Parallel Graph Algorithms for Combinatorial Scientific Computing

Md. Mostofa Ali Patwary

Dissertation for the degree of Philosophiae Doctor (PhD)

University of Bergen
Norway

February 2011

To my dear wife and friend - Kazi Nadira Parvez,
who stood by me all these years, with unconditional love, support, patience, and cooperation.

Acknowledgements

All praises is due to Allah, Lord of the World, who granted me the ability to finish the thesis. I remember my father who passed away after three months I started this study. May Allah keep him in eternal peace.

First, I would like to acknowledge the tremendous help and support that I received from my advisor, Fredrik Manne. The thesis would not have existed without him. His ideas, knowledge, and insights were always of great help to my intellectual advancement. My ideas were always clearer when leaving his office than they were before. I also thank him for his patience in reviewing my so many inferior drafts, suggesting new ways of thinking, and encouraging me to continue my research. He was also helpful in many non-academic matters whenever needed.

Thanks to Martin Vatshelle, my friend and colleague, for helping me in many practical things. I want to thank Daniel Lokshtanov and Jesper Nederlof for being my partner in playing table-tennis. Thanks to Johannes Langguth. We had many discussions on the ongoing research in our discipline and also on our research. Thanks to my co-advisor Pinar Heggernes, an always smiling face, Jean Blair, Daniel Meister, Fedor V. Fomin, Jan Arne Telle, Sadia Sharmin, Binh-Minh Bui-Xuan, Yngve Villanger, and all other members of the algorithm group in my department.

Thanks to Rob Bisseling, *hpc-europa2*, and SARA super computing center for facilitating my visit to Utrecht University, the Netherlands in January and in April, 2010.

My sincere thanks to Alex Pothen for his generous help during my visit to Purdue University, USA in July-December, 2010. I thank his family members for making the stay enjoyable and for their hospitality. Thanks also to Assefaw H. Gebremedhin, Ariful Azad, Duc Nguyen, and Sven Köhler.

Thanks to the Norwegian Research Council for their financial support and allowing the freedom of research. My sincere thank goes to Ida Holen, Maria Marta Lopez, and Tor M. Bastiansen for easing many of the administrative activities.

Finally, I am grateful to my parents, brothers and sisters, and all my relatives for their continuous supports and encouragement. Last, but not least, my heartfelt thanks to Kazi Nadira Parvez for her love, support, and endless cooperation in my life.

Bergen, February 2011
Md. Mostofa Ali Patwary

Contents

Introduction

Combinatorial algorithms have played a significant role in the solution of many scientific computing problems for a long time. The impact of these algorithms does not only include the application of graph algorithms in sparse matrix computations, but also mesh generation, optimization, computational biology, chemistry, physics, parallelization, and others. Although researchers working in these disciplines might be far apart in traditional scientific taxonomy, they acknowledge the importance of discrete algorithms and share common intellectual aesthetics, techniques, and interests. Therefore, they have recently been brought under common roof known as *Combinatorial Scientific Computing* (CSC). CSC is an interdisciplinary field which involves discrete mathematics in scientific computing and refers to the development, analysis, and application of combinatorial algorithms to solve problems in computational science and engineering. In this thesis, we present new results on various research problems in CSC, many of which are related to parallel computing.

This chapter is organized as follows. Section 1 presents an overview of CSC. Section 2 describes parallel computing and its importance in CSC. Section 3 discusses some of the graph problems in CSC along with those studied in this thesis. Finally, Section 4 presents a summary of our contributions and also list some open problems. Following this we present the research papers.

1 Combinatorial Scientific Computing

Scientific computing is traditionally considered to be a field of applied and numerical mathematics. But there are many combinatorial problems that arise naturally in scientific computing. Research on such problems has been pursued since the 1960s [29, 34, 103] and it has recently been recognized as a field of its own known as Combinatorial Scientific Computing [68, 88]. This formal labeling has increased the awareness of CSC among researchers and has lead to many conferences and workshops being organized over the last few years with broad international participation from academia, government laboratories, and industry.

CSC is contributing significantly in various fields of scientific computing, for example, partitioning, coloring and matching in graphs [57, 66, 112], decomposition in mesh generation [125], factorizations in numerical linear algebra [103, 115], non-linear optimization problems [34, 111], cellular automata in statistical physics [140], molecular analysis in computational chemistry [121], DNA and RNA alignment in bioinformatics [45], and information retrieval from complex networks in information processing [81, 102]. Many of these problems are also studied in the context of parallel computing [66, 68].

Many scientific applications can be modeled using combinatorial problems [57, 133] and for several of these problems, researchers have developed software packages. These packages can run on sequential computers, but, since most combinatorial problems consist of large data sets, many such packages have been adapted to run on parallel computers [38, 76]. These packages have seen widespread use and have had a significant impact in the scientific community. To understand this significance, consider the following. The problem of scheduling computational tasks so as to minimize computational dependencies arises in a number of areas, including derivative computation, frequency assignment in radio and wireless networks, scheduling, and concurrency discovery and data movement operations in parallel and distributed computing [55, 58]. The problem is often modeled using a graph where the vertices represent computational tasks and the edges represent dependencies. A coloring of this graph can in many instances be used as a routine to identify the independent subtasks that can be accomplished concurrently. The coloring assigns positive integers (colors) to the vertices of the graph such that adjacent vertices have different colors. Tasks (vertices) with the same color can then be performed concurrently. The number of colors used is equivalent to the required computational steps and is therefore expected to be as small as possible. Obtaining such a graph coloring is a pure combinatorial problem and researchers have developed several software packages to solve both the pure coloring problem [38, 57], as well as several variations of it [74, 75, 129]. Another such example is the problem of multiplying a sparse matrix by a vector. This operation is the most computationally expensive part of iterative methods of linear systems and eigensystems. When this problem is carried out on a parallel computer, one need to partition the matrix in such a way that each processor gets an almost equal share of data elements while at the same time minimizing the required communication. Obtaining such a partitioning is also a pure combinatorial problem and several software packages have been developed to solve this problem [38, 74, 75, 129]. Such packages, whether for coloring or partitioning, are sophisticated and complex, still they are easy to use, runs fast, and produce high quality solutions in many cases. The best way to observe their impact is how they are referenced in scientific publications where researchers would previously develop and explain their own software in great detail, while now they will just use such a package and cite it.

As stated in [68], research in CSC typically comprises of the following three steps. The first step consists of finding an appropriate combinatorial model for a problem in scientific computing to make the computation feasible, fast, and efficient. This is the most time consuming step, as developing the right combinatorial model is often critical to the computation of an efficient solution. The second step involves the design, analysis, and implementation of algorithms to solve the combinatorial subproblem. The emphasis is always on practical and efficient algorithms since an algorithm with quadratic time complexity in the input size could be too slow to be useful compared to the other computational steps. The algorithm could either be an exact, approximate, or heuristic solution to the problem. The third and last component involves the development of software, evaluating its performance on a collection of large test sets, making it publicly available, and integrating with a larger software library if possible.

Since CSC typically involves time consuming computations on large data sets, it is only natural that one employs parallel computers and algorithms for faster processing.

2 Parallel Computing

Parallel computing has a long history within scientific computing. The hardware used for this has mainly been large, expensive, and specially built computers that could only be acquired by larger organizations. However, with the fairly recent appearance of multicore computers in the mass market, parallel computing is now becoming mainstream technology. All projections indicate that this trend will continue and escalate in the foreseeable future. It is estimated that the number of cores per chip will be doubling almost every two years while the production cost remains fixed [8]. Combining these chips into larger multi-processor systems is also becoming increasingly affordable [118]. But, being able to exploit the performance gain from these computers depends very much on algorithms and software. Therefore, there is an essential need to develop parallel algorithms that can take advantage of the parallelism that is now becoming available. As stated, scientific computing often uses parallel computers and algorithms for faster processing. To avoid that the combinatorial parts become bottlenecks, these have to be parallelized as well. In the following we give a brief overview of the different types of parallel computers and their related programming models.

Parallel computers are typically classified as either shared memory or distributed memory computers, depending on how the memory is organized. We start by presenting shared memory computers.

2.1 Shared Memory Parallel Computers

With the proliferation of multicore chips, shared memory computers are today by far the most common type of parallel computers. They range from laptops to high speed supercomputers. Although these computers vary widely, what they have in common is that multiple processors share the same memory as a global address space. The processors operate independently and any change in a memory location made by one processor is immediately visible to all other processors. This makes the task of sharing data fast and uniform and provides a user friendly programming environment. Although the typical number of cores is relatively modest, for example dual-core and quad-core, this number is expected to increase rapidly.

Along with the hardware development, programming models are also needed to obtain high performance from these computers. As there is no universally accepted parallel programming language, the main choice has been to develop APIs to extend existing programming languages in order to express parallelism.

OpenMP [26] and POSIX Threads [64, 139] are two such widely used APIs for programming shared memory computers. OpenMP is available for C/C++ and Fortran and is based on compiler directives using `pragma` statements in the code. This enables the programmer to explicitly define parallelism in an existing sequential program. In particular, loop based parallelism can easily be exploited [79]. We now discuss briefly how to use OpenMP.

OpenMP programs execute serially until they encounter the `parallel` directive. This directive creates a group of threads that executes in parallel with the original sequential thread as the master thread. The threads then execute a structured block of code in parallel and once finished all the threads except the master are killed automatically and the program continues sequential execution.

There are several directives that specify how iterations and tasks are executed in parallel [26, 100]. Loop based directives are used to split an iteration space among multiple threads. How the iteration space is partitioned can be controlled using the `schedule` directive. An example is `static` scheduling which splits the iteration space into equal size chunks and assign them to threads in a round-robin fashion. But, different iterations might have widely varying execution times. For this reason, OpenMP gives the `dynamic` scheduling method where chunks of the iteration space are dynamically assigned to idle threads. Common to all loop based directives is that all threads perform the same kind of operations, but on different data sets. OpenMP also supports non-iterative parallel task assignment using the `sections` directive. With this, different code blocks are assigned to different threads. There are also ways to declare variables as `private` to each thread or `shared` among all threads. However, one must be careful when using shared variables as multiple threads might try to write to the same variable at the same time, resulting in inconsistencies. However, there are directives that help with this task. The `critical` directive allows one thread at a time to execute a critical region (a block of code) and the `atomic` directive allows a thread to update a memory location at a time. OpenMP also supports a number of library functions for controlling and monitoring the threads, such as, for setting the default number of threads to be created in a parallel region or for getting the thread ID. It is worthwhile to note that in OpenMP, the programmer does not need to create, join, or kill threads by explicit commands.

On the other hand, POSIX Threads, also known as *Pthreads*, is a library based model and requires explicit parallel coding. This provides flexibility but the programmer has to deal with communication, threads, and synchronization himself. Therefore, this model is not so popular with application programmers who typically do not want to deal with low level primitives [64]. We omit the details of Pthreads as we will be using OpenMP in this thesis when dealing with shared memory computers.

Threading Building Block (TBB) is another parallel programming approach for shared memory computers, which was recently introduced by Intel [114]. TBB is a task-based abstraction for parallel programming that at runtime maps tasks to threads. Tasks are relatively light-weight compared to threads and no context switching is needed. However, a recent study [79] indicates that TBB requires a considerable program redesign compared to OpenMP. Therefore, this approach might be less appropriate for parallelizing existing sequential codes.

So far, we have only talked about the features of multicore computers and how these are programmed. But, it is also useful to have a theoretical model so that one can analyze the performance of a parallel program as one can for a sequential program. The Parallel Random Access Machine (PRAM) is one such computational model for shared memory computers [80]. In its standard form, the model assumes an unbounded number of processors and a shared memory of unbounded size. However, it is typically assumed that the number of processors is polynomial in the input size. Still, this should be compared to real applications where the number of processors is typically several orders of magnitude lower than the input size. Despite this serious deviation from the nature of real parallel computers, the PRAM model can be useful for theoretical runtime analysis in order to get an idea about the parallelism inherent in an algorithm, leaving the communication and synchronization costs aside.

2.2 Distributed Memory Parallel Computers

In this type of architecture, processors are connected through a common communication channel and each processor has its own local memory. Conceptually, one can view this as if several desktop computers have been connected through a network to operate as one parallel computer. Therefore, memory scales well with the number of processors. These computers are traditionally large, consisting of up to thousands of processors and having terabytes of memory. One advantage of such systems is that they are less complex to build and also, more scalable. Today, systems executing many teraflops/second are not uncommon [5, 134]. Currently, the worlds fastest computer, the Chinese Tianhe-1A system at the National Supercomputer Center in Tianjin is such a system and can execute 2.57

petaflops/second with 229,376 gigabytes of memory [127]. However, the largest systems are very expensive and therefore, they are only used in national labs and large companies to conduct complex tasks such as nuclear weapons simulations, pharmaceutical drug modeling, mining of large data sets, weather forecasting, or geological analysis to find oil deposits [134].

In distributed memory systems, each processor operates independently and changes that one processor makes to its own local memory has no effect on the memory of other processors. When data has to be relayed between processors, the programmer has to specify explicitly how and when the data is communicated between the processors. An extra challenge with distributed memory computers is that it might be difficult to map an existing sequential data structure to the different processors. Therefore, designing parallel algorithms for distributed memory computers is typically more challenging than it is for shared memory computers. Similar to shared memory computers, APIs are used for programming these computers. In the following we outline some of the programming approaches.

Interaction between processors is usually accomplished using messages, hence the name *message passing* computers. The exchange of messages is used to transfer data, tasks, or to synchronize actions among the processors. The basic operations, the sending and receiving of messages are performed by `send` and `receive` operations (the corresponding call may differ across APIs). In addition, since the send and receive operations must specify target addresses, there must be a mechanism to assign a unique identification (ID) to each processor executing a parallel program. The ID is typically made available to the program using a `whoami` function. The final function needed to complete the basic set of message passing operations is `numprocs`, which specify the number of processors participating in the ensemble. The Message Passing Interface (MPI) [65] is the de facto standard message passing library used to develop portable message passing programs using either C/C++ or Fortran. MPI supports all the basic operations as well as a variety of higher level functionality. In total, the MPI library contains over 125 routines, but it is possible to write fully functional message passing programs using only six routines.

As communication is the main obstacle to obtaining efficient parallel code, we now discuss how this can be done. The two main ways of doing this is either as synchronous and asynchronous operations. *Synchronous* message passing requires the sender and receiver to wait for each other at a synchronization point to transfer the complete message. In this way there is no need for buffer storage. Synchronous message passing is also known as *blocking* message passing as it blocks computation until the message passing has been completed. On the other hand, *asynchronous* message passing delivers a message from the sender to the receiver, without waiting for the receiver to be ready. The advantage in this case is that both the sender and the receiver can overlap the time spent on communication with computation. Therefore, this is also known as *nonblocking* message passing since communication does not block further computations. The objective of this is to reduce the overall execution time. Buffers are used internally to perform this technique and therefore, at a later point in the program, a processor that has started a nonblocking send or receive operation must make sure that this operation has completed before it proceeds with its computations. In this thesis, we have mainly used asynchronous message passing when using distributed memory computers.

So far, we have discussed the features of distributed memory computers and how to program these using message passing. We now explore a model, known as the *Bulk Synchronous Parallel* (BSP) model, to design parallel algorithms for these computers [16, 132]. BSP serves a similar purpose as the PRAM model for shared memory computers, but differs from PRAM by not taking communication and synchronization for granted. A BSP algorithm consists of a sequence of supersteps. Each superstep contains first a computation step which is then followed by a communication step. Finally, there is a global barrier synchronization. In the computation step, each processor performs a sequence of operations on local data whereas in the communication step, each processor sends and receives a number of messages. At the end of a superstep, each processor synchronizes by checking whether it has finished all its obligations in the current superstep before proceeding to the next one.

One advantage of the BSP model is that the execution time of an algorithm can be predicted by theoretically analyzing the cost of the algorithm and independently measuring performance parameters of the computer when using a fixed number of processors. These parameters are the single-processor computing rate, the time taken by one processor to send or receive one data word, and the time taken to synchronize all processors. The predicted time will then give an upper bound on the measured time. We refer to [16] for more details about the BSP model.

There are currently three libraries, the Oxford BSP Toolset [69], the Paderborn University BSP library [18, 19], and recently introduced `BSPonMPI` [120], that implements the BSP model [70] (also known as BSPlib). These libraries take some of the tediousness away from message passing. Moreover, much of the communication optimization is left to the system; for instance, different messages to the same destination are automatically detected and combined. However, the BSP model is not that widely used and most established libraries that implement it are not platform independent. Another issue is that the BSP model, as stated, does not take advantage of overlapping computation and communication.

In this thesis we have used both pure BSP type programming and also a hybrid approach. In the pure BSP type programming, we have used `BSPonMPI` which is platform independent and developed on top of MPI. In the hybrid approach we have combined elements from the pure asynchronous communication model with the programming style of BSP. Thus we are able to overlap communication with computation while at the same time taking advantage of the structured BSP style of programming. The main difference from pure BSP is that we use asynchronous MPI send and receive operations and that we start these operations as soon as possible during the computation step. In this way we are able to overlap communication with computation and as soon as a processor reaches the end of the computation step, all data that the processor expects to receive should ideally already have arrived. The processors receive and process all incoming messages in any order, unlike the BSP model where all messages are received before being processed. The processors do not explicitly synchronize with each other and move on to the next superstep once they are done with the communication. However, since we typically use an all-to-all communication pattern, there is an implicit synchronization between the processors. In this setting, a processor could be one superstep ahead of the other processors, but not more. Thus, one can view our algorithms as following a relaxed BSP style.

2.3 Other Types of Parallel Computers

We note that there exists other types of parallel computers than the pure shared memory and distributed memory computers discussed so far. For instance the *Hybrid Distributed-Shared Memory Computers* connect several shared memory computers under a common message passing communication network and therefore, have both shared memory and distributed memory functionalities. We omit the details of these types of machines as they are not considered in this thesis.

Besides the use of traditional processors or cores in parallel computing, researchers have over the past few years started using Graphic Processing Units (GPUs) to do general purpose scientific and engineering computing. This branch of computing is recognized as GPU computing or GPGPU. The model for GPU computing is to use one CPU and several GPUs together in a heterogeneous computing model where the sequential part of the application runs on the CPU whereas the computationally intensive part is accelerated by the GPUs. As expected, GPU functionality has traditionally been very limited, and therefore, they can only be used efficiently to solve problems where similar computations are performed for each computing element. But there is an ongoing effort, for example by NVIDIA [98], to make these chips fully usable for scientific applications and also to add support for high level languages.

2.4 Performance Metrics

We now present a few performance metrics that are typically used to analyze the performance of parallel programs.

Speedup: While evaluating a parallel program, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is one such measure. Speedup using p processors is defined by $S_p = T_s/T_p$, where T_s and T_p are the execution times of the best sequential algorithm and the parallel algorithm, respectively. The ideal case, $S_p = p$, is known as *linear speedup*. However, in practice it is possible to get $S_p > p$ (known as *superlinear speedup*). One reason why this can appear is that the accumulated size of the caches from the p processors can fit more of the data set compared to what one processor can. Thus, the memory access time reduces drastically giving extra speedup in addition to the linear speedup.

Efficiency: Efficiency is a measure of the fraction of time for which a processor is performing actual computation. This is defined by $E_p = S_p/p = T_s/(pT_p)$. This value is typically between zero and one. It is used to estimate how well the processors are utilized in solving a problem, compared to how much performance is lost due to communication and synchronization.

Scalability: The scalability of a parallel program is a measure of its capacity to increase speedup in proportion to the number of processors. Thus it reflects a parallel programs ability to utilize increasing processing resources effectively. Typically we use two types of scalability. *Strong scalability* measures speedup when the number of processors varies, while the problem size remains fixed. To measure *weak scalability*, we vary the number of processors but now we also change the problem size so that the amount of work per processor is kept fixed.

3 Graph Problems in CSC

A graph is an abstract representation used to model many scientific computing problems. The vertices of a graph often represent computational tasks whereas the edges between the vertices reflect interdependencies between the tasks.

Graph algorithms have for a long time been a key aspect in a wide variety of areas in scientific computing. For example, graph partitioning has been used in sparse linear algebra since the 1960s [103]. Graph coloring algorithms have been used for computing Jacobian and Hessian matrices in optimization since the 1970s [29, 34]. Various types of matching problems in graphs are found in the solution of sparse linear systems, for instance in numerical preprocessing, block triangular decomposition, and also in the coarsening phase of multilevel methods for graph partitioning [43, 61, 109]. Graph and hyper-graph based models can capture the needs for reordering the nodes and elements within an unstructured mesh to improve runtime performance [119, 125]. Graph algorithms also play an important role in computational chemistry, biology, and in bioinformatics, for example, characterizing molecules [113, 121], genomic and proteomic analysis

[15, 40, 99], aligning DNA and RNA sequences and generating phylogenetic trees [13, 45], and cellular automata in statistical physics [126, 140].

Since many graph problems in scientific computing consists of large data sets, it is only natural that researchers have used parallel graph algorithms to solve these problems. In the following we elaborate on some of the issues that arise when these data sets are mapped to the memory of parallel computers.

As stated, in a shared memory parallel computer, multiple processors share the same global memory, therefore, one does not need to partition the data set before running a parallel algorithm. However, a parallel algorithm must specify which processor should work on what data elements. A proper partitioning enhances the performance of the algorithm significantly as shown in [56]. There are several ways of doing this, as discussed in Section 2.1. On the other hand, for distributed memory computers, since each processor has its own limited memory and since large data sets cannot fit in the memory of one processor, one needs to partition the data among the processors. Partitioning should typically be done in such a way that each processor gets an almost equal share of data elements, while at the same time minimizing the required communication. Achieving such a partitioning can typically be modeled as a graph problem and has been studied since the 1960s [103]. A common approach to perform this task is to partition the vertices and then assign each resulting part to a processor (while assigning the edges in a corresponding manner). An important objective is then to minimize the number of edges shared by multiple processors. This is often referred to as *graph partitioning* (also known as *vertex partitioning* or *one-dimensional partitioning*). There are several software packages such as Chaco [67] and Metis [75] to do this. An alternative approach to vertex partitioning is *edge partitioning* where the edges are partitioned instead of the vertices. This is often referred to as a *two-dimensional partitioning*. Several software packages like Mondriaan [133] and Zoltan [38] are available for edge partitioning. It has been shown that in the case of sparse matrix vector multiplication in numerical algorithms, two-dimensional partitioning can lead to lower communication volume compared to one-dimensional partitioning [130, 133]. For graph algorithms, to our knowledge, two-dimensional partitioning has not been employed yet. Thus, one of our intentions in this thesis has been to investigate whether an edge partitioning approach yields similar benefits as in the matrix vector multiplication case. For computing the edge partitioning, we use the Mondriaan [133] software package.

More specifically, we have studied three graph problems motivated from CSC. The first one is an advanced data structure used to model disjoint sets. In our work we have developed parallel algorithms for this problem along with highly tuned sequential algorithms. We have also studied parallelization of the matching problem and of the graph coloring problem in this thesis. In the following we present a brief overview of these problems.

3.1 Disjoint Sets

A *minimum spanning tree* (MST) for an edge weighted connected graph is a tree that contains all the vertices of the graph and where the sum of the weights of the edges in the tree is minimum. The problem of finding an MST is one of the most studied combinatorial problems with a number of practical applications. This includes areas such as VLSI layout, wireless communication, and distributed networks [93, 128, 143], recent problems in biology and medicine such as cancer detection [23, 77, 78, 91] and medical imaging [4, 40, 99], and national security and bioterrorism such as detecting the spread of toxins through populations in the case of biological/chemical warfare [27]. The computation of an MST is often a key step in other graph problems [90, 95, 124, 135]. Note that if the graph is not connected, then the MST problem finds a minimum spanning forest which consists of an MST for each component of the graph.

