 and 7.4. As such, this section will compare the number of messages sent by each algorithm, the number of supersteps required to compute the matching and finally the communication time/total running time ratio, as an indicator of how well the algorithms scale with a higher number of processors.

The graphs used are audikw_1 and random21_16, as they were the ones where the edge partitioned algorithm performed most comparably to the vertex partitioned algorithm, both in terms of speedup and matching quality. The values plotted are those with an MPR of 500, which on average was the best setting for both algorithms.

As can be seen in Figure 8.17, the vertex partitioned algorithm sends a large amount of messages compared to the edge partitioned algorithm. For the random graph, however, the message volumes converge with an increasing number of processes.

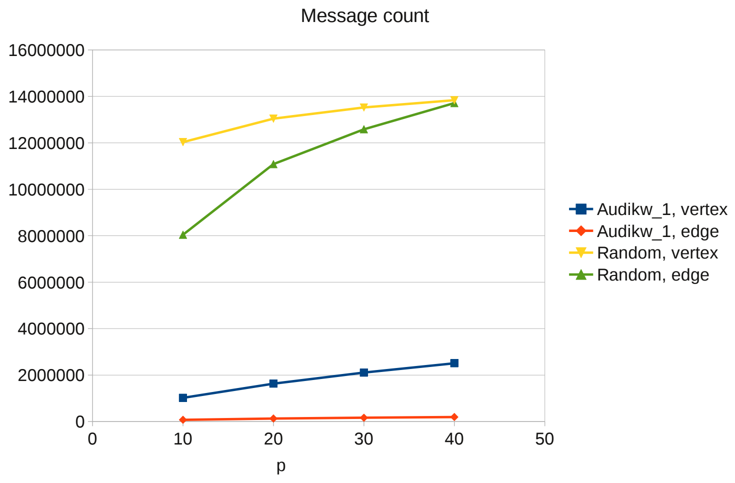

Figure 8.17: Message count for the graphs audikw_1 and random21_16 for both parallel algorithms, using an MPR value of 500.

The number of supersteps required by both algorithms for the graphs audikw_1 and random21_16 are shown in Figure 8.18. In general, the edge partitioned algorithmn requires fewer supersteps to compute a matching; for the graph audikw_1 using $p = 40$ processes, the edge partitioned algorithm requires approximately half the number of steps as its vertex partitioned counterpart.

The ratio between communication time and the total running time, shown in Figure 8.19 shows that the edge partitioned algorithm does not suffer from an increased communication burden when the number of processors is increased. However, this

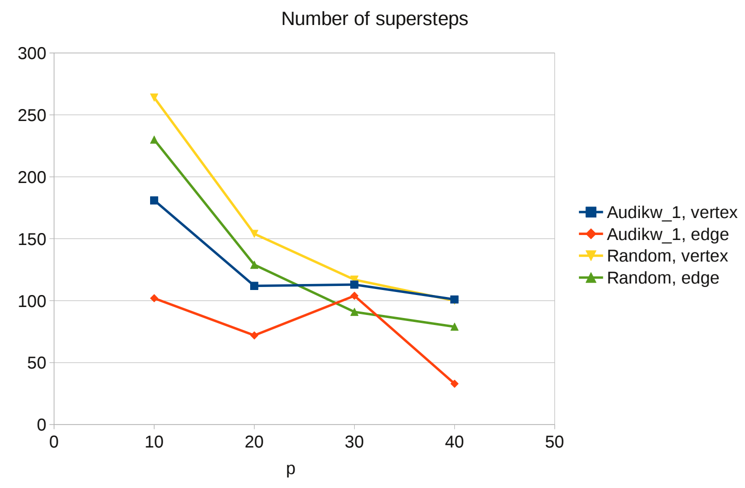

Figure 8.18: Number of supersteps for the graphs audikw_1 and random21_16 for both parallel algorithms, using an MPR value of 500.

may be due to algorithm performing an unnecessary amount of local computation in the first place.

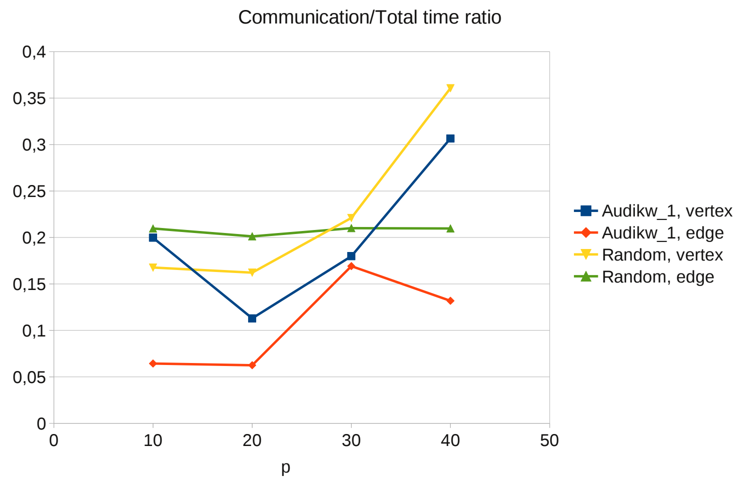

Figure 8.19: The ratio communication time / total running time for the graphs audikw_1 and random21_16 for both parallel algorithms, using an MPR value of 500.