Kruskals algorithm is one of several ways to solve the MST problem [31]. In order to solve the problem efficiently, this algorithm uses a special kind of data structure known as a *disjoint-set data structure*.

A disjoint-set data structure maintains a dynamic collection of disjoint sets that together cover all the elements of a finite universe. Each set is typically represented by a rooted tree where the root is used as the *representative* of the set. Each node (each element of the universe) has a parent pointer which can be used to reach the root of the tree. The two main operations on this data structure are to FIND which set a given element belongs to and to replace two existing sets with their UNION. In addition, there is a MAKESET operation to create a singleton set from an element. This set of algorithms is often referred to as the UNION-FIND algorithm. The algorithm has been studied since the 1970s [12, 122, 123], is taught in most algorithm courses and used in standard software libraries such as Boost [36] and Leda [94].

Other than the computation of MSTs or connected components in graphs, the UNION-FIND algorithm is also used in image processing. An important problem in image processing is to capture the essential features of a scene [30, 138, 145]. One way to do this is to extract significant regions from the image [97]. The technique for extraction is known as *region growing* (also known as *connected component labeling*) which consists of starting with the smallest regions (i.e. pixels or points in an image) and merging them until they are considered to be optimal. The region growing problem is an important part of most applications in pattern recognition and computer vision, such as character recognition [3, 25]. In many cases, region growing is one of the most time consuming tasks in pattern recognition algorithms [2]. However, it has been shown that the region growing problem leads naturally to the disjoint set problem [39] and therefore, UNION-FIND algorithms have been used in image processing for a long time [9, 47, 141].

Another application of the UNION-FIND algorithm is the computation of an *elimination tree* from a sparse, symmetric, and positive definite matrix in numerical linear algebra [86, 144]. The elimination tree provides structural information relevant to the sparse factorization process [87, 110]. The use of the elimination tree includes, among others, finding equivalent matrix reorderings [108], various sparse storage schemes [86], and symbolic factorization [60]. The elimination tree is usually defined through the Cholesky factor (in which, nonzero elements above the diagonal have been transformed to be equal to zero) [59]. However, to construct the elimination tree directly from the sparse matrix efficiently, different variations of Union-Find algorithms have been tried and found to be efficient [62, 86, 87, 144].

As stated, the Union-Find algorithm is used as a basic building block of many applications, therefore, it is of interest to have an efficient implementation. There are several well established ways to implement the Union-Find algorithm efficiently [31]. But, because of the broad use of the algorithm in numerous applications, researchers from different disciplines have suggested different simple enhancements suitable for their own specific applications [62, 87, 101]. Although it is possible to combine several of these enhancements together giving more than 50 different variations of the Union-Find algorithm, most of the comparative studies on the Union-Find algorithm consider only a few algorithms in their context. For example, [87] and [62] compared only two and six algorithms, respectively, in sparse matrix factorization and [136] and [141] compared eight and three algorithms, respectively, in image processing. The most extensive such study was [71] which compared the performance of 18 variations of the Union-Find algorithm for computing connected components in graphs. However, there has been no study showing that these techniques work well in all settings. Therefore, it is of interest to study and compare all of these algorithms and suggests a best way of implementing the Union-Find algorithm in general. Note that such a study must be experimental since the theoretical complexity of all these suggested variations is more or less the same.

In this thesis we present such a study where we consider the Union-Find algorithm for finding connected components of a simple undirected graph $G = (V, E)$ where V and E are the set of vertices and edges, respectively. In this case, the Union-Find algorithm computes a minimal subset $S \subseteq E$ such that S is a spanning forest of G. If G is weighted and the edges are processed in order of increasing weight then the Union-Find algorithm gives Kruskals algorithm [31] for computing a minimum weight spanning forest.

In our work we performed a large set of experiments comparing over 63 different variations of the Union-Find algorithms using both real world and synthetic data as input. The difference between the algorithms goes typically to how the trees are constructed and traversed. Details on these can be found in [105]. Our main conclusion was that the standard implementation using the techniques known as Path Compression and Union by Rank (PCUR) is not the method of choice [106]. It was consistently outperformed by a somewhat forgotten simple method known as Rem's algorithm. This is an interesting observation as the PCUR algorithm is taught in most standard algorithm courses and implementations of it is also found in major algorithmic software libraries such as Boost [36] and Leda [94]. Thus, we expect our results to have an impact beyond the scientific community.

We also present a framework for parallelizing the Union-Find algorithm on a distributed memory computer. Although the algorithm is inherently sequential, there has been some previous efforts at constructing parallel implementations [7, 11, 35]. These have mainly been focused on shared memory computers. As far as we know, our algorithm is the first such effort for distributed memory parallel computers. To partition the graph among processors, we use the edge based partitioning software Mondriaan [133]. Our algorithm operates in two stages. In the first stage, each processor computes a spanning forest based only on local edges. This stage can be performed without any communication. We do this to reduce the number of edges to be considered in the final stage. In the second stage, we run a parallel algorithm to pick a subset of edges from these spanning forests to obtain the global spanning forest. We have tested several variations of the algorithm using both real world and synthetic graphs and found the framework to be scalable with reasonable speedup. Details can be found in [89].

3.2 Matching

A matching in a graph is a pairing of adjacent vertices such that each vertex is matched with at most one of its neighbouring vertices. An important task in many scientific computing problems is to compute a matching that is largest according to some objective function. One such objective is to find a matching with a maximum number of edges, known as the *maximum cardinality matching.* A variant known as the *maximum weighted matching* problem asks for the heaviest matching in an edge weighted graph, where the weight of a matching is the sum of the weights of the matched edges. Another variant is *vertex weighted matching* where weights are assigned to vertices instead of edges and therefore, the problem is to find a matching such that the weight of the endpoints of the matched edges is maximum.

Our interest in studying matching problems stems from their broad use in scientific computing. One example application is in numerical pivoting in the direct solution of a linear system of equations [43]. It has been shown in [43, 44] that the greedy strategy which is typically used for choosing pivots could lead to suboptimal results. It has been demonstrated that a better result can be achieved by viewing the matrix as a bipartite graph and finding a matching in the graph. The vertices in the bipartite graph correspond to rows and columns of the matrix, edges between row vertices and column vertices represent nonzero elements in the matrix, and the magnitudes of the nonzero elements become the weights of the edges. Then, a matching of maximum cardinality with the maximum weight can be used to permute the rows and columns of the matrix so as to place large elements on the diagonal and thus, reduce the need for pivoting.

Another application of matchings is in the iterative solution of linear systems of equations where one would like to have the dominant elements on the diagonal in order to ensure rapid convergence. This problem can also be modeled as a matching problem [116]. Other applications include the use of matchings in the development of multi level graph and hyper-graph partitioning software [37, 38], aligning RNA sequences [13], and studying magnetism [14].

Although many applications consider weighted matching, cardinality matching is often used as the first step towards the computation of a weighted matching [43, 44]. Moreover, there are applications where cardinality matching is needed, for example, in computing the block triangular form (BTF) of a sparse matrix. BTF leads to savings in computational work and intermediate storage for many sparse matrix algorithms, including algorithms for solving linear systems of equations and partitioning sparse matrices in parallel computations [109].

There exists several algorithms for computing a maximum cardinality matching. Common to most of these is that they are iterative in nature. Among these, the Push-Relabel algorithm [63] has proven to be fast and robust in practice [28]. However, recent studies have shown that many matching algorithms can be accelerated significantly by using strong initialization heuristics [42, 83], such as the Karp–Sipser algorithm [73]. Infact, new results indicate that the fastest matching codes use the Karp–Sipser algorithm and the Push-Relabel algorithm [41].

The Karp–Sipser algorithm has been shown in practice to yield high-quality matchings quickly [96]. The idea of the algorithm is as follows. It repeatedly picks a *singleton* vertex (degree one vertex) if there is any and if there is no such vertex it picks an edge randomly from the current graph. The algorithm then matches the two vertices (in case of a singleton vertex, the neighbour is the adjacent vertex) and removes them from the graph along with their adjacent edges. This removal might generate new singleton vertices. This process is then repeated until there is no edge left in the graph.

In this thesis we investigate the parallelization of the Karp–Sipser algorithm for distributed memory computers. We develop a new parallel algorithm which works in rounds. During every round, each processor tries to match a fixed number of vertices using the idea of the sequential Karp–Sipser algorithm (pick a singleton vertex if there is one, otherwise pick an edge randomly). This might require tie breaking if several processors want to match the same vertex. Just like the sequential algorithm, the parallel algorithm stops when there is no edge left on any processor. We have performed experiments with our parallel algorithm using both real-world and synthetic graphs, and found that the algorithm gave good speedup and matching quality on up to 64 processors. In this work, we also performed a comparative study between the performance of edge based partitioning and vertex based partitioning. Our results show that edge based partitioning requires less communication compared to vertex based partitioning. This is similar to parallel matrix vector multiplication in numerical algorithms. Details of the algorithm can be found in [104].

We also present the first practical parallel algorithm for computing a maximum cardinality matching in a bipartite graph suitable for distributed memory computers. The presented algorithm is based on the Push-Relabel algorithm. Bipartite matching is a special case of the maximum flow problem [46] and parallelizations of the Push-Relabel algorithm for maximum flow on shared memory computers have previously been presented in [6, 10]. [6] has later been adapted to bipartite matching and studied in [117]. The Push-Relabel algorithm tries to push units of flow from the left sided vertices to the right sided ones in the bipartite graph. If a push has been performed successfully between a pair of vertices, they are considered to be matched. This indicates how to parallelize the algorithm. Multiple processors could simultaneously try to push flow from left sided vertices to the right sided ones. In our work we show that a straightforward adaptation of the sequential Push-Relabel algorithm to a parallel version is not scalable due to the amount of communication. We then modify this algorithm in order to reduce communication and increase load balance. We then experiment on this new algorithm using several large problem instances and observe the scalability of the algorithm using up to 128 processors. To the best of our knowledge, this algorithm is the first exact algorithm for this problem to achieve speed up over the sequential algorithm. We note that having a parallel algorithm makes it possible to solve instances that are too large to fit in the memory of one processor. Details can be found in [84].

3.3 Graph Coloring

Coloring is an abstraction for partitioning a set of objects into independent sets where the objects in each set receive the same color. The notion of independence and the associated coloring rules vary from context to context. In the simplest case, adjacent vertices in a graph are required to receive different colors. This is known as *distance-1* coloring. There are several other coloring variations depending on their applications in scientific computing. One such variation is *distance-2* coloring where the colors of both adjacent vertices and distance-2 neighbours have to be different. A *star coloring* is a distance-1 coloring where, in addition, every path of four vertices uses at least three colors. An *acyclic coloring* is a distance-1 coloring in which every cycle uses at least three colors. The objective of most graph coloring problems is to minimize the number of colors used.

Graph coloring problems arise in a number of applications in scientific computing. Examples include, among others, timetabling and scheduling [85], frequency assignment [48], register allocation [24], printed circuit testing [50], parallel numerical computation [1], and optimization [29]. Unfortunately, coloring a general graph with the minimum number of colors is known to be an NP-hard problem [49] and also difficult to approximate [32]. Therefore, one often relies on heuristics to obtain a reasonable solution. Among these, greedy algorithms have been found to be quite effective [29]. A greedy algorithm incrementally colors the vertices of a graph and once a vertex is colored, its color never changes [51]. There are various strategies to assign colors to the vertices, for example, smallest legal color or using a random color. However, the order in which the vertices are processed in a greedy coloring algorithm has a large influence on the number of colors used by the algorithm [82, 142]. Therefore, it is also of importance in greedy coloring to choose the vertex ordering carefully. A number of different ordering techniques can be found in [29, 57, 92, 137].

There has been several studies on parallelizing greedy coloring algorithms both for distributed memory computers [17, 21, 22, 72] as well as for shared memory computers [52, 53, 56]. As far as we know, there has not been any previous studies on how to parallelize the ordering algorithms as these are sequential by nature and thus challenging to parallelize. We also note that there has not been any comprehensive study of how to choose permissible colors in greedy coloring on shared memory computers. The only case that we are aware of is [54] which presented a randomized variation of greedy coloring to reduce the number of color conflicts. In this thesis, we propose a framework for parallelizing the ordering on shared memory computers. The framework orders several vertices of the same degree in parallel, instead of one by one as in sequential ordering. This relaxation still shows similar performance compared to sequential ordering. Different ordering techniques have been investigated in this framework and experimental results show that the ordering algorithms are scalable. Like ordering, coloring is also inherently sequential. We also present a framework for parallelizing *distance-k* coloring on shared memory computers. The framework is similar to [20, 22, 131], but in addition, it employs a number of new ingredients. It works in rounds where each round has two phases. In the first phase, we assign a speculative color in parallel to the vertices. In the second phase, we identify any coloring conflicts and proceed to the next round to recolor the set of conflicting vertices. Different strategies in selecting a permissible color have been employed in this framework and experimental results show that the coloring algorithms are scalable. We also observe that using these frameworks, the number of colors used is fairly close to the corresponding sequential algorithms. Details can be found in [107].

4 Conclusion

In this section we list the papers in this thesis and also point to possible directions for how this work can be extended in the future.

4.1 Our Contribution

This thesis consists of the following five publications [84, 89, 104, 105, 107].

I. Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne, *An Experimental Evaluation of Union-Find Algorithms for the Disjoint-Set Data Structure*, Submitted to ACM Journal of Experimental Algorithmics, November, 2010. A preliminary version has been published in [106].

II. Fredrik Manne and Md. Mostofa Ali Patwary, *A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers*, in proceedings of the Eighth International Conference on Parallel Processing and Applied Mathematics (PPAM 2009), Springer LNCS 6067, pp. 186-195, 2009.

III. Md. Mostofa Ali Patwary, Rob H. Bisseling, and Fredrik Manne, *Parallel Greedy Graph Matching using an Edge Partitioning Approach*, in proceedings of the Fourth ACM SIGPLAN Workshop on High-level Parallel Programming and Applications (HLPP 2010), pp. 45-54, September, 2010.

IV. Johannes Langguth, Md. Mostofa Ali Patwary, and Fredrik Manne, *Parallel Algorithms for Bipartite Matching Problems on Distributed Memory Computers*, Submitted to Parallel Computing, October, 2010.

V. Md. Mostofa Ali Patwary, Assefaw H. Gebremedhin and Alex Pothen, *New Multithreaded Ordering and Coloring Algorithms for Multicore Architectures*, Submitted to the 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), 2011. A presentation on this has been given at ICCS Workshop on Manycore and Accelerator-based High-performance Scientific Computing, 2011.

Besides the above mentioned papers, the author has co-authored the following paper [57] during the work on this thesis.

VI. Assefaw H. Gebremedhin, Duc Nguyen and Alex Pothen, and Md. Mostofa Ali Patwary, *ColPack: Graph Coloring Software for Derivative Computation and Beyond*, Submitted to ACM Transactions on Mathematical Software, October, 2010.

Although this paper belongs in the area of CSC, it is not included in the thesis. The reason for this is that it is part of a larger project which has been ongoing for several years [33].

4.2 Open Problems

In this section, we present some of the different ways in which the work in this thesis could be extended.

The experimental study on different techniques of the sequential Union-Find algorithm is fairly complete. But one natural question arises here: which of the algorithms are parallelizable and which are worthwhile to try and implement? In our framework for parallelizing the Union-Find algorithm, we only experimented with a subset of the possible algorithms. In particular, it could be interesting to try a few more techniques including the best one (Rem's algorithm).

Another way to extend this work could be to use the Union-Find algorithms on shared memory computers. Even though there has been previous studies on how to do this, these only considered the standard techniques. In particular, it would be interesting to see how Rem's algorithm can be implemented on shared memory computers. It could also be of interest to compare our algorithm with other parallel algorithms for computing connected components.

In case of the matching problem this thesis only considered the maximum cardinality problem. One natural extension would therefore be to look at how our algorithms can be modified to deal with weighted matching. This is true both for greedy matching in general graphs as well as for perfect matching in bipartite graphs.

One more pending and interesting problem is to merge the parallel algorithms for greedy and perfect matching together. In our current parallel code for perfect bipartite matching, we compute an initialization locally on each processor and then extend this partial matching using our parallel algorithm. The runtime of this later step depends on the quality of the initial matching. Thus, since our parallel greedy graph matching algorithm gives a good quality matching quickly, it is likely that one could use it to speedup the computation of the perfect matching.

In case of the coloring problem, we have only considered general graphs. But from an application point of view, distance-2 coloring is typically used on bipartite graphs. Therefore, this would be a natural extension of this work. Moreover, it could be of interest to see how the ideas of our ordering algorithms can used on distributed memory computers.

Bibliography

[1] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin, *A comparison of parallel graph coloring algorithms*, Tech. Rep. SCCS-666, Northeast Parallel Architecture Center, Syracuse University, 1995.

[2] H. Alnuweiri and V. Prasanna, *Parallel architectures and algorithms for image component labeling*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 14 (1992), pp. 1014–1034.

[3] A. Amin and S. Fischer, *A document skew detection method using the Hough transform*, Pattern Analysis and Applications, 3 (2000), pp. 243–253.

[4] L. An, Q. Xiang, and S. Chavez, *A fast implementation of the minimum spanning tree method for phase unwrapping*, IEEE Transactions on Medical Imaging, 19 (2000), pp. 805–808.

[5] D. M. B. Ancajas, *Trends in supercomputing.* www.eecs.berkeley.edu.

[6] R. Anderson and J. C. Setubal, *A parallel implementation of the push-relabel algorithm for the maximum flow problem*, Journal of Parallel and Distributed Computing, 29 (1995), pp. 17–26.

[7] R. J. Anderson and H. Woll, *Wait-free parallel algorithms for the union-find problem*, in proceedings of the 23rd ACM Symposium on Theory of Computing (STOC 91), 1991, pp. 370–380.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, *A view of the parallel computing landscape*, Communications of the ACM, 52 (2009), pp. 56–67.

[9] D. A. Bader, J. JáJá, D. Harwood, and L. S. Davis, *Parallel algorithms for image enhancement and segmentation by region growing, with an experimental study*, The Journal of Supercomputing, 10 (1996), pp. 141–168.

[10] D. A. Bader and V. Sachdeva, *A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic*, in proceedings of the 18th International Conference on Parallel and Distributed Computing Systems, 2005, pp. 41–48.

[11] D. J. Bader and G. Cong, *A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)*, Journal of Parallel and Distributed Computing, 65 (2005), pp. 994–1006.

[12] L. Banachowski, *A complement to Tarjan's result about the lower bound on the complexity of the set union problem*, Information Processing Letters, 11 (1980), pp. 59–65.

[13] M. Bauer, G. W. Klau, and K. Reinert, *Fast and accurate structural rna alignment by progressive lagrangian optimization*, in Computational Life Sciences, M. R. Berthold, R. Glen, K. Diederichs, O. Kohlbacher, and I. Fischer, eds., vol. 3695 of LNCS, Springer Berlin/Heidelberg, 2005, pp. 217–228.

[14] R. J. Baxter, *Exactly Solved Models in Statistical Mechanics*, Academic Press, 1982.

[15] S. Bhowmick, E. Boman, K. Devine, A. H. Gebremedhin, B. Hendrickson, P. Hovland, T. Munson, and A. Pothen, *Combinatorial algorithms enabling computational science: Tales from the front*, Journal of Physics: Conference Series, 46 (2007), pp. 453–457.

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

[17] E. Boman, D. Bozdağ, Ümit V. Çatalyürek, A. H. Gebremedhin, and F. Manne, *A scalable parallel graph coloring algorithm for distributed memory computers*, in proceedings of the 11th International European Conference on Parallel Processing (Euro-Par 2005), vol. 3646 of LNCS, Springer, 2005, pp. 241–251.

[18] O. Bonorden, M. Dynia, J. Gehweiler, and R. Wanka, *PUB-library, release 8.1-pre, user guide and function reference*, 2003.

[19] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, *The paderborn university BSP (PUB) library*, Parallel Computing, 29 (2003), pp. 187–207.

[20] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. Boman, and Ümit V. Çatalyürek, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.

[21] D. Bozdağ, Ümit V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. Boman, and F. Ozguner, *A parallel distance-2 graph coloring algorithm for distributed memory computers*, in proceedings of the 2005 International Conference on High Performance Computing and Communications, vol. 3726 of LNCS, Springer, 2005, pp. 796–806.

[22] D. Bozdağ, Ümit V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, *Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 2418–2446.

[23] M. Brinkhuis, G. Meijer, P. van Diest, L. Schuurmans, and J. Baak, *Minimum spanning tree analysis in advanced ovarian carcinoma*, Analytical and Quantitative Cytology and Histology, 19 (1997), pp. 194–201.

[24] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, *Register allocation via coloring*, Computer Languages, 6 (1981), pp. 47–57.

[25] F. Chang, C.-J. Chen, and C.-J. Lu, *A linear-time component-labeling algorithm using contour tracing technique*, Computer Vision and Image Understanding, 93 (2004), pp. 206–220.

[26] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, Cambridge, 2007.

[27] C. Chen and S. Morris, *Visualizing evolving networks: minimum spanning trees versus pathfinder networks*, IEEE Symposium on Information Visualization, 8 (2003).

[28] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi, *Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms*, ACM Journal on Experimental Algorithmics, 3 (1998).

[29] T. F. Coleman and J. J. More, *Estimation of sparse jacobian matrices and graph coloring problems*, SIAM Journal on Numerical Analysis, 1 (1983), pp. 187–209.

[30] N. Copty, S. Ranka, G. Fox, and R. V. Shankar, *A data parallel algorithm for solving the region growing problem on the connection machine*, Journal of Parallel and Distributed Computing, 21 (1994), pp. 160–168.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, third ed., 2009.

[32] P. Crescenzi and V. Kann, *A compendium of NP optimization problems*. http://www.nada.kth.se/˜viggo/wwwcompendium/, 2005.

[33] Cscapes, *The institute for combinatorial scientific computing and petascale simulations*. http://www.cscapes.org/, 2006.

[34] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, IMA Journal of Applied Mathematics, 1 (1974), pp. 117–119.

[35] G. Cybenko, T. G. Allen, and J. E. Polito, *Practical parallel algorithms for transitive closure and clustering*, International Journal of Parallel Computing, 17 (1988), pp. 403–423.

[36] B. Dawes and D. Abrahams, *The Boost C++ libraries*. http://www.boost.org, 2009.

[37] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and Ümit V. Çatalyürek, *Parallel hypergraph partitioning for scientific comuting*, in proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), 2006.

[38] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[39] M. B. Dillencourt, H. Samet, and M. Tamminen, *A general approach to connected-component labeling for arbitrary image representations*, Journal of the ACM, 39 (1992), pp. 253–280.

[40] J. Dore, J. Gilbert, E. Bignon, A. C. de Paulet, T. Ojasoo, M. Pons, J. Raynaud, and J. Miquel, *Multivariate analysis by the minimum spanning tree method of the structural determinants of diphenylethylenes and triphenylacrylonitriles implicated in estrogen receptor binding, protein kinase C activity, and MCF7 cell proliferation*, Journal of Medicinal Chemistry, 35 (1992), pp. 573–583.

[41] I. Duff, K. Kaya, J. Langguth, F. Manne, and B. Uçar, *Experiments on push-relabel based maximum cardinality matching algorithms for bipartite graphs*, 2011. In preparation.

[42] I. S. Duff, K. Kaya, and B. Uçar, *Design, implementation, and analysis of maximum transversal algorithms*, Tech. Rep. TR/PA/10/76, CERFACS, Toulouse, France, 2010.

[43] I. S. Duff and J. Koster, *The design and use of algorithms for permuting large entrees to the diagonal of sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 20 (1999), pp. 889–901.

[44] ——, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM Journal on Matrix Analysis and Applications, 22 (2001), pp. 973–996.

[45] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis*, Cambridge University Press, 1998.

[46] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM, 19 (1972), pp. 248–264.

[47] C. Fiorio and J. Gustedt, *Two linear time union-find strategies for image processing*, Theoretical Computer Science, 154 (1996), pp. 165–181.

[48] A. Gamst, *Some lower bounds for a class of frequency assignment problems*, IEEE Transactions of Vehicular Technology, 35 (1986), pp. 8–14.

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, NY, USA, 1979.

[50] M. R. Garey, D. S. Johnson, and H. C. So, *An application of graph coloring to printed circuit testing*, IEEE Transactions on Circuits and Systems, 23 (1976), pp. 591–599.

[51] A. H. Gebremedhin, *Parallel graph coloring*, Master's thesis, Department of Informatics, University of Bergen, Norway, 1999.

[52] A. H. Gebremedhin and F. Manne, *Scalable parallel graph coloring algorithms*, Concurrency: Practice and Experience, 12 (2000), pp. 1131–1146.

[53] A. H. Gebremedhin, F. Manne, and A. Pothen, *Graph coloring in optimization revisited*, Tech. Rep. 226, Department of Informatics, University of Bergen, Norway, 2002.

[54] ——, *Parallel distance-k coloring algorithms for numerical optimization*, in proceedings of the 8th International European Conference on Parallel Processing (Euro-Par 2002), vol. 2400 of LNCS, Springer Berlin/Heidelberg, 2002, pp. 912–921.

[55] ——, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Review, 47 (2005), pp. 629–705.

[56] A. H. Gebremedhin, F. Manne, and T. Woods, *Speeding up parallel graph coloring*, in Applied Parallel Computing, J. Dongarra, K. Madsen, and J. Wasniewski, eds., vol. 3732 of LNCS, Springer Berlin/Heidelberg, 2006, pp. 1079–1088.

[57] A. H. Gebremedhin, D. Nguyen, A. Pothen, and M. M. A. Patwary, *ColPack: Graph coloring software for derivative computation and beyond*, Submitted to ACM Transactions on Mathematical Software, (October, 2010).

[58] A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen, *New acyclic and star coloring algorithms with applications to computing Hessians*, SIAM Journal on Scientific Computing, 29 (2007), pp. 1042–1072.

[59] A. George, M. Heath, J. W. H. Liu, and E. G. Y. Ng, *Sparse Cholesky factorization on a local-memory multiprocessor*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 327–340.

[60] A. George and J. W. H. Liu, *An optimal agorithm for symbolic factorization of symmetric matrices*, SIAM Journal on Computing, 9 (1980), pp. 583–593.

[61] J. R. Gilbert, E. G. Ng, and G. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, in Graph Theory and Sparse Matrix Computation, Springer-Verlag, 1992, pp. 107–139.

[62] J. R. Gilbert, E. G. Ng, and B. W. Peyton, *An efficient algorithm to compute row and column counts for sparse Cholesky factorization*, Journal on Matrix Analysis and Applications, 15 (1994), pp. 1075–1091.

[63] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem*, in proceedings of the 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 136–146.

[64] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Addison-Wesley, Essex, England, second ed., 2003.

[65] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, MIT Press, Cambridge, MA, USA, 1994.

[66] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Computing, 26 (2000), pp. 1519–1534.

[67] B. Hendrickson and R. Leland, *The Chaco user's guide, version 2.0*, 1995.

[68] B. Hendrickson and A. Pothen, *Combinatorial scientific computing: The enabling power of discrete algorithms in computational science*, in High Performance Computing for Computational Science (VECPAR 2006), M. Daydé, J. Palma, Á. Coutinho, E. Pacitti, and J. Lopes, eds., vol. 4395 of LNCS, Springer Berlin/Heidelberg, 2007, pp. 260–280.

[69] J. M. D. Hill, S. R. Donaldsony, and A. McEwan, *Installation and user guide for the Oxford BSP toolset(v1.4) implementation of BSPlib*, 1998.

[70] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[71] R. Hynes, *A new class of set union algorithms*, Master's thesis, Department of Computer Science, University of Toronto, Canada, 1998.

[72] M. T. Jones and P. E. Plassmann, *A parallel graph coloring heuristic*, SIAM Journal on Scientific Computing, 14 (1993), pp. 654–669.

[73] R. M. Karp and M. Sipser, *Maximum matching in sparse random graphs*, Annual IEEE Symposium on Foundations of Computer Science, 0 (1981), pp. 364–375.

[74] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *Multilevel hypergraph partitioning: application in VLSI domain*, in proceedings of the 34th Conference on Design Automation, 1997, pp. 526–529.

[75] G. Karypis and V. Kumar, *Metis a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Version 4.0*, 1998.

[76] G. Karypis, K. Schloegel, and V. Kumar, *Parmetis: Parallel graph partitioning and sparse matrix ordering library, Version 3.1*, 2003.

[77] K. Kayser, S. Jacinto, G. Bohm, P. Frits, W. Kunze, A. Nehrlich, and H. Gabius, *Application of computer-assisted morphometry to the analysis of prenatal development of human lung*, Anatomia, Histologia, Embryologia, 26 (1997), pp. 135–139.

[78] K. Kayser, H. Stute, and M. Tacke, *Minimum spanning tree, integrated optical density and lymph node metastasis in bronchial carcinoma*, Analytical Cellular Pathology, 5 (1993), pp. 225–234.

[79] P. Kegel, M. Schellmann, and S. Gorlatch, *Using OpenMP vs. Threading Building Blocks for medical imaging on multi-cores*, in proceedings of the 15th International European Conference on Parallel Processing (Euro-Par 2009), vol. 5704 of LNCS, Springer Verlag, 2009, pp. 654–665.

[80] J. Keller, C. W. Kessler, and J. L. Träff, *Practical PRAM Programming*, Wiley, New York, 2001.

[81] J. M. Kleinberg, *Authoritative sources in a hyperlinked environment*, Journal of the ACM, 46 (1999), pp. 604–632.

[82] L. Kucera, *The greedy coloring is a bad probabilistic algorithm*, Journal of Algorithms, 12 (1991), pp. 674–684.

[83] J. Langguth, F. Manne, and P. Sanders, *Heuristic initialization for bipartite matching problems*, ACM Journal of Experimental Algorithmics, 15 (2010), pp. 1.3:1–1.3:22.

[84] J. Langguth, M. M. A. Patwary, and F. Manne, *Parallel algorithms for bipartite matching problems on distributed memory computers*, Submitted to Parallel Computing, (October, 2010).

[85] G. Lewandowski, *Practical implementations and applications of graph coloring*, PhD thesis, University of Wisconsin-Madison, 1994.

[86] J. W. H. Liu, *A compact row storage scheme for cholesky factors using elimination trees*, ACM Transactions on Mathematical Software, 12 (1986), pp. 127–148.

[87] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[88] F. Manne, P. Heggernes, and P. Bjørstad, *Parallel algorithms for combinatorial scientific computing (Parcomb).* Parcomb project description is available at http://www2.ii.uib.no/parcomb/pub/pdf/main.pdf, 2007.

[89] F. Manne and M. M. A. Patwary, *A scalable parallel union-find algorithm for distributed memory computers*, in proceedings of the Eighth International Conference on Parallel Processing and Applied Mathmatics (PPAM 2009), vol. 6067 of LNCS, Springer Verlag, 2009, pp. 186–195.

[90] Y. Maon, B. Schieber, and U. Vishkin, *Parallel ear decomposition search (eds) and st-numbering in graphs*, Theoretical Computer Science, 47 (1986), pp. 277–298.

[91] M. Matos, B. Raby, J. Zahm, M. Polette, P. Birembaut, and N. Bonnet, *Cell migration and proliferation are not discriminatory factors in the in vitro sociologic behavior of bronchial epithelial cell lines*, Cell Motility and the Cytoskeleton, 53 (2002), pp. 53–65.

[92] D. Matula, *A max-min theorem for graphs with application to graph coloring*, SIAM Review, 10 (1968), pp. 481–482.

[93] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. Srivastava, *Coverage problems in wireless ad-hoc sensor networks*, in proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM 2001), vol. 3, 2001, pp. 1380–1387.

[94] K. Melhorn and S. Näher, *LEDA, A Platform for Combinatorial Geometric Computing*, Cambridge University Press, 1999.

[95] G. L. Miller and V. Ramachandran, *Efficient parallel ear decomposition with applications.* unpublished manuscript, MSRI, Berkeley, CA, January 1986.

[96] R. H. Möhring and M. Müller-Hannemann, *Cardinality matching: Heuristic search for augmenting paths*, Tech. Rep. 439, Fachbereich Mathematik, Technische Universität Berlin, 1995.

[97] J. Muerle and D. Allen, *Experimental evaluation of techniques for automatic segmentation of objects in a complex scene*, Pictorial Pattern Recognition, (1968), pp. 3–13.

[98] Nvidia, *GPU computing.* http://www.nvidia.com/.

[99] V. Olman, D. Xu, and Y. Xu, *Identification of regulatory binding sites using minimum spanning trees*, in proceedings of the Eighth Pacific Symposium on Biocomputing (PSB 2003), Hawaii, World Scientific, Singapore, 2003, pp. 327–338.

[100] OpenMP, *The OpenMP API specification for parallel programming.* http://openmp.org/.

[101] V. Osipov, P. Sanders, and J. Singler, *The filter-Kruskal minimum spanning tree algorithm*, in proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2009), 2009, pp. 52–61.

[102] L. Page, S. Brin, R. Motwani, and T. Winograd, *The pagerank citation ranking: Bringing order to the web*, Tech. Rep. 1999-66, Stanford InfoLab, November 1999.

[103] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Review, 3 (1961), pp. 119–130.

[104] M. M. A. Patwary, R. H. Bisseling, and F. Manne, *Parallel greedy graph matching using an edge partitioning approach*, in proceedings of the Fourth ACM SIGPLAN Workshop on High-level Parallel Programming and Applications (HLPP 2010), 2010, pp. 45–54.

[105] M. M. A. Patwary, J. Blair, and F. Manne, *An experimental evaluation of union-find algorithms for the disjoint-set data structure*, Submitted to the ACM Journal of Experimental Algorithmics, (November, 2010).

[106] M. M. A. Patwary, J. R. S. Blair, and F. Manne, *Experiments on union-find algorithms for the disjoint-set data structure*, in proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010), vol. 6049 of LNCS, Springer Verlag, 2010, pp. 411–423.

[107] M. M. A. Patwary, A. H. Gebremedhin, and A. Pothen, *New multithreaded ordering and coloring algorithms for multicore architectures*, in the 17th International European Conference on Parallel Processing (Euro-Par 2011), 2011. Submitted.

[108] F. J. Peters, *Sparse matrices and substructures*, Tech. Rep. Mathematical Centre Tracts, 119, Mathematisch Centrum, Amsterdam, The Netherlands, 1980.

[109] A. Pothen and C.-J. Fan, *Computing the block triangular form of a sparse matrix*, ACM Transactions on Mathematical Software, 16 (1990), pp. 303–324.

[110] A. Pothen and S. Toledo, *Elimination structures in scientific computing*, CRC Press, Boca Raton, 2004.

[111] M. J. D. Powell and P. L. Toint, *On the estimation of sparse hessian matrices*, SIAM Journal on Numerical Analysis, 16 (1979), pp. 1060–1074.

[112] R. Preis, *Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs*, in proceedings of the Symposium on Theoretical Aspects of Computer Science in general graphs (STACS 99), vol. 1563 of LNCS, Springer, 1999, pp. 259–269.

[113] M. Randic and J. Zupan, *On interpretation of well-known topological indices*, Journal of Chemical Information and Computer Sciences, 41 (2001), pp. 550–560.

[114] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multicore Processor Parallelism*, O'Reilly Media, 2007.

[115] D. Rose, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, Graph Theory and Computing, (1972), pp. 183–217.

[116] Y. Saad, *Multilevel ILU with reorderings for diagonal dominance*, SIAM Journal on Scientific Computing, 27 (2005), pp. 1032–1057.

[117] J. C. Setubal, *New experimental results for bipartite matching*, in proceedings of the Network Optimization, Theory and Practice (NETFLOW 1993), 1992.

[118] J. Shalf, *The new landscape of parallel computer architecture*, Journal of Physics: Conference Series, 78 (2007), p. 012066.

[119] J. R. Shewchuk, *Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator*, in Applied Computational Geometry: Towards Geometric Engineering, M. C. Lin and D. Manocha, eds., vol. 1148 of LNCS, Springer-Verlag, 1996, pp. 203–222.

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

[121] J. J. Sylvester, *On an application of the new atomic theory to the graphical representation of the invariants and covariants of binary quantics, with three appendices*, American Journal of Mathematics, 1 (1878), pp. 64–104.

[122] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM, 22 (1975), pp. 215–225.

[123] ——, *A class of algorithms which require nonlinear time to maintain disjoint sets*, Journal of Computer and System Sciences, 18 (1979), pp. 110–127.

[124] R. E. Tarjan and U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM Journal on Computing, 14 (1985), pp. 862–874.

[125] T. J. Tautges, T. Blacker, and S. A. Mitchell, *The whisker weaving algorithm: A connectivity-based method for constructing all.hexahedral finite element meshes*, International Journal for Numerical Methods in Engineering, 39 (1996), pp. 3327–3349.

[126] M. Thorpe, *Continuous deformations in random networks*, Journal of Non-Crystalline Solids, 57 (1983), pp. 355–370.

[127] Top500, *Top500 supercomputer sites.* http://www.top500.org/.

[128] Y.-C. Tseng, T. T.-Y. Juang, and M.-C. Du, *Building a multicasting tree in a high-speed network*, IEEE Concurrency, 6 (1998), pp. 57–67.

[129] Ümit V. Çatalyürek and C. Aykanat, *PaToH: A multilevel hypergraph partitioning tool, version 3.0.* PaToH is available at http://bmi.osu.edu/~umit/software.htm, 1999.

[130] ——, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), 2001, pp. 609–625.

[131] Ümit V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, *Multithreaded algorithms for graph coloring.* Submitted for journal publication, 2011.

[132] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[133] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[134] S. Vaughan-Nichols, *New trends revive supercomputing industry*, Computer, 37 (2004), pp. 10–13.

[135] U. Vishkin, *On efficient parallel strong orientation*, Information Processing Letters, 20 (1985), pp. 235–240.

[136] J. Wassenberg, D. Bulatov, W. Middelmann, and P. Sanders, *Determination of maximally stable extremal regions in large images*, in proceeding of the Signal Processing, Pattern Recognition, and Applications (SPPRA 2008), Acta Press, 2008.

[137] D. J. A. Welsh and M. B. Powell, *An upper bound for the chromatic number of a graph and its application to timetabling problems*, The Computer Journal, 10 (1967), pp. 85–86.

[138] T. Westman, D. Harwood, T. Laitnen, and M. Pietikanen, *Color segmentation by hierarchical connected components analysis with image enhancement by symmetric neighborhood filters*, in proceedings of the 10th International Conference on Pattern Recognition, vol. 1, 1990, pp. 796–802.

[139] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers*, Prentice-Hall Inc, Upper Saddle River, NJ, second ed., 1999.

[140] S. Wolfram, *A New Kind of Science*, Wolfram Media, 2002.

[141] K. Wu, E. Otoo, and K. Suzuki, *Optimizing two-pass connected-component labeling algorithms*, Pattern Analysis and Applications, 12 (2009), pp. 117–135.

[142] M. Zaker, *Results on the grundy chromatic number of graphs*, Discrete Mathematics, 306 (2006), pp. 3166–3173.

[143] S. Q. Zheng, J. S. Lim, and S. S. Iyengar, *Routing using implicit connection graphs*, in proceedings of the 9th International Conference on VLSI Design: VLSI in Mobile Communication (VLSID 1996), 1996, p. 49.

[144] Y. Zhu and D. Mutchler, *On constructing the elimination tree*, Discrete Applied Mathematics, 48 (1994), pp. 93–98.

[145] S. W. Zucker, *Region growing: Childhood and adolescence*, Computer Graphics and Image Processing, 5 (1976), pp. 382–399.

I

An Experimental Evaluation of UNION-FIND Algorithms for the Disjoint-Set Data Structure

Md. Mostofa Ali Patwary* Jean R. S. Blair**

Fredrik Manne*

Abstract

The disjoint-set data structure is used to maintain a collection of non-overlapping sets of elements from a finite universe. Algorithms that operate on this data structure are often referred to as UNION-FIND algorithms. They are used in numerous practical applications and are also available in several software libraries. This paper presents an extensive experimental study comparing the time required to execute 63 variations of UNION-FIND algorithms. The study includes all the classical algorithms, several recently suggested enhancements, and also different combinations and optimizations of these. Our results clearly show that a somewhat forgotten simple algorithm that combines a novel UNION technique developed by Rem in 1976 with a one-pass compression technique is the fastest, in spite of the fact that its worst-case time complexity is inferior to that of the commonly accepted "best" algorithms.

Keywords: UNION-FIND, Disjoint Set, Experimental Algorithms.

1 Introduction

Let U be a set of n distinct elements and let S_i denote a subset of U. Two sets S_1 and S_2 are disjoint if $S_1 \cap S_2 = \emptyset$. A disjoint-set data structure maintains a dynamic collection $\{S_1, S_2, \ldots, S_k\}$ of disjoint sets that together cover the universe U. Each set is identified by a representative x, which is usually some member

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, {Mostofa.Patwary,fredrikm}@ii.uib.no

**Office of the Dean, United States Military Academy, West Point, NY 10996, USA, Jean.Blair@usma.edu.

of the set. The two main operations are to FIND which set a given element belongs to by locating its representative element and to replace two existing sets with their UNION. In addition, there is a MAKESET operation which adds a new element to U as a singleton set.

The underlying data structure of each set is typically a rooted tree represented by a parent function $p(x) \in S_i$ for each $x \in U$; the element in the root of a tree satisfies $p(x) = x$ and is the representative of the set [9]. Then MAKESET(x) is achieved by setting $p(x) \leftarrow x$ and the output of FIND(x) is the root of the tree containing x. This is found by following x's *find-path*, which is the path of parent pointers from x up to the root of x's tree. A set of algorithms that operate on this data structure is often referred to as a UNION-FIND algorithm.

This disjoint-set data structure is frequently used in practice, including in application areas such as image decomposition, clustering, sparse matrix computations, and graph algorithms. It is also a standard subject taught in most algorithms courses.

Early theoretical work established algorithms with worst-case time complexity $\Theta(n + m \cdot \alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [3, 4, 17, 18, 19, 20, 21], where α is the very slowly growing inverse of Ackermann's function [1]. These theoretically best classical algorithms include a standard UNION method using either LINK-BY-RANK or LINK-BY-SIZE and a FIND operation incorporating one of three standard compression techniques: PATH-COMPRESSION, PATH-SPLITTING, or PATH-HALVING. Other early algorithms either use a different compression technique like COLLAPSING, use a more naive union technique or interleave the two FIND operations embedded in a UNION operation, as was the case with Rem's algorithm. The worst case time complexity of these variations are not optimal [21].

The current work presents an extensive experimental study comparing the time required to execute a sequence of UNION operations, each with two embedded FIND operations. Altogether, we consider 63 variations; 36 of these had been well studied in the theoretical literature by 1984. We call these the classical algorithms. The remaining 27 variations implement a number of improvements. Our results clearly show that a slight variation on a somewhat forgotten simple algorithm developed by Rem in 1976 [7] is the fastest, in spite of the fact that its worst-case complexity is inferior to that of the commonly accepted "best" algorithms.

Related experimental studies have compared only a few UNION-FIND algorithms, usually in the context of a specific software package. In particular, [12] and [10] compared only two and six UNION-FIND algorithms, respectively, in the context of sparse matrix factorization. The works in [23] and [24] compared eight and three UNION-FIND algorithms, respectively, in the setting of image processing. More recently, [15] compared a classic algorithm with a variation described here in the IPC subsection of Section 2.2. The most extensive previous experimental study was Hynes' masters thesis [11] where he compared the performance of 18 Union-Find algorithms used to find the connected components of a set of Erdös-Rényi style random graphs.

The remainder of this paper is organized as follows. In the next section we give the specific setting for our sequences of Union and Find operations and briefly review the classical algorithms that were studied from a theoretical perspective in [21], along with several well-known and a few not-so-well known efficiency-driven implementation techniques. Our experimental methodology and test set details are described in Section 3, followed by discussion and results in Section 4. Concluding remarks are given in Section 5.

2 Union-Find Algorithms

This section outlines the primary Union-Find algorithms. We also include a number of suggested enhancements designed to speed up implementations of these algorithms.

Our presentation is from the viewpoint of its use in finding connected components of a simple undirected graph $G(V, E)$ as shown in Algorithm 1. In this case, the Union-Find algorithm computes a minimal subset $S \subseteq E$ such that S is a spanning forest of G. Thus a set corresponds to vertices in the same connected component and the representative element is one of the vertices in the set. Note that if G is weighted and edges are processed in order of increasing weight then Algorithm 1 is Kruskal's algorithm [4] for computing minimum weight spanning forests.

Algorithm 1. Use of Union-Find

```
$S \leftarrow \emptyset$
for each $x \in V$ do
    Makeset($x$)
for each $(x, y) \in E$ do
    if Find($x$) $\neq$ Find($y$) then
        Union($x, y$)
        $S \leftarrow S \cup \{(x, y)\}$
```

2.1 Classical Algorithms

Here we discuss the classical Union techniques and then present techniques for compressing trees during a Find operation. Finally, we describe classical algorithms that interleave the Find operations embedded in a Union along with a compression technique that can only be used with this type of algorithm.

2.1.1 Union Techniques

The Union(x, y) operation merges the sets containing x and y, typically by finding the roots of their respective trees and then linking them together by setting the parent pointer of one root to point to the other.

Clearly, storing the results of the two Find operations on line 5 and then using these as input to the Union operation in line 6 will speed up Algorithm 1. This replacement for lines 5–7 in Algorithm 1 is known as Quick-Union (Quick) in [8]. Throughout the remainder of this paper we use Quick.

Four classic variations of the Union algorithm center around the method used to Link the two roots. Let r_x and r_y be the roots of the two trees that are to be merged. Then Union with Naive-Link (nl) arbitrarily chooses one of r_x and r_y and sets it to point to the other. This can result in a tree of height $O(n)$.

A similar approach is Union with Link-by-Index (li). Here it is assumed that r_x and r_y are both originally stored in a table and thus each has a unique index typically in the range 1 through n. When performing the Union operation whichever of r_x and r_y that has the lowest index is set to point to the other. This method can also result in a tree of height $O(n)$. We note that one could replace the use of the index of an element in li with any unique identifier (*id*) that can be ordered. In our presentation of algorithms using li we do not distinguish between an element and its original index.

In Union with Link-by-Size (ls) we set the root of the tree containing the fewest nodes to point to the root of the other tree, arbitrarily breaking ties. To implement ls efficiently we maintain the size of the tree in the root.

For the Union with Link-by-Rank (lr) operation we associate a rank value, initially set to 0, with each node. If two sets are to be merged and the roots have equal rank, then the rank of the root of the combined tree is increased by one. In all other lr operations the root with the lowest rank is set to point to the root with higher rank and all ranks remain unchanged. Note that when using lr the parent of a node x will always have higher rank than x. This is known as the *increasing rank property*. The union algorithm presented in most textbooks uses the Quick and lr enhancements to implement lines 5–7 of Algorithm 1. Details are given in Algorithm 2.

Algorithm 2. Quick-Union with lr(x, y)

```
$r_x \leftarrow x, r_y \leftarrow y$
while $r_x \neq p(r_x)$ do
    $r_x \leftarrow p(r_x)$
while $r_y \neq p(r_y)$ do
    $r_y \leftarrow p(r_y)$
if $r_x \neq r_y$ then
    if $rank(r_x) \leq rank(r_y)$ then
        $p(r_x) \leftarrow r_y$
        if $rank(r_x) = rank(r_y)$ then
            $rank(r_y) \leftarrow rank(r_y) + 1$
    else
        $p(r_y) \leftarrow r_x$
    $S \leftarrow S \cup \{(x, y)\}$
```

Both ls and lr ensure that the find-path of an n vertex graph will never be longer than $\log n$. The alleged advantage of lr over ls is that a rank value requires less storage than a size value, since the rank of a root in a set containing n vertices will never be larger than $\log n$ [4]. Also, sizes must be updated with every Union operation whereas ranks need only be updated when the two roots have equal rank. On the other hand LR requires an additional comparison before each LINK operation.