Chapter 9

Conclusion

This chapter is structured as follows: Section 9.1 summarizes the findings made in this thesis, while Section 9.2 gives suggestions for further work that can be done, either by improving on shortcomings of the implemented algorithms, or by following up on new ideas.

9.1 Summary

The speedup obtained with the vertex partitioned algorithm (Section 8.4) is for some graphs an order of magnitude greater than that of the edge partitioned algorithm (Section 8.5); however, the matching quality offered by the two algorithms is comparable.

The edge partitioned algorithm obtains better speedups when the average vertex degree is higher; in hindsight, it might have been an idea to test the algorithms on graphs with a greater average degree than those presented in Table 8.1.

The experimental results show that the ratio of communication time vs total running time is favourable for the edge partitioned algorithm, as it does not increase significantly as the number of processes is increased. While this indicates that it offers better load balancing, the algorithm needs to be more efficient when performing operations on the local graph before any decisive conclusion can be made.

9.2 Further work

As noted in Sections 7.3 and Section 7.4, and as confirmed by the experimental results presented in Section 8.5, the *EP_ Graph* framework for an edge partitioned graph should be extended to ensure that the work done in each computation step is as efficient as possible. By separating the storage of *local* and *global* edges, a local neighbour to a vertex may be found in constant time, instead of a time proportional to the local vertex degree.

Given that the vertex partitioned algorithm, with a higher message volume, outperforms the edge partitioned one, it is worth investigating whether the *EP_ Graph* framework could be made more lightweight, by storing less information about the graph state locally and instead relying on redundant messages in some cases, such as by not maintaining an adjacency list for ghost vertices.

Alternatively, the algorithms should be tested on graphs with a higher average vertex degree; as summarized in Section 9.1, the edge partitioned algorithm performed comparably better for more dense graphs.

On another note, given that the algorithms have been implemented using an abstraction layer for communication, new experiments could be performed using the novel MulticoreBSP library [39], which avoids running a BSP library on top of MPI.

Chapter 10

References

[1] Karp, R. M., and M. Sipser: *Maximum matching in sparse random graphs.* In Proceedings of the 22nd Annual Symposium on Foundations of Computer Science, pp. 364-375. IEEE Computer Society, 1981.

[2] Karypis, George, and Vipin Kumar: *A fast and high quality multilevel scheme for partitioning irregular graphs.* SIAM Journal on scientific Computing 20, no. 1 (1998): 359-392.

[3] Pellegrini, François, and Jean Roman: *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs.* In High-Performance Computing and Networking, pp. 493-498. Springer Berlin Heidelberg, 1996.

[4] Suijlen, Wijnand J.: *BSPonMPI: An implementation of the BSPlib standard on top of MPI, version 0.3.* http://bsponmpi.sourceforge.net/ 2010.

[5] Bisseling, Rob H.: *Parallel Scientific Computation: A structured approach using BSP and MPI.* Oxford University Press. 2004.

[6] Patwary, Md Mostofa Ali, Rob H. Bisseling, and Fredrik Manne: *Parallel greedy graph matching using an edge partitioning approach.* In Proceedings of the fourth international workshop on High-level parallel programming and applications, pp. 45-54. ACM, 2010.

[7] Vastenhouw, Brendan, and Rob H. Bisseling: *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication.* SIAM review 47, no. 1 (2005): 67-95.

[8] Auer, Bas O. Fagginger, and Rob H. Bisseling: *A GPU algorithm for greedy graph matching.* In Facing the Multicore-Challenge II, pp. 108-119. Springer Berlin Heidelberg, 2012.

[9] Mucha, Marcin, and Piotr Sankowski: *Maximum matchings via Gaussian elimination.* In Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on, pp. 248-255. IEEE, 2004.

[10] Micali, Silvio, and Vijay V. Vazirani: *An $O(\sqrt{|v|}|E|)$ algoithm for finding maximum matching in general graphs.* In Foundations of Computer Science, 1980., 21st Annual Symposium on, pp. 17-27. IEEE, 1980.

[11] Shokoufandeh, Ali, and Sven Dickinson: *Applications of bipartite matching to problems in object recognition.* In Proceedings, ICCV Workshop on Graph Algorithms and Computer Vision, vol. 18. 1999.

[12] Berge, Claude: *Two theorems in graph theory.* Proceedings of the National Academy of Sciences of the United States of America 43, no. 9 (1957): 842.