2.1.2 Compression Techniques

Altogether we describe six classical compression techniques used to compact the tree, thereby speeding up subsequent FIND operations. The term NF will represent a FIND operation with no compression.

Using PATH-COMPRESSION (PC) the find-path is traversed a second time after the root is found, setting all parent pointers on the find-path to point to the root. Two alternatives to PC are PATH-SPLITTING (PS) and PATH-HALVING (PH). With PS the parent pointer of every node on the find-path is set to point to its grandparent. This has the effect of partitioning the find-path nodes into two disjoint paths, both hanging off the root. In PH this process of pointing to a grandparent is only applied to every other node on the find-path. The advantage of PS and PH over PC is that they can be performed without traversing the find-path a second time. On the other hand, PC compresses the tree more than either of the other two.

Note that when using ranks PC, PS, and PH all maintain the increasing rank property. Furthermore, any one of the three combined with either LR or LS has the same asymptotic time bound of $O(m \cdot \alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [4, 18, 20, 21].

Another set of compression techniques is REVERSAL-OF-TYPE-k. With this the first node x on the find-path and the last k nodes are set to point to the root while the remaining nodes are set to point to x. In a REVERSAL-OF-TYPE-0 (R0) every node on the find-path from x, including the root, is set to point to x and x becomes the new root, thus changing the representative element of the set. Both R0 and REVERSAL-OF-TYPE-1 (R1) can be implemented efficiently, but for any values of $k > 1$ implementation is more elaborate and might require a second pass over the find-path [21]. We limit $k \leq 1$. Using either R0 or R1 with any of NL, LI, LR, or LS gives an asymptotic running time of $O(n + m \log n)$ [21].

In COLLAPSING (CO) every node of a tree will point directly to the root so that all find-paths are no more than two nodes long. When merging two trees in a UNION operation, nodes of one of the trees are changed to point to the root of the other tree. To implement this efficiently the nodes are stored in a linked list using a sibling pointer in addition to the parent pointer. The asymptotic running time of CO with either LS or LR is $O(m + n \log n)$; CO with NL or LI is $O(m + n^2)$ [21].

It is possible to combine any of the four different UNION methods with any of the seven compression techniques (including NF), thus giving rise to a total of 28 different algorithm combinations. We denote each of these algorithms by combining the abbreviation of its UNION method with the abbreviation of its compression technique (e.g., lrpc). The asymptotic running times of these classical algorithms are summarized in the tables on page 280 of [21].

2.1.3 Classical Interleaved Algorithms

Interleaved (Int) algorithms differ from the Union-Find algorithms mentioned so far in that the two Find operations in line 5 of Algorithm 1 are performed as one interleaved operation. The main idea is to move two pointers r_x and r_y alternatively along their respective find-paths so that if x and y are in the same component then $p(r_x) = p(r_y)$ when they reach their lowest common ancestor and processing can stop. Also, if x and y are in different components, then in certain cases the two components can be linked together as soon as one of the pointers reaches a root. Thus, one root can be linked into a non-root node of the other tree. The main advantage of the Int algorithms is that they can avoid traversing portions of find-paths.

The first Int algorithm is Rem's algorithm (Rem) [7] which, like li, builds trees based on increasing index values. Thus the constructed trees always have the property that a node either points to a higher numbered node or to itself (if it is a root).

Instead of performing Find(x) and Find(y) separately, these are executed simultaneously by first setting $r_x \leftarrow x$ and $r_y \leftarrow y$. Then whichever of r_x and r_y has the smaller parent value is moved one step upward in its tree. In this way it follows that if x and y are in the same component then at some stage of the algorithm we will have $p(r_x) = p(r_y) =$ the lowest common proper ancestor of x and y. The algorithm tests for this condition in each iteration and can, in this case immediately stop.

As originally presented, Rem integrates the Union operation with a compression technique known as Splicing (sp) which works as follows: In the case when r_x is to be moved to $p(r_x)$, let $z = p(r_x)$, then the value of $p(r_x)$ is set to $p(r_y)$ before r_x is set to point to z. Thus, following the operation the subtree originally pointed to by r_x is now a sibling of r_y. This neither compromises the increasing parent property (because $p(r_x) < p(r_y)$) nor invalidates the set structures (because the two sets will have been merged when the operation ends.) This also takes care of a possible final Union operation once r_x (or r_y) reaches the root of its subtree. The effect of sp is that each new parent has a higher value than the

Algorithm 3. RemSP(x, y)

```
$r_x \leftarrow x$, $r_y \leftarrow y$
while $p(r_x) \neq p(r_y)$ do
    if $p(r_x) < p(r_y)$ then
        if $r_x = p(r_x)$ then
            $p(r_x) \leftarrow p(r_y)$, break
        $z \leftarrow p(r_x)$, $p(r_x) \leftarrow p(r_y)$, $r_x \leftarrow z$
    else
        if $r_y = p(r_y)$ then
            $p(r_y) \leftarrow p(r_x)$, break
        $z \leftarrow p(r_y)$, $p(r_y) \leftarrow p(r_x)$, $r_y \leftarrow z$
```

value of the old parent, thus compressing the tree. The full algorithm is given as Algorithm 3. The running time of Rem with SP (RemPS) is $O(m \log_{(2+m/n)} n)$ [21].

Tarjan and van Leeuwen present a variant of Rem that uses ranks rather than identifiers. This algorithm is slightly more complicated than Rem, as it also checks if two roots of equal rank are being merged and if so updates the rank values appropriately. Details are given on page 279 of [21]. We label this algorithm as TvL. The running time of TvL with SP (TvLSP) is $O(m \cdot \alpha(m, n))$.

Note that SP can easily be replaced in either Rem or TvL with NF, PC, or PS. However, it does not make sense to use PH with either because PH might move one of r_x and r_y past the other without discovering that they are in fact in the same tree. Also, since R0 and R1 would move a lower numbered (or ranked) node above higher numbered (ranked) nodes, thus breaking the increasing (rank or *id*) property, we will not combine an Int algorithm with either R0 or R1.

2.2 Implementation Enhancements

We now consider three different ways that the classical algorithms can be made to run faster by: *i)* making the algorithm terminate faster, *ii)* rewriting Int algorithms so that the most likely case is checked first, and *iii)* reducing memory requirements.

2.2.1 Immediate Parent Check (IPC)

This is a recent enhancement that checks before beginning Quick if x and y already have the same parent. If they do, Quick is not executed, otherwise execution continues normally. This idea is motivated by the fact that trees often have height one, and hence it is likely that two nodes in the same tree will have the same parent. The method was introduced by Osipov et al. [15] and used together with LR and PC in an algorithm to compute minimum weight spanning trees. IPC can be combined with any classical algorithm except Rem, which already implements the IPC test.

2.2.2 Better Interleaved Algorithms (Int)

The TvL algorithm, as presented in [21], can be combined with the IPC enhancement. However, the TvL algorithm will already, in each iteration of the main loop, check for $p(r_x) = p(r_y)$ and break the current loop iteration if this is the case. Still, three comparisons are needed before this condition is discovered. We therefore move this test to the top of the main loop so that the loop is only executed while $p(r_x) \neq p(r_y)$. (This is similar to the IPC test.)

In addition, it is possible to handle the case when $rank(p(r_x)) = rank(p(r_y))$ together with the case when $rank(p(r_x)) < rank(p(r_y))$. This will, in most cases, either reduce or at least not increase the number of comparisons; the only exception is when $rank(p(r_x)) < rank(p(r_y))$, which requires either one or two more comparisons. The new algorithm which we call eTvL is given as Algorithm 4 where SP is again a part of the algorithm.

Algorithm 4. eTvLSP(x, y)

```
$r_x \leftarrow x$, $r_y \leftarrow y$
while $p(r_x) \neq p(r_y)$ do
    if $rank(p(r_x)) \leq rank(p(r_y))$ then
        if $r_x = p(r_x)$ then
            if $rank(p(r_x)) = rank(p(r_y))$ then
                $r_y \leftarrow p(r_y)$
                if $r_y = p(r_y)$ then
                    $rank(r_y) \leftarrow rank(r_y) + 1$
            $p(r_x) \leftarrow p(r_y)$
            break
        $z \leftarrow r_x$, $p(r_x) \leftarrow p(r_y)$, $r_x \leftarrow p(z)$
    else
        if $r_y = p(r_y)$ then
            break
        $z \leftarrow r_y$, $p(r_y) \leftarrow p(r_x)$, $r_y \leftarrow p(z)$
```

A different variation of the TvL algorithm, called the Zigzag (ZZ) algorithm, was used in [13] for designing a parallel Union-Find algorithm where each tree could span across several processors on a distributed memory parallel computer. The main difference between the ZZ algorithm and eTvL is that the ZZ algorithm compares the ranks of r_x and r_y rather than the ranks of $p(r_x)$ and $p(r_y)$. Due to this it does not make sense to combine the ZZ algorithm with SP.

2.2.3 Memory Smart Algorithms (MS)

We now look at ways to reduce the amount of memory used by each algorithm. In the algorithms described so far each node has a parent pointer and, for some algorithms, either a size or rank value. In addition, for the CO algorithm each node has a sibling pointer. It follows that we will have between one and three fields in the record for each node. (Recall that we use the corresponding node's index into the array of records as its "name").

It is well known, although as far as we know undocumented, that when the parent pointer values are integers, one can eliminate one of the fields for most Union-Find implementations. The idea capitalizes on the fact that usually only the root of a tree needs to have a rank or size value. Moreover, for the root the parent pointer is only used to signal that the current node is in fact a root. Thus it is possible to save one field by coding the size or rank of the root into its parent pointer, while still maintaining the "root property." This can be achieved by setting the parent pointer of any root equal to its negated rank (or size) value.

This Memory-Smart (MS) enhancement of combining the rank/size field with the parent pointer can be incorporated into any of the classical algorithms except those using an Int algorithm (because they require maintaining the rank at every node, not just the root) or PH (because PH changes parent pointers to the grandparent value, which, if negative, will mess up the structure of the tree.) MS can also be combined with the IPC enhancement. Because LI and Rem do not use either size or rank, we will also classify these as MS algorithms.

3 Experimental Methodology

For the experiments we used a Dell PC with an Intel Core 2 Duo 2.4 GHz processor with 2 GB of memory, 4MB of shared level 2 cache, and running Fedora 10. All algorithms were implemented in C++ and compiled with GCC using the -O3 flag.

Table 1: Structural properties of the input graphs

Graph	$\|V\|$	$\|E\|$	Comp	Max Deg	Avg Deg	# Edges Processed
rw1 (m_t1)	97,578	4,827,996	1	236	99	692,208
rw2 (crankseg_2)	63,838	7,042,510	1	3,422	221	803,719
rw3 (inline_1)	503,712	18,156,315	1	842	72	5,526,149
rw4 (ldoor)	952,203	22,785,136	1	76	48	7,442,413
rw5 (af_shell10)	1,508,065	25,582,130	1	34	34	9,160,083
rw6 (boneS10)	914,898	27,276,762	1	80	60	11,393,426
rw7 (bone010)	986,703	35,339,811	2	80	72	35,339,811
rw8 (audikw_1)	943,695	38,354,076	1	344	81	10,816,880
rw9 (spal_004)	321,696	45,429,789	1	6,140	282	28,262,657
sw1	50,000	6,897,769	17,233	6,241	276	6,897,769
sw2	75,000	12,039,043	9,467	8,624	321	12,039,043
sw3	100,000	16,539,557	34,465	10,470	331	16,539,557
sw4	175,000	26,985,391	43,931	14,216	308	26,985,391
sw5	200,000	34,014,275	68,930	16,462	340	34,014,275
er1	100,000	453,803	24	25	9	453,803
er2	100,000	1,650,872	1	61	33	603,141
er3	500,000	2,904,660	8	30	12	2,904,660
er4	1,000,000	5,645,880	31	31	11	5,645,880
er5	500,000	9,468,353	1	70	38	3,476,740
er6	1,000,000	20,287,048	1	76	41	7,347,376

We used three test sets. The first consists of nine real world graphs (rw) of varying sizes drawn from different application areas such as linear programming, medical science, structural engineering, civil engineering, and the automotive industry [5]. The second includes five random small world graphs (sw) and the third contains six Erdös-Rényi style random graphs (er). For each synthetic graph (sw or er), we generated five different simple random graphs with the same number of vertices and with edge probability varying slightly around a given probability using the GTGraph package [2]. Statistics reported here about these graphs are an average for the five different random graphs of that type and size. For structural properties of the test sets see Table 1. The number of edges reported in the table for the synthetic graphs is the average number of edges in the corresponding five different random graphs. The first six columns are the name, number of vertices, number of edges, number of components, maximum degree, and average degree. The last column shows the number of edges (i.e., attempted Union operations) the algorithms processed before stopping.

To compute the run-time of an algorithm for a given rw graph, we execute the algorithm five times using each of five different random orderings of the edges, taking the average time as the result. The same "random orderings" of edges are used for all algorithms. The same process is used for each of the sw and er graphs. Hence we compute the average run-time of each graph in rw by taking the average of 25 total runs and for sw and er by taking the average of 125 total runs (5 runs for each of 5 orderings for each of 5 random graphs). Algorithms stop if and when they find that the entire graph is a single connected component. The time for all runs of reasonable algorithms (not including the extremely slow algorithms NLNF and NLCO) ranged from 0.007 seconds to 33 seconds.

4 Results

This section describes the results of experiments from 63 different algorithms. We first compare the classical algorithms and then algorithms that use the various enhancements presented in Section 2.2. Finally, we compare and discuss the 11 overall fastest algorithms. For each set of algorithms we give a table in which each cell represents an algorithm that combines the row's union method with the column's compression technique. The combinations for crossed out cells are either not possible or non-sensical. The rows with gray background are repeated from an earlier table.

Throughout we will say that an algorithm X *dominates* another algorithm Y if X performs at least as well as Y (in terms of the calculated average run-time) on every input graph. For illustrative purposes we will pick five specific dominating algorithms numbered according to the order in which they are first applied. These will be marked in the tables with their number inside a colored circle, as in ❶. If algorithm X dominates algorithm Y, an abbreviation for X with its number as a subscript will appear in Y's cell of the table, as in LRPC$_1$. Algorithms that are not dominated by any other algorithm are marked as undominated. It turns out that every undominated algorithm remains undominated even when compared with the algorithms in subsequent subsections. In total eight algorithms are undominated.

Each subsection where we present run times will show one or more plots of the performance of dominated algorithms relative to the performance of the algorithm that dominates them. That is, the plots are based on run-time expressed as a percent of the run-time of the dominating algorithm. The dominating algorithm is also plotted, showing at the 100% line. Where appropriate, the last figure of a subsection shows the performance of the undominated algorithms in that subsection relative to a fictitious algorithm (call it Min) that has, for each input graph, a run-time that is equal to the best of any algorithm considered in the subsection. Each subsection concludes with a short discussion and recommendations.

Following the presentation of results from the different types of algorithms we compare the overall performance of all 63 algorithms. Here we use a different metric to measure performance and using this show that eleven algorithms clearly outperform the others. As will be seen, six of the eight undominated algorithms are among the 11 overall best.

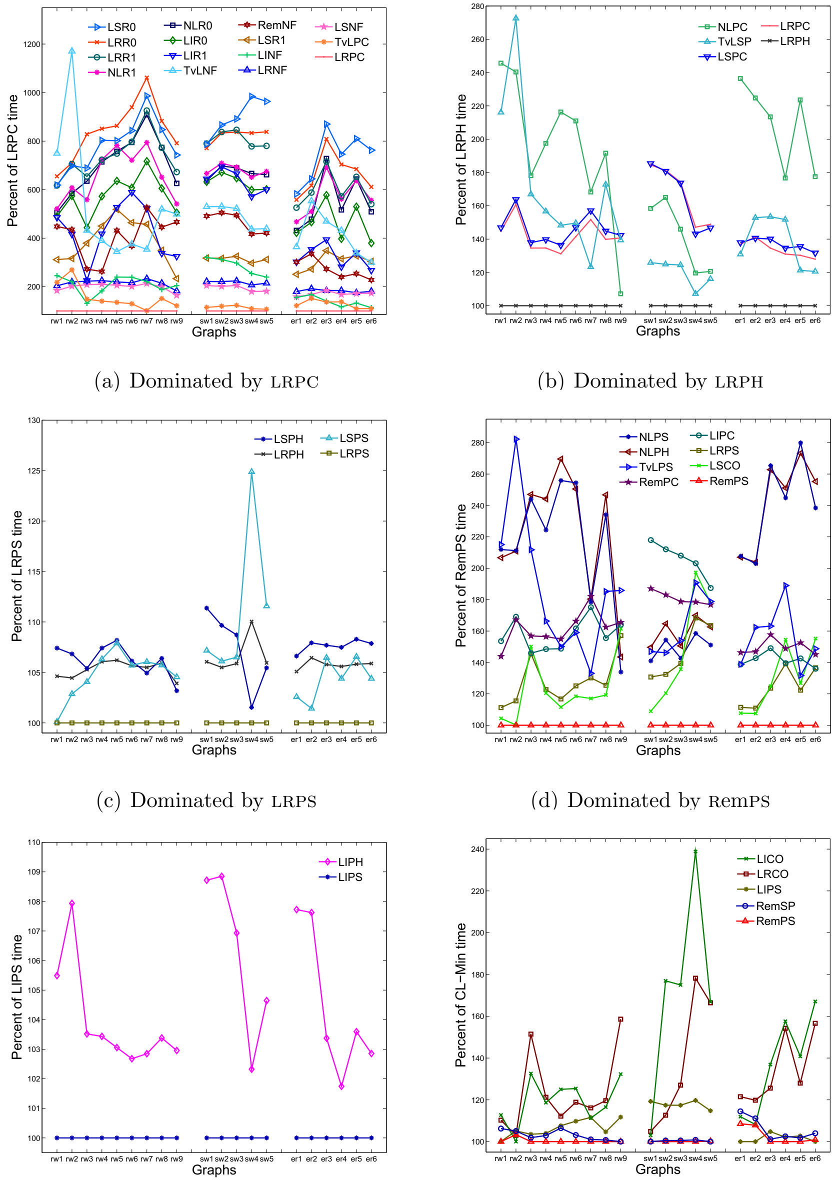
Finally, we present results on profiling these 11 best algorithms before making our final recommendations.

4.1 Classical Algorithms

Table 2: Relative performance of the classical Union-Find algorithms

	NF	PC	PH	PS	CO	R0	R1	SP
NL	LRPC$_1$	LRPH$_2$	RemPS$_4$	RemPS$_4$	LRPC$_1$	LRPC$_1$	LRPC$_1$	╳
LR	LRPC$_1$	❶ LRPH$_2$	❷ LRPS$_3$	❸ RemPS$_4$	undom.	LRPC$_1$	LRPC$_1$	╳
LS	LRPC$_1$	LRPH$_2$	LRPS$_3$	LRPS$_3$	RemPS$_4$	LRPC$_1$	LRPC$_1$	╳
LI	LRPC$_1$	RemPS$_4$	LIPS$_5$	❺ undom.	undom.	LRPC$_1$	LRPC$_1$	╳
Rem	LRPC$_1$	RemPS$_4$	╳	❹ undom.	╳	╳	╳	undom.
TvL	LRPC$_1$	LRPC$_1$	╳	RemPS$_4$	╳	╳	╳	LRPH$_2$

The two algorithms LRPC and LRPH have generally been accepted as the best, and so we begin by examining these. As indicated in Table 2, sixteen algorithms are dominated by LRPC and four more are dominated by LRPH. Furthermore, since LRPH dominates LRPC, LRPH also dominates all 16 algorithms dominated by LRPC. From this, we can infer that at least among this set of algorithms, the one-pass compression technique Path-Halving is better than the two-pass Path-Compression. Figures 1(a) and 1(b) show the performance of the dominated algorithms relative to LRPC and LRPH, respectively.



(a) Dominated by LRPC

(b) Dominated by LRPH

(c) Dominated by LRPS

(d) Dominated by RemPS

(e) Dominated by LIPS

(f) Undominated

Figure 1: Relative performance of classical Union-Find algorithms

Switching to the less well known one-pass compression technique PATH- SPLITTING, we see that three more algorithms are dominated by LRPS, including LRPH (Figure 1(c)). We can infer from this that in practice PATH-SPLITTING is a better one-pass compression technique than is PATH-HALVING.

Using this dominates criteria, neither the RANK nor SIZE UNION technique consistently outperforms the other, although RANK does admit one undominated algorithm, specifically LRCO.

Interestingly, two asymptotically inferior algorithms, RemPS and LIPS, each dominate LRPS and further dominate seven additional algorithms. These two algorithms, however, are incomparable using the dominates criteria since RemPS does not dominate LIPH while LIPS does and LIPS does not dominate LSCO while RemPS does. Because RemPS performs better overall than LIPS, we show all the algorithms it dominates in Figure 1(d) and only the one additional algorithm that LIPS dominates in Figure 1(e).

The remaining five algorithms are undominated. Their performance relative to CLASSICAL-MIN is shown in Figure 1(f). As will be seen in Subsection 4.5, only three of the undominated classical algorithms (LIPS, RemPS, and RemSP) are among the overall best 11 algorithms. In addition, LIPH from this classical set algorithms is in the overall best.

In the remainder of the paper we do not report results for algorithms using NF, R0, or R1 as these compression techniques consistently require several orders of magnitude more time than the others.

4.2 IPC Algorithms

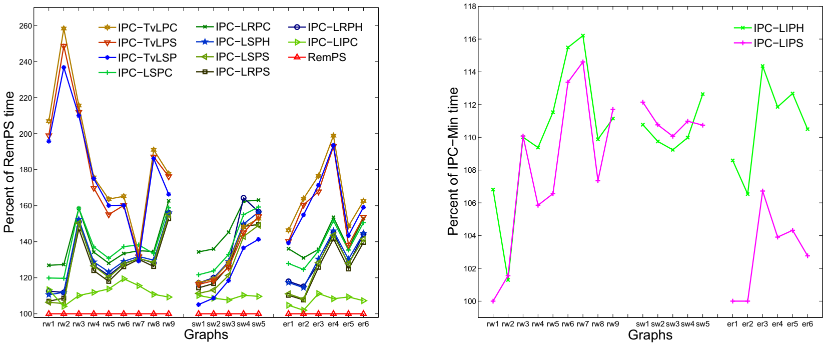
Table 3 shows the relative performance of algorithms enhanced with IPC. RemPS, which is carried over from the previous subsection, dominates all but four IPC algorithms. Figure 2(a) shows the relative performance of the 10 newly dominated algorithms relative to RemPS and Figure 2(b) shows the performance of the remaining two undominated algorithms relative to IPC-MIN. Note that the undominated algorithms all use a theoretically inferior UNION technique, either LI or Rem.

Table 3: IPC relative performance

	PC	PH	PS	SP
IPC-LR	RemPS_4	RemPS_4	RemPS_4	╳
IPC-LS	RemPS_4	RemPS_4	RemPS_4	╳
IPC-LI	RemPS_4	undom.	undom.	╳
Rem	RemPS_4	╳	❹ undom.	undom.
IPC-TvL	RemPS_4	╳	RemPS_4	RemPS_4

Both new undominated IPC algorithms are among the overall best 11. In addition, IPC-LIPC from this IPC set of algorithms is in the overall best.

It is worthwhile to also consider here the impact of the IPC enhancement on a given algorithm. Table 4 addresses this, showing the average and range of percent



(a) Dominated by RemPS

(b) Undominated

Figure 2: Relative performance of IPC Union-Find algorithms

improvement using the IPC version of each algorithm over its non-IPC counterpart. For any algorithm where the minimum improvement is positive, using IPC consistently (i.e., for all input graphs) performed better than the corresponding non-IPC version. This is true for seven algorithms. Since Rem is, by definition, an IPC algorithm, there are no changes between its "IPC" and "non-IPC" versions. Hence, it is not included in the table. Note that every IPC algorithm is on average better than its corresponding non-IPC algorithm.

Table 4: Average (min, max) % improvement using IPC

	PC	PH	PS	SP
LR	29.92 (15.08, 47.48)	5.84 (-0.78, 15.53)	3.67 (-2.19, 12.53)	╳
LS	33.06 (20.75, 52.65)	6.96 (0.66, 19.84)	9.49 (0.74, 32.03)	╳
LI	33.25 (21.12, 49.46)	2.37 (-8.39, 14.57)	0.87 (-6.35, 7.3)	╳
TvL	38.46 (19.62, 60.65)	╳	6.36 (-4.94, 23.97)	22.20 (8.68, 39.74)

4.3 Interleaved Algorithms

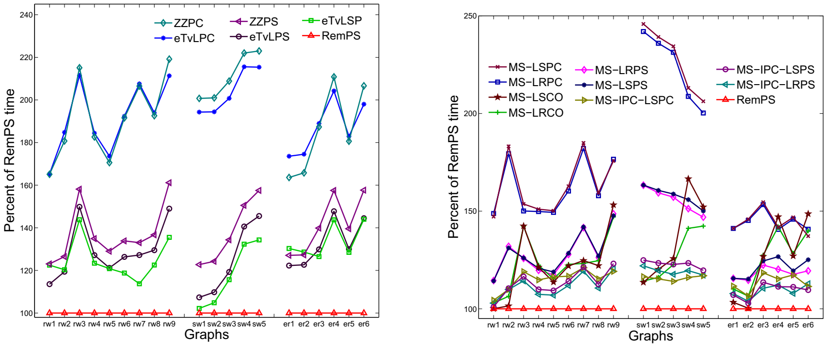
Recall that we included in our experiments two additional interleaved algorithms: eTvL, which uses the IPC test to control the main loop and ZZ, which has been used effectively for a parallel Union-Find implementation. In our experiments we see that both of these techniques are inferior to Rem on sequential machines. That is, RemPS dominates all five new Int algorithms. Table 5 demonstrates this and Figure 3(a) shows the performance of the newly dominated algorithms relative to RemPS.

Table 5: Int relative performance

	PC	PS	SP
Rem	RemPS$_4$	❹ undom.	undom.
TvL	LRPC$_1$	RemPS$_4$	LRPH$_2$
IPC-TvL	RemPS$_4$	RemPS$_4$	RemPS$_4$
eTvL	RemPS$_4$	RemPS$_4$	RemPS$_4$
zz	RemPS$_4$	RemPS$_4$	✕

Comparing eTvL and zz it is clear that the compression technique is more important than how the trees are traversed. While there is little difference for these algorithms between using Splicing and Path-Splitting, both of these compression techniques are better than using Path-Compression.

Interestingly, the only interleaved algorithms that make it into the 11 overall best algorithms are Rem's original algorithm, RemSP, and the variation on it that replaces Splicing with Path-Splitting.



(a) Int dominated by RemPS

(b) ms dominated by RemPS

Figure 3: Relative performance of Int and ms Union-Find algorithms

4.4 Memory-Smart Algorithms

Table 6 shows the relative performance of algorithms using the Memory-Smart enhancement. RemPS dominates all but one. Figure 3(b) shows the performance of the newly dominated algorithms relative to RemPS.

In addition to the undominated ms-ipc-lrpc, three algorithms from this subsection make it into the best 11 listed in the next subsection: ms-ipc-lrps, ms-ipc-lspc, and ms-ipc-lsps. The performance of the undominated algorithm

(MS-IPC-LRPC) is shown together with these other best algorithms in the next subsection.

Table 7 addresses the impact of the MEMORY-SMART (MS) enhancement on a given algorithm. Interestingly, all algorithms using PATH- COMPRESSION show consistent improvement with the MS enhancement, while the performance of MS implemented together with the other two compression techniques is inconsistent. No MS algorithm is consistently worse than its non-MS counterpart.

Table 6: MS relative performance

	PC	PS	CO
MS-LR	RemPS_4	RemPS_4	RemPS_4
MS-LS	RemPS_4	RemPS_4	RemPS_4
LI	RemPS_4	❺ undom.	undom.
MS-IPC-LR	undom.	RemPS_4	╳
MS-IPC-LS	RemPS_4	RemPS_4	╳
IPC-LI	RemPS_4	undom.	╳
Rem	RemPS_4	❹ undom.	╳

Table 7: Average (min, max) % improvement using MS

	PC	PS	CO
LR	15.17 (5.43, 27.21)	-1.49 (-24.92, 13.65)	2.81 (-9.01, 20.76)
LS	15.54 (4.34, 28.35)	3.51 (-16.58, 25.86)	3.18 (-6.52, 15.58)
IPC-LR	19.68 (10.85, 29.33)	10.30 (-6.66, 22.31)	╳
IPC-LS	16.01 (4.28, 26.57)	8.62 (-12.19, 22.49)	╳

4.5 The Fastest Algorithms

We now compare all algorithms using a different metric than the "dominates" technique used in the previous subsections. To compute the new metric we begin by calculating, for each input graph and each algorithm, its average run-time relative to the best average run-time for that graph (GLOBAL-MIN). We then average these percentages across each type of input graph (rw, sw, er). The result is three measures of goodness for each algorithm. Smaller values imply better algorithms for that type of input graph. An overall ranking and measure of goodness is then computed by averaging the three values.

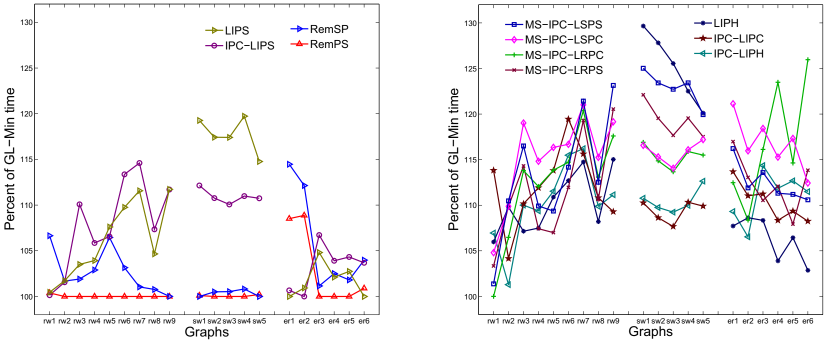
The results for the top 11 algorithms ranked using the overall average are given in Table 8. Each cell in the table contains both the algorithm's rank for the given type of graph and its relative timing reported as a percent of GLOBAL-MIN. The last row is included to show how far the last ranked algorithm is from the algorithms that are not in the top 11. Eleven was chosen as the cutoff rank

Table 8: Rank order(% of Global-Min) of the fastest algorithms based on graph type

	All graphs	Real-World	Small-World	Erdős-Rényi
RemPS	1 (101.17)	1 (100.41)	1 (100.06)	2 (103.05)
RemSP	2 (103.16)	2 (103.10)	2 (100.37)	4 (106.01)
IPC-LIPS	3 (107.48)	4 (108.29)	5 (110.94)	3 (103.22)
LIPS	4 (108.65)	3 (106.48)	8 (117.72)	1 (101.77)
IPC-LIPH	5 (110.70)	5 (110.59)	4 (110.48)	7 (111.04)
IPC-LIPC	6 (110.72)	8 (112.50)	3 (109.36)	6 (110.32)
LIPH	7 (114.02)	6 (110.62)	15 (125.13)	5 (106.31)
MS-IPC-LRPS	8 (114.56)	7 (112.01)	10 (119.28)	8 (112.41)
MS-IPC-LRPC	9 (115.01)	9 (112.83)	6 (115.37)	11 (116.84)
MS-IPC-LSPC	10 (116.08)	11 (115.63)	7 (115.84)	10 (116.76)
MS-IPC-LSPS	11 (116.33)	10 (113.62)	13 (122.91)	9 (112.47)
Fastest not listed	12 (126.64)	12 (119.52)	9 (117.93)	12 (121.84)

for best algorithms because there is a significant jump in values between the algorithms ranked 11 and 12; the values jump by 10.31% for all graphs, by 3.89% for rw graphs and by 5.00% for er graphs.

The relative performance of the algorithms with ranks 1-4 on the overall average is plotted in Figure 4(a) and the performance of the remaining seven fastest algorithms (those with ranks 5-11) is plotted in Figure 4(b). Both figures use the same vertical scale for easy comparison. The figure clearly shows that RemPS outperformed all other algorithms.



(a) Algorithms ranked 1 through 4

(b) Algorithms ranked 5 through 11

Figure 4: The eleven fastest Union-Find algorithms

The results show clearly that Rem is the best Union technique, and that it is best combined with either Path-Splitting or Splicing. The next best Union technique is Link-by-Index, with Path-Splitting doing better than any other compression technique. In this case, we see that ipc has a positive influence on lips overall, but not for rw or er. Link-by-Index remains strong even when combined with Path-Halving or Path-Compression. Only after these seven algorithms do we see any variation of the generally-accepted-as-best algorithms, combining either the Rank or Size Union technique with either Path-Splitting or Path-Compression. Moreover, these generally-accepted-as-best algorithms only do well when both the ipc and Memory-Smart enhancements are used.

In each of the following subsections we take a closer look at the results from a particular perspective. First, we use the dominates metric from Subsections 4.1 – 4.4 to annotate the results of this section. Next, we highlight the differences between the sw test set and the other two test sets. Last, we compare and contrast our results with related experimental studies in the literature.

4.5.1 Dominates verses Ranked Metrics

The dominates metric establishes a partial order of the algorithms based on consistent better performance. Using this metric, eight of the 63 algorithms were undominated. Six of those eight have ranks 1 – 5, and 9 for overall graphs using the average-of-averages metric of this section. This shows that generally the same algorithms come out at the top using both metrics.

The two undominated algorithms that did not make it into the 11 overall best are lrco and lico, which had ranks for all graphs (rw, sw, and er) of 22 (20, 29, and 29) and 34 (16, 46, and 33) respectively. This shows that while the Collapsing compression technique performs well for some graphs, it cannot be relied on to do well in general.

The dominates partial order of the best 11 algorithms is shown in Figure 5. Three algorithms, RemSP (rank 2), ipc-liph (rank 5) and ms-ipc-lrpc (rank 9), are undominated and do not dominate any other of the best 11 algorithms.

4.5.2 Different Behavior for Small World Graphs

While the results of our study consistently show that RemPS is among the two best algorithms for each test set (rw, sw, and er) and in general the same algorithms tend to do better for the rw and er test sets, the algorithms perform somewhat differently for the sw test set. This supports the general hypothesis that small world graphs exhibit different characteristics (and hence algorithms exhibit different behaviors on them) than might be seen in graphs that do not have the relatively small diameter that is the defining characteristic of a small world graph.

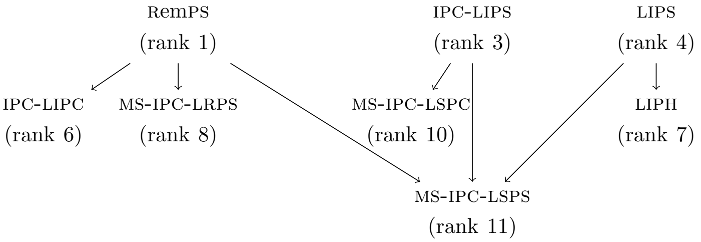


Figure 5: Dominates relationships among the 11 best algorithms. Algorithms not shown are undominated and do not dominate any other top 11 algorithm.

For the study here, this is due in part to the fact that the sw graphs had multiple components, and hence processed all edges in the sw graphs whereas the processing stopped for some of the other graphs once the algorithm determined that all nodes were already in a single connected component. This does not, however, account for all of the differences since one of the rw graphs and three of the er graphs had multiple components.

The differences in performance for sw graphs show up both in the overall ranking of algorithms and also in the plotted comparisons of algorithms using the dominates metric. With respect to the overall ranking, the fastest algorithm not in the table is faster than four of the algorithms in the table and only 0.21% slower than the algorithm ranked 8 (LIPS). Furthermore, IPC-LIPS and LIPS, although in the top four algorithms for rw and er, are not in the top four on the sw graphs. There is also a larger jump between the algorithms ranked second and third for sw (8.99% as compared to 4.32%, 3.38%, and 0.17%).

The different behavior for sw graphs is also highlighted using the dominates metric. For instance, Figure 1(f) clearly shows that for some sw graphs LICO requires almost twice as much time as the worst rw or er graph relative to Classical-Min. One can also see in the same figure that RemSP and RemPS dominate all other classical algorithms with respect to sw graphs. Also of note is the performance of MS-LRPC and MS-LSPC in Figure 3(b) where both algorithms are clearly substantially worse (relative to RemPS) for sw graphs than they are for rw and er graphs.

4.5.3 Comparison with Other Experimental Results in the Literature

In the following we compare our results and recommendations with other related experimental investigations.

The first such study that we are aware of was conducted by Liu [12]. This compared using LRNF with LRPC in an application from sparse matrix computations. The input consisted of grid graphs as well as real world matrices. The conclusion was that lrnf is about 50% better than lrpc.

Unlike the study by Liu we found that lrpc is about 50% better than lrnf.

In a similar study Gilbert et al. [10] compared a larger number of algorithms also using an application from sparse matrix computations. They implemented nlnf, nlpc, nlph, lrnf, lrpc, and lrph using sparse graphs as input. Their conclusion was that nl was better than lr for the Union operation, and that ph was a better compression technique than pc.

Unlike the study by Gilbert et al. we found that it it is always better to use the Union technique lr instead of nl when using either of ph, pc, or nf. But similarly to this study we got consistently better results when using ph over using pc.

The most comprehensive study to date was presented by Hynes in his master thesis [11]. This computed spanning trees on Erdős-Rényi graphs and compared all combinations of the following Union techniques nl, li, lr, and ls with each of the compression techniques co, pc, ps, and ph except for the combination lrco. The study found that lico was the best, closely followed by lsco. Of the remaining algorithms there was little difference between using li, lr, or ls. For the compression techniques pc was found to be worse than ph and ps, both of which gave similar results.

In our tests of the aforementioned algorithms lips and liph performed best and even made it into the overall top seven algorithms. In fact, for er graphs lips was the best algorithm. But li did not always outperform all other Union methods, the exception being when using co where lrco and lsco both were about 7% better than lico, contradicting the findings of Hynes. When comparing the compression technique we also found that pc was inferior to both ps and ph.

We are aware of two comparative studies concerning the use of Union-Find algorithms in image processing. In these studies the task is generally to extract connected regions from a bit-mapped image. Thus the input has very regular structure and the area that has to be searched for any pixel never exceeds the eight adjacent pixels. In [23] Wassenberg et al. compared algorithms using the following compression techniques: nf, pc, ps, ph, and co. The Union technique is not clearly specified but we believe it to be lr. Their ranking of the different Find techniques is the same as the order of listing, with co being between 1% and 7% faster than ph. In [24] Wu et al. compared lipc with lrpc and a version of lrpc which periodically flattens all tree structures (i.e. making all trees of height at most one). Their experiments concluded that that lipc was the method of choice.

Our study also gave the same ranking as Wasseberg et al. except that ps and ph has switched places, although they remain fairly similar (134% and 142% of Global-Min, respectively). Our results also agree with those of Wu et al. showing that lipc is better than lrpc.

The ipc technique was introduced by Osipov et al. in [15] where they compare

LRPC with IPC-LRPC in an application to compute a minimum weight spanning tree. Their conclusion is to use IPC-LRPC. Similarly, in our study we found that using IPC improved LRPC by 31%.

In a preliminary version of the current paper [16], we also concluded that RemPS and RemSP were the two best algorithms but with their order reversed. This is not surprising considering how close they are in performance (1.99% difference). Also, the earlier study did not include the non-interleaved LI algorithms, which make up five of the seven best algorithms in the current study.

4.6 Profile of the Fastest Algorithms

We collected profile statistics for the top 11 algorithms using the `cachegrind` profiler provided with the `valgrind` instrumentation framework [22].

For each algorithm and type of graph (rw, sw, er) we ran the algorithm on all graphs of that type using all five orderings and recorded the number of instruction reads (ir), the number of data reads (dr) and the number of data writes (dw) performed. The sum of these (ir + dr + dw) then represents the measure of work performed by that algorithm on that type of graph. Each of these numbers was then normalized by taking its percentage relative to the minimum value for that type of graph across all 11 algorithms (Profile-Min).

Note that Profile-Min is only relative to the top 11 algorithms, rather than all 63 algorithms as was the case for Global-Min. The other main difference in these profile computations from our overall fastest computations in the previous subsection is that here we effectively sum the operations across *all graphs of that type* before computing Profile-Min, whereas for Global-Min we computed the minimum and normalized for *each graph of that type* and then computed the average.

The result here is again three measures of goodness, one for each of rw, sw, and er, where smaller values closer to 100% are better. As with the timing results, we also average these for an overall measure of goodness. The resulting profile ranks and percent of Profile-Min are reported in Table 9. The rows of the table appear in the same order as they did in Table 8, i.e., in order of increasing timing ranks.

Note that while the profile ranks are not exactly the same as the timing ranks, they are still quite consistent. In particular, the top three profile ranked algorithms are the same as the top three timing ranked algorithms. With profile ranks, however, the jump between algorithms with rank 3 and 4 is more pronounced.

Figure 6 shows the profile rank of algorithms for each type of graph. The algorithms are arranged across the horizontal axes by increasing overall timing rank.

Table 9: Profile rank order (% of Profile-Min) of the fastest algorithms based on graph type

	All graphs	Real-World	Small-World	Erdős-Rényi
RemPS	2 (100.53)	1 (100.00)	2 (100.09)	3 (101.49)
RemSP	1 (100.45)	2 (100.06)	1 (100.00)	2 (101.30)
IPC-LIPS	3 (105.12)	3 (105.06)	3 (110.29)	1 (100.00)
LIPS	6 (123.54)	6 (123.71)	7 (133.90)	4 (113.02)
IPC-LIPH	5 (121.15)	5 (121.07)	5 (126.58)	5 (115.82)
IPC-LIPC	4 (118.56)	4 (118.09)	4 (117.03)	7 (120.56)
LIPH	11 (138.81)	11 (139.17)	11 (150.13)	9 (127.14)
MS-IPC-LRPS	7 (127.49)	7 (127.39)	9 (136.34)	6 (118.74)
MS-IPC-LRPC	10 (133.64)	10 (133.00)	8 (134.32)	10 (133.59)
MS-IPC-LSPC	9 (133.61)	8 (132.77)	6 (131.85)	11 (136.21)
MS-IPC-LSPS	8 (132.97)	9 (132.81)	10 (141.74)	8 (124.38)

4.7 Recommendations

The clear winner in our study is RemPS. This was the fastest algorithm for all graphs, rw and sw. On average it was 6.38% faster (with a range of −8.15% to 14.61%) compared to the best non-Rem algorithm and 2.06% faster (with a range of −0.21% to 6.06%) than the second best Rem algorithm. We believe that this is due to several factors: it has low memory overhead; Int algorithms perform less operations than other classical algorithms; it incorporates the ipc enhancement at every step of traversal, not only for the two initial nodes; and even when integrated with ps the algorithm is relatively simple with few conditional statements.

Considering all results, our overall recommendations are as follows.

- If possible, use the Rem Union technique.
- Link-by-Index is better than either Link-by-Rank or Link-by-Size.
- In general, Path-Splitting is the best compression technique.
- Path-Compression is more useful than Path-Halving.
- Both the Immediate-Parent-Check and Memory-Smart enhancements are important and should always be included.

5 Concluding Remarks

This paper reports the findings of 1600 experiments on each of 63 different Union-Find algorithms: 34 classical variations that were studied from a theoretical perspective up through the 1980s, 12 ipc variations, five more Int variations,

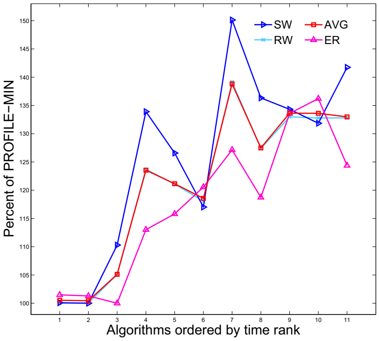


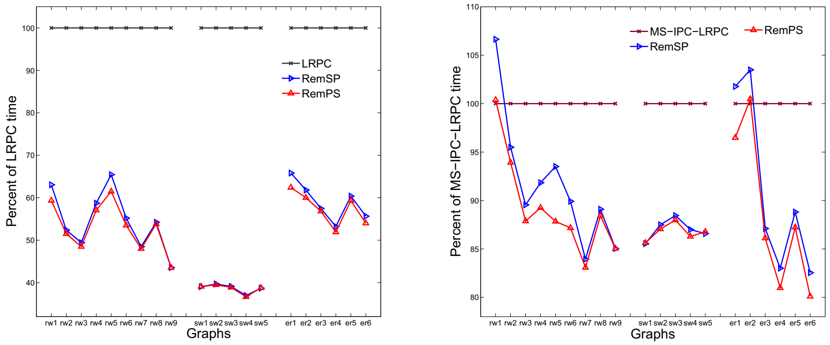
Figure 6: Profile rank by graph type of the eleven fastest algorithms

and 10 additional MS variations. In order to validate the results, we reran the 100,800 experiments on the same machine and ran it two more times on a similar machine. While the absolute times varied somewhat, the same set of algorithms had the top five ranks as did the set of algorithms with the top 10 ranks, and RemPS remained the clear top performer. We have also constructed spanning forests using both DFS and BFS, finding that use of the Union-Find algorithms are substantially more efficient.

The most significant result is that RemPS substantially outperforms LRPC even though RemPS is theoretically inferior to LRPC. This is even more surprising because LRPC is both simple and elegant, is well studied in the literature, is often implemented for real-world applications, and typically is taught as best in standard algorithms courses. In spite of this RemPS improved over LRPC by an average of 49.29%, with a range from 37.60% to 63.33%. Furthermore, it improved over LRPH (which others have argued uses the best one-pass compression technique) by an average of 26.96%, with a range from 14.08% to 46.03%.

Even when incorporating the MS and IPC enhancements, RemPS still improved over these other two classical algorithms on all inputs except two, where MS-IPC-LRPC improved over RemPS by only 0.38% and 0.47%. On average, RemPS improves over MS-IPC-LRPC by 11.60%, with a range from -0.47% to 19.90%. The savings incurred over RemPS and MS-IPC-LRPC are illustrated in Figure 7 where the times for the top two ranked algorithms are plotted relative to the times for LRPC and MS-IPC-LRPC.

To verify that our results hold regardless of the cache size, we ran experiments using twice, three times, and four times the memory for each node (simulating a smaller cache). The relative times for the algorithms under each of these scenarios were not significantly different than with the experiments reported here.



(a) Improvement over LRPC

(b) Improvement over MS-IPC-LRPC

Figure 7: Improvement of Rem algorithms over LRPC algorithms

We believe that our results should have implications for developers of software libraries like [6] and [14], which currently only implement LRPC and LRPH in the first case and LSPC in the second.

These Union-Find experiments were conducted under the guise of finding the connected components of a graph. As such, the sequences of operations tested were all Union operations as defined by the edges in graphs without multiedges. It would be interesting to study the performances of these algorithms for arbitrary sequences of intermixed Makeset, Union, and Find operations.

References

[1] W. Ackermann, *Zum hilbertschen aufbau der reellen zahlen*, Mathematische Annalen, 99 (1928), pp. 118–133.