[13] Hopcroft, John E., and Richard M. Karp: *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.* SIAM Journal on computing 2, no. 4 (1973): 225-231.

[14] Edmonds, Jack: *Paths, trees, and flowers.* Canadian Journal of mathematics 17, no. 3 (1965): 449-467.

[15] Coppersmith, Don, and Shmuel Winograd: *Matrix multiplication via arithmetic progressions.* Journal of symbolic computation 9, no. 3 (1990): 251-280.

[16] Williams, V. Vassilevska: *Breaking the coppersmith-winograd barrier.* Unpublished manuscript, Nov (2011).

[17] Robinson, Sara: *Toward an optimal algorithm for matrix multiplication.* SIAM news 38, no. 9 (2005): 1-5.

[18] Strassen, Volker: *Gaussian elimination is not optimal.* Numerische Mathematik 13, no. 4 (1969): 354-356.

[19] Korte, Bernhard, and Dirk Hausmann: *An analysis of the greedy heuristic for independence systems.* Algorithmic Aspects of Combinatorics 2 (1978): 65-74.

[20] Alt, Helmut, Norbert Blum, Kurt Mehlhorn, and Markus Paul: *Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/logn})$.* Information Processing Letters 37, no. 4 (1991): 237-240.

[21] Kaya, Kamer, Johannes Langguth, Fredrik Manne, and Bora Uçar: *Push-relabel based algorithms for the maximum transversal problem.* Computers & Operations Research (2012).

[22] Aronson, Jonathan, Alan Frieze, and Boris G. Pittel: *Maximum matchings in sparse random graphs: Karp-Sipser revisited.* Random Structures and Algorithms 12, no. 2 (1998): 111-177.

[23] Wilkinson, Barry, and Michael Allen: *Parallel Computers.* In Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, 3 - 42. Upper Saddle River: Pearson Prentice Hall, 2005. Pearson Prentice Hall (2005): 3-42

[24] Goldberg, Andrew V., and Robert E. Tarjan: *A new approach to the maximum-flow problem.* Journal of the ACM (JACM) 35, no. 4 (1988): 921-940.

[25] Setubal, João C.: *New experimental results for bipartite matching.* Proceedings of netflow93 (1993): 211-216.

[26] Langguth, Johannes, Md Mostofa Ali Patwary, and Fredrik Manne: *Parallel algorithms for bipartite matching problems on distributed memory computers.* Parallel Computing 37, no. 12 (2011): 820-845.

[27] Azad, Ariful, Mahantesh Halappanavar, Sivasankaran Rajamanickam, Erik G. Boman, Arif Khan, and Alex Pothen: *Multithreaded Algorithms for Maxmum Matching in Bipartite Graphs.* In Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, pp. 860-872. IEEE, 2012.

[28] Dufossé, Fanny, Kamer Kaya, and Bora Uçar: *Randomized matching heuristics with quality guarantees on shared memory parallel computers.* (2013).

[29] Tofteberg, Martin: *Parallel computing using edge partitioning.* Master's thesis, University of Bergen, 2011.

[30] Catalyurek, Umit V., Florin Dobrian, Assefaw Gebremedhin, Mahantesh Halappanavar, and Alex Pothen: *Distributed-memory parallel algorithms for matching and coloring.* In Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pp. 1971-1980. IEEE, 2011.

[31] Manne, Fredrik, and Rob H. Bisseling: *A parallel approximation algorithm for the weighted maximum matching problem.* In Parallel Processing and Applied Mathematics, pp. 708-717. Springer Berlin Heidelberg, 2008.

[32] Davis, T.A., and Y. Hu: *The University of Florida Sparse Matrix Collection.* ACM Transactions on Mathematical Software, Vol 38, Issue 1, 2011, pp 1:1 - 1:25. http://www.cise.ufl.edu/research/sparse/matrices

[33] Intel Corporation: *Intel Xeon Phi Product Family.* Accessed 6th December 2013. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[34] Cray Inc.: *Cray XMT.* Accessed 6th December 2013. http://www.cray.com/Assets/PDF/products/xmt/CrayXMTBrochure.pdf

[35] Message Passing Interface Forum: *Message Passing Interface Forum.* Accessed 6th December 2013. http://www.mpi-forum.org/

[36] Devine, Karen D., Erik G. Boman, Robert T. Heaphy, Rob H. Bisseling, Umit V. Catalyurek: *Parallel Hypergraph Partitioning for Scientific Computing.* IPDPS'06. IEEE. 2006.

[37] LEMON: *LEMON.* Accessed December 2013. http://lemon.cs.elte.hu/

[38] Bader, David A., and Kamesh Madduri: *GTgraph: A suite of synthetic graph generators.* http://www.cse.psu.edu/ madduri/software/GTgraph/

[39] MulticoreBSP: *MulticoreBSP homepage.* Accessed December 2013. http://www.multicorebsp.com/