[2] D. A. Bader and K. Madduri, *GTGraph: A synthetic graph generator suite*. http://www.cc.gatech.edu/~kamesh/GTgraph, 2006.

[3] L. Banachowski, *A complement to Tarjan's result about the lower bound on the complexity of the set union problem*, Information Processing Letters, 11 (1980), pp. 59–65.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, third ed., 2009.

[5] T. A. Davis, *The university of florida sparse matrix collection*, IEEE Transactions on Circuits and Systems. To appear.

[6] B. Dawes and D. Abrahams, *The Boost C++ libraries.* http://www.boost.org, 2009.

[7] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, 1976.

[8] Z. Galil and G. F. Italiano, *Data structures and algorithms for disjoint set union problems*, ACM Computing Surveys, 23 (1991), pp. 319–344.

[9] B. A. Galler and M. J. Fisher, *An improved equivalence algorithm*, Communications of the ACM, 7 (1964), pp. 301–303.

[10] J. R. Gilbert, E. G. Ng, and B. W. Peyton, *An efficient algorithm to compute row and column counts for sparse Cholesky factorization*, Journal on Matrix Analysis and Applications, 15 (1994), pp. 1075–1091.

[11] R. Hynes, *A new class of set union algorithms*, Master's thesis, Department of Computer Science, University of Toronto, Canada, 1998.

[12] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[13] F. Manne and M. M. A. Patwary, *A scalable parallel union-find algorithm for distributed memory computers*, in proceedings of the Eighth International Conference on Parallel Processing and Applied Mathmatics (PPAM 2009), vol. 6067 of LNCS, Springer Verlag, 2009, pp. 186–195.

[14] K. Melhorn and S. Näher, *LEDA, A Platform for Combinatorial Geometric Computing*, Cambridge University Press, 1999.

[15] V. Osipov, P. Sanders, and J. Singler, *The filter-Kruskal minimum spanning tree algorithm*, in proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2009), 2009, pp. 52–61.

[16] M. M. A. Patwary, J. R. S. Blair, and F. Manne, *Experiments on union-find algorithms for the disjoint-set data structure*, in proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010), vol. 6049 of LNCS, Springer Verlag, 2010, pp. 411–423.

[17] J. A. L. Poutré, *Lower bounds for the union-find and the split-find problem on pointer machines*, Journal of Computer and System Sciences, 52 (1996), pp. 87–99.

[18] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM, 22 (1975), pp. 215–225.

[19] ——, *A class of algorithms which require nonlinear time to maintain disjoint sets*, Journal of Computer and System Sciences, 18 (1979), pp. 110–127.

[20] ——, *Data structures and network algorithms*, SIAM, 1983.

[21] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, Journal of the ACM, 31 (1984), pp. 245–281.

[22] Valgrind. http://valgrind.org/, 2010.

[23] J. Wassenberg, D. Bulatov, W. Middelmann, and P. Sanders, *Determination of maximally stable extremal regions in large images*, in proceeding of the Signal Processing, Pattern Recognition, and Applications (SPPRA 2008), Acta Press, 2008.

[24] K. Wu, E. Otoo, and K. Suzuki, *Optimizing two-pass connected-component labeling algorithms*, Pattern Analysis and Applications, 12 (2009), pp. 117–135.

A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers

Fredrik Manne Md. Mostofa Ali Patwary*

Abstract

The Union-Find algorithm is used for maintaining a number of non-overlapping sets from a finite universe of elements. The algorithm has applications in a number of areas including the computation of spanning trees, sparse linear algebra, and in image processing.

Although the algorithm is inherently sequential there has been some previous efforts at constructing parallel implementations. These have mainly focused on shared memory computers. In this paper we present the first scalable parallel implementation of the Union-Find algorithm suitable for distributed memory computers. Our new parallel algorithm is based on an observation of how the Find part of the sequential algorithm can be executed more efficiently.

We show the efficiency of our implementation through a series of tests to compute spanning forests of very large graphs.

Keywords: disjoint sets, parallel Union-Find, distributed memory.

1 Introduction

The disjoint-set data structure is used for maintaining a number of non-overlapping sets consisting of elements from a finite universe. Its uses include among other, image decompositions, the computation of connected components and minimum spanning trees in graphs, and is also taught in most algorithm courses. The algorithm for implementing this data structure is often referred to as the Union-Find algorithm.

More formally, let U be a collection of n distinct elements and let S_i denote a set of elements from U. Two sets $\{S_1, S_2\}$ are disjoint if $S_1 \cap S_2 = \emptyset$. A disjoint set data structure maintains a collection $\{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets selected from U. Each set is identified by a representative x, which is usually

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, {fredrikm,Mostofa.Patwary}@ii.uib.no

some member of the set. The two main operations are then to FIND which set a given element belongs to by locating its representative element and also to create a new set from the UNION of two existing sets.

The underlying data structure of each set is typically a rooted tree where the element in the root vertex is the representative of the set. Using the two techniques LINK-BY-RANK and PATH-COMPRESSION the running time of any combination of m UNION and FIND operations is $O(m\alpha(m,n))$ where α is the very slowly growing inverse Ackerman function [1].

From a theoretical point of view using the UNION-FIND algorithm is close to optimal. However, for very large problem instances such as those that appear in scientific computing this might still be too slow. This is especially true if the UNION-FIND computation is applied multiple times and take up a significant amount of the total time. It might also be that the underlying problem is too large to fit in the memory of one processor. One recent application that makes use of repeated applications of the UNION-FIND algorithm is a new algorithm for computing Hessian matrices using substitution methods [7]. Hence, designing parallel algorithms is necessary to keep up with the very large problem instances that appear in scientific computing.

The first such effort was by Cybenko et al. [5] who presented an algorithm using the UNION-FIND algorithm for computing the connected components of a graph and gave implementations both for shared memory and distributed memory computers. The distributed memory algorithm duplicates the vertex set and then partitions the edge set among the processors. Each processor then computes a spanning forest using its local edges. In $\log p$ steps, where p is the number of processors, these forests are then merged until one processor has the complete solution. However, the experimental results from this algorithm were not promising and showed that for a fixed size problem the running time increased with the number of processors used.

Anderson and Woll also presented a parallel UNION-FIND algorithm using *wait-free* objects suitable for shared memory computers [2]. In doing so they showed that parallel algorithms using concurrent UNION-operations risk creating unbalanced trees. However, they did not produce any experimental results for their algorithm.

We note that there exists an extensive literature on designing parallel algorithms for computing a spanning forest or the connected components of a graph. However, up until the recent paper by Bader and Cong [4] such efforts had failed to give speedup on arbitrary graphs. In [4] the authors present a novel scalable parallel algorithm for computing spanning forests on a shared memory computer.

Focusing on distributed memory computers is of importance since these have better scalability than shared memory computers and thus the largest systems tend to be of this type. However, their higher latency makes distributed memory computers more dependent on aggregating sequential work through the exploitation of locality.

The current work presents a new parallel UNION-FIND algorithm for distributed memory computers. The algorithm operates in two stages. In the first stage each processor performs local computations in order to reduce the number of edges that need to be considered for inclusion in the final spanning tree. This is similar to the approach used in [5], however, we use a sequential UNION-FIND algorithm for this stage instead of BFS. Thus when we start the second parallel stage each processor has a UNION-FIND type forest structure that spans each local component.

In the second stage we merge these structures across processors to obtain a global solution. In both the sequential and the parallel stage we make use of a novel observation on how the UNION-FIND algorithm can be implemented. This allows both for a faster sequential algorithm and also to reduce the amount of communication in the second stage.

To show the feasibility and efficiency of our algorithm we have implemented several variations of it on a parallel computer using C++ and MPI and performed tests to compute spanning trees of very large graphs using up to 40 processors. Our results show that the algorithm scales well both for real world graphs and also for small-world graphs.

The rest of the paper is organized as follows. In Section 2 we briefly explain the sequential algorithm and also how this can be optimized. In Section 3 we describe our new parallel algorithm, before giving experimental results in Section 4.

2 The Sequential Algorithm

In the following we first outline the standard sequential UNION-FIND algorithm. We then point out how it is possible to speed up the algorithm by paying attention to the rank values. This is something that we will make use of when designing our parallel algorithm.

The standard data structure for implementing the UNION-FIND algorithm is a forest where each tree represents a connected set. To implement the forest each element x has a pointer $p(x)$ initially set to x. Thus each x starts as a set by itself. The two operations used on the sets are then FIND(x) and UNION(x, y) where x and y are distinct elements. FIND(x) returns the root of the tree that x belongs to. This is done by following pointers starting from x. UNION(x, y) merges the two trees that x and y belong to. This is achieved by making one of the roots of x and y point to the other. With these operations the connected components of a graph $G(V, E)$ can be computed as shown in Algorithm 1.

When the algorithm terminates the vertices of each tree will consist of a connected component and the set of edges in S define a spanning forest on G.

Algorithm 1 The sequential Union-Find algorithm

```
$S \leftarrow \emptyset$
**for** each $x \in V$ **do**
    $p(x) \leftarrow x$
**for** each $(x, y) \in E$ **do**
    **if** Find$(x) \neq$ Find$(y)$ **then**
        Union$(x, y)$
        $S \leftarrow S \cup \{(x, y)\}$
```

There are two standard techniques for speeding up the Union-Find algorithm. The first is Link-by-Rank. Here each vertex is initially given a rank of 0. If two sets are to be merged where the root elements are of equal rank then the rank of the root element of the combined set will be increased by one. In all other Union operations the root with the lowest rank will be set to point to the root with the higher rank while all ranks remain unchanged. Note that this ensures that the parent of a vertex x will always have higher rank than the vertex x itself.

The second technique is Path-Compression. In its simplest form, following any Find operation, all traversed vertices will be set to point to the root. This has the effect of compressing the path and making subsequent Find operations using any of these vertices faster. Note that even when using Path-Compression the rank values will still be strictly increasing when moving upwards in a tree.

Using the techniques of Link-by-Rank and Path-Compression the running time of any combination of m Union and Find operations is $O(m\alpha(m, n))$ where α is the very slowly growing inverse Ackerman function [1].

We now consider how it is possible to implement the Union-Find algorithm in a more efficient way. It is straight forward to see that one can speed up Algorithm 1 by storing the results of the two Find operations and use these as input to the ensuing Union operation which then only has to determine which of the two root vertices should point to the other.

In the following we describe how it is possible in certain cases to terminate the Find operation before reaching the root. Let the rank of a vertex z be denoted by $rank(z)$. Consider two vertices x and y belonging to different sets with roots r_x and r_y respectively where $rank(r_x) < rank(r_y)$. If we Find r_x before r_y then it is possible to terminate the search for r_y as soon as we reach an ancestor z of y where $rank(z) = rank(r_x)$. This follows since the rank function is strictly increasing and we must therefor have $rank(r_y) > rank(r_x)$ implying that $r_y \neq r_x$. At this point it is possible to join the two sets by setting $p(r_x) \leftarrow p(z)$. Note that this will neither violate the rank property nor will it increase the asymptotic time bound of the algorithm. However, if we perform Find(y) before Find(x) we will not be able to terminate early. To avoid this we perform the two Find operations in an interleaved fashion by always continuing the search from the vertex with the lowest current rank. In this way the FIND operation can terminate as soon as one reaches the root with the smallest rank. We label this as the ZIGZAG FIND operation as opposed to the *classical* FIND operation.

The ZIGZAG FIND operation can also be used to terminate the search early when the vertices x and y belong to the same set. Let z be their lowest common ancestor. Then at some stage of the ZIGZAG FIND operation the current ancestors of x and y will both be equal to z. At this point it is clear that x and y belong to the same set and the search can stop.

We note that the ZIGZAG FIND operation is similar to the *contingently unite* algorithm as presented in [9], only that we have extracted out the specifics of the PATH-COMPRESSION technique.

3 The Parallel Algorithm

In the following we outline our new parallel UNION-FIND algorithm. We assume a partitioning of both the vertices and the edges of G into p sets each, $V = \{V_0, V_1, \ldots, V_{p-1}\}$ and $E = \{E_0, E_1, \ldots, E_{p-1}\}$ with the pair (V_i, E_i) being allocated to processor i, $0 \leq i < p$. If $v \in V_i$ (or $e \in E_i$) processor i *owns* v (or e) and v (or e) is *local* to processor i.

Any processor i that has a local edge $(v, w) \in E_i$ such that it does not own vertex v will create a *ghost vertex* v' as a substitution for v. We denote the set of ghost vertices of processor i by V_i'. Thus an edge allocated to processor i can either be between two vertices in V_i, between a vertex in V_i and a vertex in V_i', or between two vertices in V_i'. We denote the set of edges adjacent to at least one ghost vertex by E_i'.

The algorithm operates in two stages. In the first stage each processor performs local computations without any communication in order to reduce the number of edges that need to be considered for the second final parallel stage.

Stage 1. Reducing the input size

Initially in Stage 1 each processor i computes a spanning forest T_i for its local vertices V_i using the local edges $E_i - E_i'$. This is done using a sequential UNION-FIND algorithm. It is then clear that T_i can be extended to a global spanning forest for G.

Next, we compute a subset T_i' of E_i' such that $T_i \cup T_i'$ form a spanning forest for $V_i \cup V_i'$. Without going into the details we note that T_i' can be computed efficiently without destroying the structure of T_i. The remaining problem is now to select a subset of the edges in T_i' so as to compute a global spanning forest for G.

Stage 2. Calculating the final spanning forest

The underlying data structure for this part of the algorithm is the same as for the sequential Union-Find algorithm, only that we now allow trees to span across several processors. Thus a vertex v can set $p(v)$ to point to a vertex on a different processors other than its own. The pointer $p(v)$ will in this case contain information about which processor owns the vertex being pointed to, its local index on that processor, and also have a lower bound on its rank. Each ghost vertex v' will initially set $rank(v') \leftarrow 0$ and $p(v') \leftarrow v$. Thus the connectivity of v' is initially handled through the processor that owns v. For the local vertices the initial $p()$ values are as given from the computation of T_i.

We define the *local* root $l(v)$ as the last vertex on the find-path of v that is stored on the same processor as v. If in addition $l(v)$ has $p(l(v)) = l(v)$ then $l(v)$ is also a *global* root.

In the second stage of the algorithm processor i iterates through each edge $(v, w) \in T'_i$ to determine if this edge should be part of the final spanning forest or not. This is done by issuing a Union-Find query (UF) for each edge. A UF-query can either be resolved internally by the processor or it might have to be sent to other processors before an answer is returned. To avoid a large number of small messages a processor will process several of its edges before sending and receiving queries. A computation phase will then consist of first generating new UF-queries for a predefined number of edges in T'_i and then to handle incoming queries. Any new messages to be sent will be put in a queue and transmitted in the ensuing communication phase. Note that a processor might have to continue processing incoming queries after it has finished processing all edges in T'_i.

In the following we describe how the UF-queries are handled. A UF-query contains information about the edge (v, w) in question and also to which processor it belongs. In addition the UF-query contains two vertices a and b such that a and b are on the find-paths of v and w respectively. The query also contains information about the rank of a and b and if either a or b is a global root. Initially $a = v$ and $b = w$.

When a processor receives (or initiates) a UF-query it is always the case that it owns at least one of a and b. Assume that this is a, we then label a as the *current* vertex. Then a is first replaced by $p(l(a))$. There are now three different ways to determine if (v, w) should be part of the spanning forest or not: *i)* If $a = b$ then v and w have a common ancestor and the edge should be discarded. *ii)* If $a \neq b$, $p(a) = a$, and $rank(a) < rank(b)$ then $p(a)$ can be set to b and thus including (v, w) in the spanning forest. *iii)* If $a \neq b$, $rank(a) = rank(b)$, $p(a) = a$, while b is marked as also being a global root then $p(a)$ can be set to b while a message is sent to b to increase its rank by one.

To avoid that a and b concurrently sets each other as parents in Case *iii)* we associate a unique random number $r()$ with each vertex. Thus we must also have $r(a) < r(b)$ before we set $p(a) \leftarrow b$.

If a processor i reaches a decision on the current edge (v, w), it will send a message to the owner of the edge about the outcome. Otherwise processor i will forward the updated UF-query to a processor j (where $j \neq i$) such that j owns at least one of a and b.

In the following we outline two different ways in which the UF-queries can be handled. The difference lies mainly in the associated communication pattern and reflects the classical as opposed to the ZIGZAG UNION-FIND operation as outlined in Section 2.

In the classical parallel UNION-FIND algorithm a is initially set as the current vertex. Then while $a \neq p(a)$ the query is forwarded to $p(a)$. When the query reaches a global root, in this case a, then if b is marked as also being a global root, rules *i)* through *iii)* are applied. If these result in a decision such that the edge is either discarded or $p(a)$ is set to b then the query is terminated and a message is sent back to the processor owning the edge in question. Otherwise, the query is forwarded to b where the process is repeated (but now with b as the current vertex).

In the parallel ZIGZAG algorithm a processor that initiates or receives a UF-query will always check all three cases after first updating the current vertex z with $l(z)$. If none of these apply the query is forwarded to the processor j which owns the one of a and b marked with the lowest rank and if $rank(a) = rank(b)$ the one with lowest r value. Note that if v and w are initially in the same set then a query will always be answered as soon as it reaches the processor that owns the lowest common ancestor of v and w. Similarly, if v and w are in different sets the query will be answered as soon as the query reaches the global root with lowest rank.

Since UF-queries are handled concurrently it is conceivable that a vertex $z \in \{a, b\}$ has seized to be a global root when it receives a message to increase its rank (if Case *iii)* has been applied). To ensure the monotonicity of ranks z then checks, starting with $w = p(z)$, that $rank(w)$ is strictly greater than the updated rank of z. If not we increase $rank(w)$ by one and repeat this for $p(w)$. Note that this process can lead to extra communication.

Similarly as for the algorithm in [2] it is possible that unbalanced trees are created with both parallel communication schemes. This can happen if more than two trees with the same rank are merged concurrently such that one hangs of the other.

When a processor i receives a message that one of its edges (v, w) is to be part of the spanning forest it is possible to initiate a PATH-COMPRESSION operation between processors. On processor i this would entail to set $l(v)$ (and $l(w)$) to point to the new root which would then also have to be included in the return message. Since there could be several such incoming messages for $l(v)$ and these could arrive in an arbitrary order we must first check that the rank of the new root is larger than the rank that i has stored for $p(l(v))$ before performing the compression. If this is the case then it is possible to continue the compression by sending a message to $p(l(v))$ about the new root. We label these schemes as either 1-level or full Path-Compression.

4 Experiments

For our experiments we have used a Cray XT4 distributed memory parallel machine with AMD Opteron quad-core 2.3 GHz processors where each group of four cores share 4 GB of memory. The algorithms have been implemented in C++ using the MPI message-passing library. We have performed experiments using both graphs taken from real application as well as on different types of synthetic graphs. In particular we have used application graphs from areas such as linear programming, medical science, structural engineering, civil engineering, and automotive industry [6, 8]. We have also used small-world graphs as well as random graphs generated by the GTGraph package [3].

Table 1 give properties of the graphs. The first nine rows contains information about the application graphs while the final two rows give information about the small-world graphs. The first 5 columns give structural properties about the graphs while the last two columns show the time in seconds for computing a spanning forest using Depth First Search (DFS) and the sequential Zigzag algorithm (ZZ). We have also used two random graphs both containing one million vertices and respectively, 50 and 100 million edges. Note that all of these graphs only contains one component. Thus the spanning forest will always be a tree.

Name	$\|V\|$	$\|E\|$	Max Deg	Avg Deg	DFS	ZZ
m_t1	97,578	4,827,996	236	98.95	0.12	0.06
cranksg2	63,838	7,042,510	3,422	220.64	0.15	0.03
inline_1	503,712	18,156,315	842	72.09	0.57	0.26
ldoor	952,203	22,785,136	76	47.86	0.71	0.47
af_shell10	1,508,065	25,582,130	34	33.93	1.04	0.37
boneS10	914,898	27,276,762	80	59.63	0.86	0.38
bone010	986,703	35,339,811	80	71.63	1.05	0.47
audi	943,695	38,354,076	344	81.28	1.20	0.33
spal_004	321,696	45,429,789	6,140	282.44	1.33	0.66
rmat1	377,823	30,696,982	8,109	162.49	2.07	1.34
rmat2	504,817	40,870,608	10,468	161.92	2.71	1.81

Table 1: Properties of the graphs

Our first results concern the different sequential algorithms for computing a spanning forest. As is evident from Table 1 the Zigzag algorithm outperformed the DFS algorithm. A comparison of the different sequential Union-Find algo-

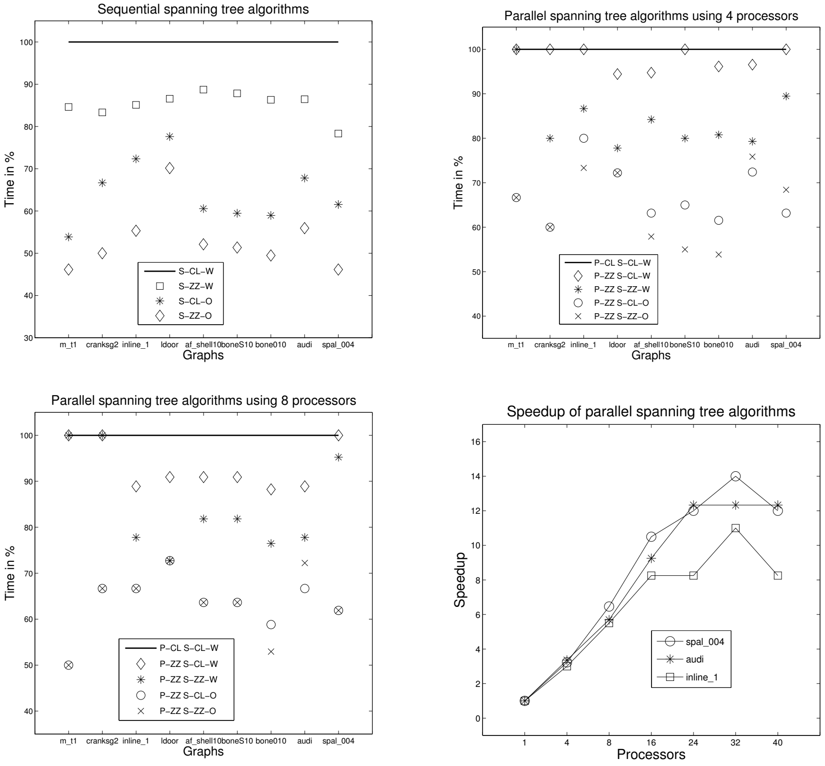


Figure 1: Performance results: S - Sequential algorithm, P- Parallel algorithm, CL - Classical Union-Find, ZZ - Zigzag Union-Find, W - With Path-Compression, O - Without Path-Compression.

rithms on the real world graphs is shown in the upper left quadrant of Figure 1. All timings have been normalized relative to the slowest algorithm, the classical algorithm (CL) using Path-Compression (W). As can be seen, removing the Path-Compression (O) decreases the running time. Also, switching to the Zigzag algorithm (ZZ) improves the running time further, giving approximately a 50% decrease in the running time compared to the classical algorithm with Path-Compression. To help explain these results we have tabulated the number of "parent chasing" operations on the form $z = p(z)$. These show that the Zigzag algorithm only executes about 10% as many such operations as the classical algorithm. However, this does not translate to an equivalent speed up due to the added complexity of the Zigzag algorithm.

The performance results for the synthetic graphs give an even more pronounced improvement when using the Zigzag algorithms. For these graphs both Zigzag algorithms outperforms both classical algorithms and the Zigzag algorithm without Path-Compression gives an improvement in running time of close to 60% compared to the classical algorithm with Path-Compression.

Next, we present the results for the parallel algorithms. For these experiments we have used the Mondrian hypergraph partitioning tool [10] for partitioning vertices and edges to processors. For most graphs this has the effect of increasing locality and thus enabling to reduce the size of T'_i in Stage 1. In our experiments $T' = \cup_i T'_i$ contained between 0.1% and 0.5 % of the total number of edges for the application graphs, between 1 % and 6 % for the small-world graphs, and between 2 % and 36 % for the random graphs. As one would expect these numbers increase with the number of processors.

In our experiments we have compared using either the classical or the Zigzag algorithm, both for the sequential computation in Stage 1 and also for the parallel computation in Stage 2. We note that in all experiments we have only used level-1 Path-Compression in the parallel algorithms as using full compression, without exception, slowed down the algorithms.

How the improvements from the sequential Zigzag algorithm are carried into the parallel algorithm can be seen in the upper right and lower left quadrant of Figure 1. Here we show the result of combining different parallel algorithms with different sequential ones when using 4 and 8 processors. All timings have again been normalized to the slowest algorithm, the parallel classical algorithm (P-CL) with the sequential classical algorithm (S-CL), and using Path-Compression (W). Replacing the parallel classical algorithm with the parallel Zigzag algorithm while keeping the sequential algorithm fixed gives an improvement of about 5% when using 4 processors. This increases to 14% when using 8 processors, and to about 30% when using 40 processors. This reflects how the running time of Stage 2 of the algorithms becomes more important for the total running time as the number of processors are increased.

The total number of sent and forwarded UF-queries is reduced by between 50% and 60% when switching from the parallel classical to the parallel Zigzag algorithm. Thus this gives an upper limit on the possible gain that one can obtain from the parallel Zigzag algorithm over the parallel classical algorithm.

When keeping the parallel Zigzag algorithm fixed and replacing the sequential algorithm in Step 1 we get a similar effect as we did when comparing the sequential algorithms, although this effect is dampened as the number of processors is increased and Step 1 takes less of the overall running time.

The figure in the lower right corner shows the speedup on three large matrices when using the best combination of algorithms, the sequential and parallel Zigzag algorithm. As can be seen the algorithm scales well up to 32 processors at which point the communication in Stage 2 dominates the algorithm and causes a slowdown. Similar experiments for the small-world graphs showed a more moderate speedup peaking at about a factor of four when using 16 processors. The random graphs did not obtain speedup beyond 8 processors and even for this configuration the running time was still slightly slower than for the best sequential algorithm. We expect that the speedup would continue beyond the current numbers of processors for sufficiently large data sets.

To conclude we note that the Zigzag Union-Find algorithm achieves considerable savings compared to the classical algorithm both for the sequential and the parallel case. However, our parallel implementation did not achieve speedup for the random graphs, as was the case for the shared memory implementation in [4]. This is mainly due to the poor locality of such graphs.

References

[1] W. Ackermann, *Zum hilbertschen aufbau der reellen zahlen*, Mathematische Annalen, 99 (1928), pp. 118–133.

[2] R. J. Anderson and H. Woll, *Wait-free parallel algorithms for the union-find problem*, in proceedings of the 23rd ACM Symposium on Theory of Computing (STOC 91), 1991, pp. 370–380.

[3] D. A. Bader and K. Madduri, *GTGraph: A synthetic graph generator suite.* http://www.cc.gatech.edu/˜kamesh/GTgraph, 2006.

[4] D. J. Bader and G. Cong, *A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs)*, Journal of Parallel and Distributed Computing, 65 (2005), pp. 994–1006.

[5] G. Cybenko, T. G. Allen, and J. E. Polito, *Practical parallel algorithms for transitive closure and clustering*, International Journal of Parallel Computing, 17 (1988), pp. 403–423.

[6] T. A. Davis, *The university of florida sparse matrix collection*, IEEE Transactions on Circuits and Systems. To appear.

[7] A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen, *New acyclic and star coloring algorithms with applications to computing Hessians*, SIAM Journal on Scientific Computing, 29 (2007), pp. 1042–1072.

[8] J. Koster, *Parasol matrices.* http://www.parallab.uib.no/projects/, 1999.

[9] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, Journal of the ACM, 31 (1984), pp. 245–281.

[10] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

III

Parallel Greedy Graph Matching using an Edge Partitioning Approach

Md. Mostofa Ali Patwary* Rob H. Bisseling**
Fredrik Manne*

Abstract

We present a parallel version of the Karp–Sipser graph matching heuristic for the maximum cardinality problem. It is bulk-synchronous, separating computation and communication, and uses an edge-based partitioning of the graph, translated from a two-dimensional partitioning of the corresponding adjacency matrix. It is shown that the communication volume of Karp–Sipser graph matching is proportional to that of parallel sparse matrix–vector multiplication (SpMV), so that efficient partitioners developed for SpMV can be used. The algorithm is presented using a small basic set of 7 message types, which are discussed in detail. Experimental results show that for most matrices, edge-based partitioning is superior to vertex-based partitioning, in terms of both parallel speedup and matching quality. Good speedups are obtained on up to 64 processors.

Keywords: Bulk-synchronous parallel, heuristics, matching, partitioning, sparse matrix.

1 Introduction

Let $G = (V, E)$ be a graph with vertex set V and edge set E. A *matching* $M \subseteq E$ is a pairing of adjacent vertices such that each vertex is matched with at most one other vertex. The objective of maximum cardinality matching is to match as many vertices as possible. In this paper, we investigate the parallelization of one particular algorithm for maximum cardinality matching, the Karp–Sipser algorithm [9], which is a heuristic that has been shown in practice to yield high-quality matchings quickly [15]. Heuristic matching algorithms are often the common choice in practical applications as they are much faster for large

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, {Mostofa.Patwary,fredrikm}@ii.uib.no

**Department of Mathematics, Utrecht University, the Netherlands, R.H.Bisseling@uu.nl

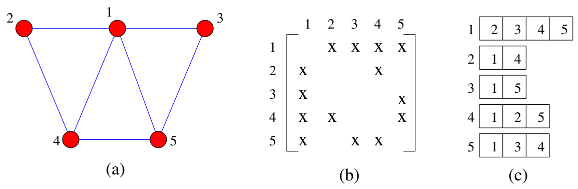


Figure 1: *cage3* matrix of size 5 × 5 [6]. (a) The graph representation, (b) the corresponding adjacency matrix where each x represents a nonzero, and (c) the rowcols $\{1, 2, \ldots, 5\}$.

problem sizes than optimal algorithms, and because they are easier to implement and parallelize. For bipartite graphs, it has been shown [12] that Karp–Sipser outperforms other heuristic algorithms such as minimum-degree matching. This motivates our choice to parallelize the Karp–Sipser algorithm.

We view the graph $G = (V, E)$ as an adjacency matrix A of size $n \times n$ where $n = |V|$ and where for each edge $(i, j) \in E$, A has two nonzeros a_{ij} and a_{ji} such that $a_{ij} = a_{ji}$. In matrix language, our problem then is to find a matching of maximum cardinality among the rows. Since the problem is unweighted, we assume that the numerical value $a_{ij} = 1$ for all nonzeros. The adjacency list of a vertex $i \in V$ is equivalent to row i of A and column i of A. We note that A is symmetric and $a_{ii} = 0$ for all $i = 1, 2, \ldots, n$. (We number vertices/rows from 1 onwards.) In sparse matrix computations, it has been customary already for many years to view matrices for certain purposes as graphs, see e.g. [7], but in this paper we will exploit the reverse connection, by viewing a graph as a matrix to benefit from sparse matrix partitioning methodology for the purpose of parallelizing a graph algorithm.

Since row i is identical to column i, instead of mentioning row and column i separately, we sometimes call them together *rowcol* i. We maintain only one adjacency list for both of them. The list contains all entries $\{a_{ij} : 1 \leq j \leq n \text{ and } a_{ij} \neq 0\}$. We get the nonzeros of row i or column i by accessing the entries in rowcol i. Figures $1(a)$ and $1(b)$ show the transformation of a graph to an adjacency matrix and Figure $1(c)$ shows the corresponding rowcols, which can also be seen as adjacency lists.

Our parallelization of the Karp–Sipser algorithm will be done in bulk-synchronous parallel (BSP) style [19] (see also [3]), which is characterized by alternating between computation phases and communication phases, each ended by a global barrier synchronization; these phases are commonly called *supersteps*. A computation phase uses only locally available data and can last as long as there is something to compute locally, but it can also be terminated earlier, for instance after a fixed amount of work. This enhances load balancing by detecting at an earlier stage that a processor has run out of work. The BSP style gives a high-level framework for algorithmic development, which eases parallelization of irregular algorithms such as graph algorithms. Examples where this style has been employed are graph coloring [4], edge-weighted graph matching [14], and single-source shortest paths [13].

An advantage of using BSP at a programming level is that the BSPlib communication library [8] takes some of the tediousness away of message-passing for irregular computations; in particular, the bulk-synchronous message passing primitive `bsp_send` is helpful, as it allows sending data to an arbitrary processor without the need for a corresponding receive request. The data is sent to a remote buffer, which can be emptied at the next superstep. BSPlib is available on almost all computer architectures, through a library called BSPonMPI [17], which can be linked to the program, thus effectively turning it into an MPI program. Another advantage of using BSP is that many communication optimizations can be left to the system; for instance, different messages to the same destination are automatically detected and combined. One goal of this paper is to demonstrate how to parallelize a graph algorithm with a high level of irregularity by using BSP.

Instead of using BSP, we could also use message passing, immediately sending matching data once they become available. To make use of this, for instance to improve the quality of the matching, these data must also be received as soon as possible, requiring frequent polling for incoming messages (assuming communication is nonblocking). This would increase communication time and incur more message latency costs, and it would remove the global notion of time that is given by the supersteps of the BSP model.

A fundamental question when parallelizing an algorithm for a distributed-memory computer is how to distribute the data among the processors. A common approach for graph algorithms has been to partition the vertices and then assign each resulting part to a processor (assigning the edges in a corresponding manner). In matrix terms, this leads to a one-dimensional row distribution. Often, the graph has been partitioned beforehand using software such as Metis [10] or Scotch [16]. This has also been the approach taken in our previous work [14].

An alternative approach would be to partition the edges instead of the vertices. In matrix terms, this leads to a general two-dimensional distribution, with in principle a larger space of possible solutions. For parallel sparse matrix–vector multiplication, this is known to lead to much lower communication volumes for certain types of matrices such as web-link matrices, originating outside the traditional application area of Finite Element Methods, see [18, 20]. These matrices from nontraditional areas often have rows and columns with widely varying numbers of nonzeros. One-dimensional methods avoid communication in one direction, but often pay a heavy price in the other direction, especially for relatively dense rows or columns. Two-dimensional methods are able to handle these much better. For graph algorithms, as far as we know, two-dimensional methods have not been employed yet. One of our goals is thus to investigate whether the edge-partitioning approach yields similar benefits as in the matrix–vector case.

We now define some notations that we use throughout the paper. The number of nonzeros in a row i of A is denoted by nz_i. We call a row i a *singleton* if $nz_i = 1$. We let the matching algorithm use p processors denoted by $P_0, P_1, \ldots, P_{p-1}$. We use the two-dimensional partitioning approach of the Mondriaan package [20] to distribute A symmetrically among p processors before the matching starts. This means that a_{ij} and a_{ji} are assigned to the same processor. The nonzeros of a row i could be distributed among several processors, say, q_i processors. Let $lc_{i,s}$ denote the local number of nonzeros of row i in P_s. We choose one of the q_i processors as the *owner* of i, denoted by $P(i)$, and the other $q_i - 1$ processors as the *nonowners* of i, given by the set $nonOwners(i)$, where each nonowner is denoted by $P'(i)$. Note that $P(i)$ stores $nonOwners(i)$ and nz_i, whereas each $P'(i)$ knows only about $P(i)$. Both the owner and the nonowners maintain their value of $lc_{i,s}$. We use $isMatched(i)$ for the status of matching (either *true* or *false*) and $m(i)$ for the matching partner; both are stored at $P(i)$.

The remainder of this paper is organized as follows. In Section 2, we present the sequential and parallel Karp–Sipser algorithm and analyze the communication requirements of the parallel algorithm. In Section 3, we describe our experimental methodology, test set details, and our results. We conclude in Section 4.

2 Matching Algorithms

2.1 The Sequential Karp–Sipser Algorithm

Algorithm 1. Sequential Karp–Sipser (A)

```
$M \leftarrow \emptyset$
while $A \neq \emptyset$ do
    if $A$ has singleton rows then
        Pick a singleton row $i$ uniformly at random
        Let $a_{ij}$ be the nonzero entry in row $i$
    else
        Pick a nonzero entry $a_{ij}$ uniformly at random
    $M \leftarrow M \cup \{(i,j)\}$
    $A \leftarrow A \setminus (\{a_{i*}\} \cup \{a_{*i}\} \cup \{a_{j*}\} \cup \{a_{*j}\})$
return $M$
```

The Karp–Sipser algorithm [9] is a simple greedy algorithm for maximum cardinality graph matching. We will express it in terms of its adjacency matrix formulation. The idea of the algorithm is as follows. Let A be a symmetric matrix and M the set of matches. If the current matrix A has singleton rows, then the algorithm randomly chooses one such row i and adds (i, j) to the matching M, where a_{ij} is the unique nonzero entry in row i, and removes all the entries from rowcols i and j, and then continues. If the current matrix has more than one entry in each row, hence has no singleton rows, then it picks a random entry a_{ij}, adds (i, j) to the matching and deletes all the entries from rowcols i and j, and then continues. The algorithm stops when A has become empty. Algorithm 1 gives the formal description of the sequential Karp–Sipser algorithm. Note that while executing the algorithm, the deletion of rowcols generates new singleton rows.

There are two phases in the execution of the Karp–Sipser algorithm. The first phase starts at the beginning of the whole algorithm and ends when the current matrix has more than one entry in each row. Phase two is the remainder of the algorithm. We note that if M_1 is the set of entries chosen in phase one, then there still exists some maximum cardinality matching that contains M_1, see [1, Fact 1]. Thus the algorithm may have reduced the number of optimal solutions that it can find, but there still is at least one. Furthermore, it has been shown that almost all the remaining rows are matched by the Karp–Sipser algorithm in the special case where A is a random matrix [1, 5].

2.2 The Parallel Karp–Sipser Algorithm

In the remainder, we present our parallel implementation of the Karp–Sipser algorithm. As stated, the algorithm starts with the non-zero entries distributed among the processors and for each row i there is one dedicated owner of that row. Each processor then operates in synchronized rounds where it first performs a local version of the sequential algorithm followed by communication. Here, the match for a singleton row is performed at the processor that has the only (remaining) entry of that row.

In the sequential part, a processor P_s will try to match a predefined number, TpR, of its remaining unmatched rows. Priority is given to singleton rows but if P_s runs out of them before having performed TpR matching attempts, it will try to match some of its remaining rows with random neighbors. This is continued until TpR matching attempts have been reached or until P_s has run out of available rows. In our program texts, P_s will always denote the current processor which will execute the statements of the text.

To see how the algorithm differs from the sequential one, consider when P_s wants to match row i (which it owns) with row j. If P_s also owns row j it can immediately perform the match, but if another processor P_z owns it, then P_s must send a matching request to P_z. Depending on the outcome of this request the match will succeed or fail. Note that the only reason why a matching request could fail is if there were multiple requests to match with the same row in the same or the previous round. For termination, the algorithm relies on some random requests succeeding, which works well in practice. We could also have implemented a stricter mechanism to guarantee termination (e.g. by only requesting matches with higher numbered rows).

In addition to attempting to match its own rows, a processor must also process and answer incoming requests following the communication stage. The overall structure is outlined in Algorithm 2. In the algorithm, Q_s is a queue containing all singleton rows on processor P_s, while *StpR* and *RpR* denote the number of attempts per round to perform singleton and random matches.

Algorithm 2. PARALLEL KARP–SIPSER ()

```
while $A \neq \emptyset$ do
    PROCESS-MESSAGES()
    $StpR \leftarrow 0, RdpR \leftarrow 0$
    while $StpR + RdpR < TpR$ and $A \neq \emptyset$ do
        if $Q_s \neq \emptyset$ then
            PICK-SINGLETON-ROW()
        else
            PICK-RANDOM-ROW()
    BSP-SYNC()
```

2.2.1 The Different Message Types

Our algorithm relies on different types of messages to exchange information between the processors. The different types are summarized in Table 1 and explained in this subsection.

Table 1: Summary of message types used.

Type	Call	Meaning
Singleton request	$smr(i, j, P_z)$	Matches singleton row i to j
Random request	$rmr(i, j, P_z)$	Matches random row i to j
Confirmation	$cf(i, P_z)$	Confirms success of matching i
Unavailability	$u(i)$	Removes all nonzeros in rowcol i
Handover	$h(i)$	Hands over row i to a nonowner
Give-up	$g(i, P_z)$	Removes P_z from $nonOwners(i)$
Criticality	$ct(i, P_z))$	Local count of row i became 1

The first type of message is a *singleton match request*. Suppose processor P_s wants a singleton row i to match with row j, but $P(j) \neq P_s$. Therefore, P_s must send a message to $P(j)$ requesting to match j with i. We use *smr* to denote such a match request. Since $P(j)$ could receive several match requests from several processors and $P(j)$ can match j only with one i, $P(j)$ sends back reply messages, called *confirmation message* (denoted by cf), to update the requesters about the success of the match request. (If a requester does not receive a confirmation back within two rounds, this means that the request has failed.) The third type of message is the *unavailability message* (denoted by u). When a row i is matched with a row j, we need to remove all the entries from row i, column i, row j, and column j. Since we have only one adjacency list called rowcol i for each row i and column i, to remove all entries from both of them, it is sufficient to remove the entries from rowcol i. Therefore, we remove rowcol i and j. We first remove the entries in rowcol i and j from P_s, and then send unavailability messages to the other $q_i - 1$ and $q_j - 1$ processors holding row i and row j, respectively, to remove the entries in rowcol i and j from them. There is another type of match request, called *random match request* (denoted by rmr), used to match a row i with row j, given that $P(i) = P_s$. This type of message is only initiated when P_s can perform more work in the current round but $Q_s = \emptyset$, where Q_s denotes the queue of singleton rows in P_s.

We now discuss the remaining three types of messages. Consider a situation where i is a singleton row and the local count for row i in $P(i)$ is 0. Then, $P(i)$ must send a message to $P'(i)$, the only nonowner of i, to insert i into its singleton queue. We call this message a *handover message*, denoted by h. The next type of message is called *give-up message*, denoted by g, which is always sent from a nonowner, $P'(i)$ of i, to $P(i)$ to remove $P'(i)$ from $nonOwners(i)$. Processor $P'(i)$ sends such a message when its local count for row i reduces to 0. The last type of message, *criticality message* (denoted by ct), is sent from a nonowner $P'(i)$ to $P(i)$. We use this message to update $P(i)$ that the local count for row i in $P'(i)$ has been reduced to 1. This message enables the owner $P(i)$ to verify whether i has become a singleton row. The criticality message(s) for row i together with the knowledge of nz_i enable the owner of row i at the earliest possible moment to detect that a locally empty row has become singleton and thus to insert it into the appropriate queue on the only nonempty processor by using a handover message. We could have decided not to use criticality messages. Then, we would need a mechanism for transferring ownership, which is more complicated and would involve extra communication.

2.2.2 The Functions

Process-Messages **function**: This function processes all the incoming messages. We do the following, based on the message type. For each singleton match request $smr(i, j, P_z)$ received from processor P_z, we call Match-Rowcol($i, j, P_z, singleton$) to match i with j. For each unavailability message $u(i)$, we call Remove-RowCol(i) to remove i from P_s. For each handover message $h(i)$, we check whether i is singleton. If so, we push i into the singleton queue, Q_s. For a criticality message $ct(i, P_z)$, we reduce the nonzero count of row i, nz_i, by $lc_{i,z} - 1$. We also check whether this reduction makes i a singleton. If so, we send a handover message $h(i)$ to P_z to push i into its singleton queue. For each confirmation message $cf(i, P_z)$, we call Confirm(i, P_z) to remove row i. For each random match request $rmr(i, j, P_z)$, we call Match-Rowcol($i, j, P_z, random$) to match j with i. For each give-up message $g(i, P_z)$, we remove P_z from $nonOwners(i)$.

Since processor P_z sends the give-up message only when it removes its last local entry in row i, we reduce the nonzero count of row i, nz_i, by 1 and if relevant, insert row i into the singleton queue. We do this by calling Decrement(i).

In our implementation, all messages types have the same priority, and they are processed in the order they were entered into the receive buffer of the BSP system. It is possible, however, to sort them by type first, e.g., to give preference to singleton match requests over random match requests.

Pick-Singleton-Row **function** (given by Algorithm 3): The goal here is to pick and match a singleton row. The function first pops a row i from Q_s and then verifies whether i is still a singleton row by checking if $lc_{i,s} = 1$. (This check is necessary because unavailable rows are not removed from queues.) If not, it continues to the next singleton row in Q_s. If the answer is yes, it does as follows. Let a_{ij} be the unique entry in row i. Although j is the only option for i to match with, j could have several such singleton candidates and it can match only with one of them. Now, irrespective of which singleton row it is matching with, all entries from row j must be removed, because j will match with this i or one of the other candidate singleton rows, which eventually leads to the removal of rowcols i and j. So we can safely remove all entries from row j in P_s by calling Remove-RowCol(j). We also remove the only entry a_{ij} in row i, by setting $lc_{i,s} \leftarrow 0$. The next step is to check where the owner of j, $P(j)$ is. If $P(j) = P_s$, we call Match-Rowcol($i, j, P_s, singleton$) immediately to match j with i. Otherwise, we send a singleton match request $smr(i, j, P_s)$ to $P(j)$. The parameters P_s and *singleton* of Match-Rowcol mean that P_s has invoked the function and the matching request is of singleton type.

Algorithm 3. Pick-Singleton-Row()

```
i ← $Q_s.pop()$
if $lc_{i,s} = 1$ then
    $lc_{i,s} \leftarrow 0$
    Let $a_{ij}$ be the entry in row $i$
    Remove-RowCol($j$)
    if $P(j) = P_s$ then
        Match-Rowcol($i, j, P_s, singleton$)
    else
        send a singleton match request $smr(i, j, P_s)$ to $P(j)$
    $StpR \leftarrow StpR + 1$
```

Pick-Random-Row **function**: This is similar to Pick-Singleton-Row except that it picks a random row i owned by this processor to match with a random neighbor j. If there are multiple choices for j then priority is first given to local neighbors. If no local neighbor exists, a random match request is sent to $P(j)$.

Algorithm 4. Match-Rowcol($i, j, P_z, type$)

```
if $isMatched(j) = false$ then
    $m(j) \leftarrow i, isMatched(j) \leftarrow true, nz_j \leftarrow 0$
    if $type = random$ then
        if $P_z = P_s$ then
            Confirm($i, P_s$)
        else
            Remove-RowCol($i$)
            send a confirmation message $cf(i, P_s)$ to $P_z$
    if $P_z \neq P_s$ then
        Remove-RowCol($j$)
    send unavailability $u(j)$ to each $P'(j) \in nonOwners(j)$
```

Match-Rowcol **function** (given by Algorithm 4): The goal here is to match row j with row i if possible and take necessary actions if the matching is successful. We first verify whether j has already been matched by checking $isMatched(j)$. The next step is to check the *type* of the matching. If the type is singleton, we do not need to give any confirmation back to the owner $P(i) = P_z$, because its only remaining job was to remove the unique entry a_{ij} in row i, which has already been done. If the type is random, we need to send a confirmation back to $P(i)$ to let it remove row i. If $P_z = P_s$, we call Confirm(i, P_s) to remove row i from P_z and *nonOwners*(i). If $P_z \neq P_s$, we first remove row i from P_s and then send a confirmation message $cf(i, P_s)$. The next step is to remove rowcol j from $P_s = P(j)$ and *nonOwners*(j). Note that j has been removed from processor P_z in case of a singleton match request, and it will be removed following the confirmation in case of a random match request. If $P_z \neq P_s$, we remove rowcol j from P_s locally. We then send unavailability messages $u(j)$ to all the other $q_j - 1$ or $q_j - 2$ nonowners. We set $m(j) = i$, as this can be done locally by $P(j)$, but we do not set $m(i) = j$ as this would require communication with $P(i)$, which would be unnecessary since rowcol i will be removed immediately afterwards and hence cannot be matched anymore. No redundant matching information is thus communicated or stored.

Algorithm 5. Remove-RowCol(i)

```
while $lc_{i,s} > 0$ do
    Let $a_{ij}$ be the last entry in rowcol $i$
    swap $a_{ji}$ with the last entry in rowcol $j$
    $lc_{j,s} \leftarrow lc_{j,s} - 1$
    if $P(j) = P_s$ then
        Decrement($j$)
    else if $lc_{j,s} = 1$ then
        send a criticality message $ct(j, P_s)$ to $P(j)$
    else if $lc_{j,s} = 0$ then
        send a give-up message $g(j, P_s)$ to $P(j)$
    $lc_{i,s} \leftarrow lc_{i,s} - 1$
```

Remove-RowCol **function** (given by Algorithm 5): This function removes all entries from rowcol i in P_s. At every iteration of the while-loop, it picks the last entry a_{ij} from the adjacency list of row i. We remove a_{ij} from the adjacency list of rowcol i by reducing the local count $lc_{i,s}$ by 1. Since the matrix is symmetric, we also have to remove a_{ji} from rowcol j. We do this by swapping a_{ji} with the last entry of the adjacency list of rowcol j and reducing the local count $lc_{j,s}$ by 1. The swap and reduction operations make the removal efficient. Since j has not been matched yet, we consider whether the removal of a_{ji} creates any of the following three cases. The first case is where P_s owns j, so that we can safely reduce the nonzero count of row j by 1 and insert j into the singleton queue if possible. We do this by calling Decrement(j). The second case is where the removal of a_{ji} reduces the local count $lc_{j,s}$ to 1, so that we have to send a criticality message $ct(j, P_s)$ to $P(j)$ to reduce the nonzero count nz_j by $lc_{j,s} - 1$, where $lc_{j,s}1$ is the initial local count. The third case is where $lc_{j,s} = 0$, so that P_s does not have any entry in row j anymore and we can send a give-up message $g(j, P_s)$ to $P(j)$ to remove P_s from $nonOwners(j)$.

Confirm **function**: The goal here is to remove all entries in row i from $P(i)$ and $nonOwners(i)$. We remove row i from $P(i)$ by calling Remove-RowCol(i) and from the nonowners by sending unavailability messages $u(i)$ to all of them, except to a processor P_z that previously sent a confirmation message to $P(i)$ causing this function to be called.

Decrement **function**: This function first decrements the nonzero count nz_i of row i. It then checks whether this turns row i into a singleton row. If so, it looks where the last remaining entry a_{ij} of row i is. If a_{ij} is local, that is, $lc_{i,s} = 1$, we push i into the singleton queue Q_s. Otherwise, we send a handover message $h(i)$ to the only nonowner of i, $P'(i)$, asking $P'(i)$ to push i into its singleton queue.

2.3 Communication Requirements

Following the BSP model [3] for our parallel matching algorithm, we separate computation and communication into distinct, rather than intermingled, stages. The parallelism in the computation is obtained from the assumption that each processor will have a large number of local matches to perform between the communication supersteps. This allows us to analyse the computation and communication requirements separately. The computation part will be studied experimentally in the next section. For the communication part, we can obtain theoretical bounds on the total communication volume of the algorithm, as follows.

We analyse the communication requirements by considering a row i, with q_i processors. We assume that $q_i \geq 1$, because we can remove empty rows and columns. Let j be the requested matching partner of i. We distinguish between requests that succeed and those that fail. We will examine what the current processor P_s needs to communicate for row i. We count each message as one data word.

First, consider the case where i is a singleton row, $i \in Q_s$, see Algorithm 3.

It does not matter here whether or not $P_s = P(i)$.

- Case $P_s \neq P(j)$: P_s sends one message (a matching request) to $P(j)$. If i succeeds to match with j, then $P(j)$ sends $q_j - 2$ messages asking for removal of rowcol j to the nonowners of j, except P_s which initiated the matching request and therefore already removed rowcol j. If i fails to match with j, then $P(j)$ does not send any message at all. The total number of messages is $q_j - 1$ for success and 1 for failure.

- Case $P_s = P(j)$: as the previous case, but no matching request needs to be sent, and the number of removal requests in case of success is $q_j - 1$. The total number of messages is $q_j - 1$ for success and 0 for failure.

Therefore, for each singleton row i, the communication volume is at most $q_j - 1$. A similar analysis yields that for each randomly picked row, the volume is at most $q_i + q_j - 2$.

For the other three types of messages summarized in Table 1, row i incurs at most one handover message, and $q_i - 1$ give-up messages during the whole algorithm. But if $P(i)$ sends a handover message to $P'(i)$, then $P'(i)$ will never send a give-up message to $P(i)$, and vice versa. Therefore for each row i we get at most $q_i - 1$ give-up and handover messages, and $q_i - 1$ critical messages, with a total upper bound of $2q_i - 2$.

We can now add all the communication bounds. Let s be the number of matchings that involve at least one singleton row. Since the matrix A has n rows, the number of matched rows picked randomly is at most $\frac{n-2s}{2} = \frac{n}{2} - s$. Without loss of generality we renumber the rows, so that the matched singleton rows come first, and the matched random rows second, so they are in the range $1 \leq i \leq n/2$, and we also take care that their matches are in the second half, $n/2 + 1 \leq m(i) \leq n$. Assume for a moment that all match requests succeed. An upper bound for the total communication volume is then

$$\begin{aligned}
&\mathit{Vol}(\mathit{Matching}) \\
&\leq \sum_{i=1}^{n/2} q_{m(i)} + \sum_{i=s+1}^{n/2} q_i + 2\sum_{i=1}^{n} q_i - 3n + s \\
&\leq \sum_{i=1}^{n/2} q_{m(i)} + \sum_{i=1}^{n/2} q_i + 2\sum_{i=1}^{n} q_i - 3n \\
&= 3\sum_{i=1}^{n} (q_i - 1) = \frac{3}{2} \cdot \mathit{Vol}(\mathit{SpMV}).
\end{aligned}$$

Here, we express the upper bound in terms of sparse matrix–vector multiplication, which has a volume of $\mathit{Vol}(\mathit{SpMV}) = 2\sum_{i=1}^{n} (q_i - 1)$ for a symmetrically partitioned matrix. This is useful because the SpMV kernel is important and many partitioning algorithms and software packages exist that minimize its volume.

Now we drop the non-failure assumption. For singleton rows, at most one data word is sent and the row is removed and remains unmatched. The upper bound then still holds. For randomly picked rows, the situation is more complicated. If the request fails and $P(i) \neq P(j)$, this incurs one message. In principle, the number of such failures is unbounded, since randomly picked rows can be tried again, but in practice the volume will be limited as preference is given to local matches (not causing communication in case of failure) and the penalty in the non-local case is only one communication. This will add a number R of failed random requests to the upper bound.

A lower bound on the communication can be obtained as follows. Assume the best case, where all matches are local. For each row i, we need $q_i - 1$ messages to remove it. This leads to

$$\mathit{Vol}(\mathit{Matching}) \geq \sum_{i=1}^{n} (q_i - 1) = \frac{1}{2} \cdot \mathit{Vol}(\mathit{SpMV})$$

3 Experimental Results

3.1 Experimental Setup

We performed experiments on *Huygens*, an IBM pSeries 575 supercomputer at SARA in Amsterdam, consisting of 104 nodes, each with 16 processors and 128 GByte of memory. Each processor is an IBM Power6 dual-core 4.7 GHz processor where each core has 128 kByte of L1 cache and 4 MByte of L2 cache. Each processor has 32 MByte of L3 cache. The machine is running Linux (kernel version 2.6.27.45-0.1.2-ppc64). All algorithms were implemented in C++ using the BSPonMPI library (version 0.3) [17] and compiled with the IBM XL C/C++ compiler (version 10.01.0000.0002) using the -O3 optimization level.

We obtained the BSP parameters of the system by BSP benchmarking [3] for a given number of processors p, as shown in Table 2. These parameters are r, the single-processor computing rate in Gflop/s, g, the time taken by one processor to send or receive one data word, and l, the time taken to synchronize all processors.

We use four test sets of matrices. Test sets 1 and 2 consist of 10 real-world symmetric matrices and four real-world unsymmetric square matrices, respectively, of varying sizes drawn from different application areas such as medical science, structural engineering, civil engineering, circuit simulation, electrical engineering, DNA electrophoresis, information retrieval, and the automotive industry [6, 11]. Test set 3 includes three synthetic small-world matrices and test set 4 contains

Table 2: Benchmarked BSP parameters for the IBM pSeries 575.

p	r	(flop)		(μs)	
	(Gflop/s)	g	l	g	l
1	1.343	355	3,926	0.26	2.92
2	1.336	358	15,681	0.27	11.73
4	1.338	384	27,543	0.29	20.58
8	1.340	384	56,875	0.29	42.44
16	1.334	366	124,430	0.27	93.30
32	1.329	417	268,559	0.31	202.10
64	1.339	408	717,400	0.30	535.74

Table 3: Structural properties of the input matrices.

	n	nz	nz/n			n	nz	nz/n	
			avg	max				avg	max
rw1	999,999	3,995,992	3	4	rw11	281,903	3,985,272	14	38,625
rw2	1,585,478	6,075,348	3	5	rw12	16,783	9,306,644	554	14,671
rw3	52,804	10,561,406	200	2,702	rw13	683,446	13,269,352	19	83,470
rw4	2,063,494	12,964,640	6	95	rw14	343,791	26,493,322	77	434
rw5	63,838	14,085,020	220	3,422	sw1	50,000	14,112,206	282	5,096
rw6	504,855	17,084,020	33	39	sw2	75,000	24,466,808	326	6,273
rw7	503,712	36,312,630	72	842	sw3	100,000	33,727,170	337	7,989
rw8	952,203	45,570,272	47	76	er1	100,000	3,319,658	33	59
rw9	1,508,065	51,164,260	33	34	er2	150,000	6,753,302	45	76
rw10	914,898	54,553,524	59	80	er3	200,000	12,008,022	60	100

three synthetic Erdös-Rényi style random square matrices generated by the GT-Graph package [2]. For convenience, we label the test sets rw, sw, and er, for real-world, small-world, and Erdös-Rényi, respectively. The matrices from test sets 2–4 were made symmetric by adding A and A^T. All diagonal entries were removed from the matrices.

The structural properties of the test matrices are given in Table 3. The columns are the labels, number of rows, number of nonzeros, average and maximum number of nonzeros per row. The names of the matrices are given in Table 4. To obtain the runtime of an algorithm for a given matrix, we execute the algorithms three times and then take the minimum time, based on the assumption that this timing suffers the least from interference by other use of the hardware resources. In all three runs, the actual computations and hence the quality of the matching are the same (we start with the same random number seeds), so that timing differences between the runs are not due to different amounts of work performed.

3.2 Scalability Experiments

To check how well the edge partitioning approach works, we first compare it with a vertex partitioning, where we can use the same Mondriaan framework. In the

Table 4: Communication volume in 1000 words for $p = 32$.

	SpMV		*Matching*			*SpMV*		*Matching*	
Name	*1D*	*2D*	*1D*	*2D*	*Name*	*1D*	*2D*	*1D*	*2D*
rw1 (ecology2)	53	51	60	55	rw11 (Stanford)	340	141	479	234
rw2 (G3_circuit)	81	65	92	73	rw12 (gupta3)	710	44	1,305	61
rw3 (crankseg_1)	78	78	155	152	rw13 (St_Berk.)	716	448	1,152	812
rw4 (kkt_power)	118	120	106	107	rw14 (F1)	139	130	148	139
rw5 (crankseg_2)	92	90	181	171	sw1	1,007	417	2,111	303
rw6 (af_shell8)	51	47	85	65	sw2	1,957	829	3,999	563
rw7 (inline_1)	104	105	115	118	sw3	2,017	832	4,255	528
rw8 (ldoor)	131	128	140	148	er1	1,856	1,133	1,788	1,157
rw9 (af_shell10)	113	105	169	150	er2	3,451	1,841	3,721	1,635
rw10 (boneS10)	150	145	228	189	er3	5,476	2,569	6,350	1,990

vertex partitioning, we simply impose the extra constraint that all nonzeros in a row up to the matrix diagonal are assigned to the same processor. This way, we can view vertex partitioning as a special case of edge partitioning. Table 4 presents the communication volumes of sparse matrix–vector multiplication and matching, both for a vertex (1D) partitioning and an edge (2D) partitioning on 32 processors. The SpMV volume is a direct outcome of the partitioning by the Mondriaan package [20] (version 2.01) in symmetric mode, where the maximum number of edges per processor is not allowed to exceed the average by more than 3%. The processor with most nonzeros in row i was chosen as the owner $P(i)$, because it is more likely to possess the last remaining nonzero of the row after the other nonzeros have been removed, thus saving a handover message. The volume for matching is the volume measured by counters in the program, which register the number of (integer) data words sent.

Table 4 shows that on 32 processors, the volume for the matching is in a range from 0.63 to 2.11 times the SpMV volume. We also observed a range between 0.63 and 2.18 for 2, 4, 8, 16, 32, and 64 processors. This shows that partitioning for the SpMV objective is also a good optimizer for matching, and possibly for other graph problems as well. The table shows a savings in communication volume of a factor of 2 for small-world and Erdös-Rényi matrices when moving from 1D to 2D, and even larger savings for the real-world matrices from test set 2. Note the large 16-fold decrease for the linear programming matrix rw12 (gupta3). For the symmetric real-world matrices (test set 1), only some modest gains can be observed, but also a few cases with a small loss.

Table 5 gives the speedup of our parallel Karp–Sipser implementation on 32 processor cores compared to the time of our sequential implementation. We examine the performance as a function of the input parameter TpR, which is the total number of rows processed in a round and which represents the chosen granularity of the computation. Choosing a small value of TpR leads to many rounds in the whole algorithm, and hence many supersteps and synchronizations. For $p = 32$, one synchronization costs about $l = 270,000$ flop time units, see Table 2,

Table 5: Speedup as a function of TpR for $p = 32$. Boldface denotes the highest speedup obtained.

$TpR =$	100	200	400	800	1600		100	200	400	800	1600
rw1	0.67	**0.74**	0.62	0.40	0.24	rw11	4.25	5.32	6.15	6.17	**6.45**
rw2	0.66	**0.72**	0.59	0.38	0.20	rw12	25.36	18.99	**30.55**	29.55	30.35
rw3	12.65	13.07	**15.13**	14.53	14.42	rw13	1.18	1.59	1.83	**1.85**	1.73
rw4	**1.55**	1.30	0.72	0.31	0.17	rw14	13.15	16.67	19.54	21.63	**24.23**
rw5	14.11	16.62	19.69	**21.09**	19.99	sw1	29.49	33.38	**34.63**	30.58	30.82
rw6	6.26	9.29	12.92	**14.03**	13.82	sw2	27.87	31.16	33.85	**33.91**	33.75
rw7	9.19	11.17	12.09	12.85	**12.88**	sw3	33.35	40.83	42.18	**44.64**	42.43
rw8	6.93	8.45	9.22	**9.25**	8.83	er1	5.20	6.02	7.64	8.60	**9.51**
rw9	6.44	9.66	12.19	**13.08**	11.50	er2	7.15	9.60	11.00	12.71	**13.63**
rw10	7.07	8.41	**8.82**	7.97	6.60	er3	14.31	15.97	18.14	19.72	**21.55**

Table 6: Matching quality (in %) for the experiments of Table 5. Boldface denotes the highest quality obtained.

$TpR =$	100	200	400	800	1600		100	200	400	800	1600
rw1	**98.15**	98.14	98.13	98.08	98.12	rw11	**71.75**	71.61	71.48	71.32	71.11
rw2	**96.71**	96.69	96.61	96.52	96.45	rw12	**98.31**	98.00	97.35	97.35	97.35
rw3	**99.21**	99.15	99.13	99.16	99.19	rw13	**66.19**	66.15	66.09	65.99	65.87
rw4	88.55	**88.58**	**88.58**	88.57	88.57	rw14	**99.54**	99.52	99.53	99.51	99.49
rw5	**99.26**	99.24	99.24	99.20	99.18	sw1	**79.81**	78.07	77.06	75.66	75.59
rw6	**99.93**	**99.93**	99.92	**99.93**	**99.93**	sw2	**90.74**	88.87	86.25	84.09	81.89
rw7	**99.56**	99.55	99.55	99.54	99.53	sw3	**81.87**	80.13	78.47	77.29	76.01
rw8	**98.58**	**98.58**	**98.58**	**98.58**	98.57	er1	**97.50**	93.45	85.67	78.69	74.13
rw9	**99.94**	**99.94**	**99.94**	**99.94**	**99.94**	er2	**98.43**	95.63	89.12	82.54	76.07
rw10	**99.58**	99.56	99.55	99.55	99.55	er3	**95.98**	93.14	88.94	83.42	77.59

so the number of operations carried out per processor in a round should at least be this number. As a rough estimate, for the matrix er3, this means handling about 1500 rows of 60 nonzeros each, with (an estimated) three operations per nonzero, where for simplicity we assume that every row is completely assigned to one processor; in reality, some rows are partitioned. The advantage of a small TpR is better load balance: when a processor runs out of work this will be detected earlier, and communications are performed more frequently, thus enabling processors to carry out work that otherwise would have to wait until later.

Finding the right value of TpR is important to get good speedups. Fortunately, the parameter is not very sensitive, and a whole range of values gives the highest obtainable speedup; e.g. for er3, this is the range 400–1600. The overall highest speedup obtained (44.64 for sw3) is superlinear, which must be due to beneficial cache-effects or to the fact that the sequential and parallel algorithms do not perform exactly the same amount of work. (The parallel algorithm may be forced to pick random rows more often than the sequential algorithm thus performing less work and delivering lower quality.) Other problem instances may have benefited from these effects as well.

The choice of TpR also influences the quality of the solution (defined as the ratio between the number of matched rows and the total number of rows) for the matrices from test sets 3 and 4, see Table 6. Here, the quality decreases with increasing TpR. For these matrices, which have high communication volumes due to their random nature, few singleton rows can be processed in a round, forcing the processing of random rows in many cases.

Tables 7 and 8 present the speedups for the vertex and edge partitioning approaches. In all cases, the value of TpR was set at an optimal value based on an empirical parameter search, choosing the value among 100, 200, 400, 800, and 1600 that gave the highest speedup. In general, it can be observed that vertex partitioning and edge partitioning do not differ much in time and quality for test set 1, but that edge partitioning is much faster for test sets 2–4. The matrix rw12 shows a much larger maximum speedup (30.55) for edge partitioning than for vertex partitioning (5.78), and also a better quality. This holds for most cases, but there are exceptions, cf. sw1–sw3 for $p = 64$. The higher speedups for edge partitioning are primarily caused by the lower communication volume, see Table 4 for $p = 32$, but other factors play a role as well. For instance, the smallest problem rw1 shows no speedup at all, which is most likely caused by severe load imbalance and a relatively large synchronization overhead.

4 Conclusion

In this work, we have demonstrated how a graph matching algorithm, the Karp-Sipser algorithm, can be parallelized efficiently by viewing it as a sparse matrix algorithm, and by making use of sparse matrix partitioning methodology. A number of conclusions can be drawn:

- Edge-based partitioning gives for certain types of graphs, such as small-world graphs, a large improvement compared to vertex-based partitioning. For other types of matrices, a more modest improvement is obtained. In the remaining few cases, the differences are small.

- Improvements obtained by better partitioning lead to better locality, thus reducing the amount of communication required and hence making the parallel algorithm run faster. They also enable more computations to be done locally within a superstep, keeping work queues filled longer and hence improving the matching quality, i.e., the percentage of matched vertices.

- We have established a theoretical relation between the communication volume of parallel graph matching by the Karp–Sipser algorithm and sparse matrix-vector multiplication,

$$\frac{1}{2} \cdot \mathit{Vol}(\mathit{SpMV}) \leq \mathit{Vol}(\mathit{Matching}) \leq \frac{3}{2} \cdot \mathit{Vol}(\mathit{SpMV}) + R$$

Table 7: Speedup (Su) and matching quality in % (Ql) using vertex (1D) partitioning.

	Seq	$p = 2$		$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 64$	
	Ql	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*
rw1	100.00	0.17	99.84	0.15	98.02	0.29	98.11	0.45	97.88	0.70	97.92	**0.84**	98.09
rw2	99.93	0.12	96.95	0.18	96.62	0.28	96.51	0.44	96.45	0.71	96.40	**0.86**	96.18
rw3	99.59	1.60	99.57	3.49	99.48	5.82	99.46	12.62	99.39	**20.35**	99.16	13.01	99.42
rw4	91.54	0.49	88.09	0.62	88.44	0.79	88.35	1.37	88.48	**1.47**	88.45	1.29	88.42
rw5	99.60	1.78	99.61	3.68	99.56	6.92	99.50	13.56	99.36	**22.90**	99.07	17.58	99.36
rw6	99.99	1.58	99.97	2.80	99.97	4.80	99.96	8.11	99.94	13.28	99.92	**17.29**	99.90
rw7	99.62	1.32	99.58	2.00	99.56	2.98	99.58	6.05	99.57	14.07	99.52	**28.95**	99.48
rw8	98.53	1.30	98.72	2.11	98.74	3.24	98.73	5.38	98.72	9.39	98.73	**13.99**	98.73
rw9	99.99	1.71	99.99	2.90	99.98	5.04	99.97	8.16	99.96	13.99	99.95	**19.73**	99.93
rw10	99.70	1.37	99.67	2.26	99.65	3.56	99.64	5.65	99.62	8.27	99.60	**11.34**	99.58
rw11	74.26	1.00	72.09	1.66	72.02	2.38	71.57	3.98	71.20	**5.46**	70.81	4.61	70.26
rw12	99.06	1.91	73.18	3.07	57.98	4.83	64.03	5.13	82.40	**5.78**	86.68	3.59	97.78
rw13	68.56	0.62	66.19	0.81	66.29	0.89	66.16	1.37	65.94	1.81	66.03	**1.92**	65.08
rw14	99.65	1.49	99.61	2.65	99.60	4.81	99.57	10.54	99.55	20.93	99.49	**33.30**	99.42
sw1	82.77	2.28	82.56	4.64	82.53	8.46	82.39	14.39	82.32	**17.33**	82.04	14.51	79.89
sw2	93.68	2.22	93.33	4.65	93.28	8.73	93.16	15.27	92.87	**21.43**	92.93	20.94	90.27
sw3	82.76	2.28	82.65	5.41	82.47	9.28	82.44	17.19	82.42	26.75	82.37	**27.97**	81.35
er1	99.99	1.32	98.87	1.66	98.66	1.82	97.81	2.35	86.87	3.27	63.63	**5.23**	46.66
er2	99.99	1.48	99.16	2.26	99.18	2.60	99.17	2.90	96.63	3.75	71.99	**5.31**	53.75
er3	100.00	1.47	99.33	2.48	99.39	2.88	99.33	3.38	95.60	3.99	89.41	**5.05**	60.85

where R represents the number of random match requests that failed during the algorithm. The range we encountered in practice for edge partitioning, is between 0.63 to 1.95 times $Vol(SpMV)$ for 2, 4, 8, 16, 32, and 64 processors.

- We have obtained good speedups for many matrices without compromising the quality of the matching. Up to 16 processors, the matching quality stays constant, see Table 8. After that, for some matrices it decreases as work queues become empty more quickly, thereby forcing random rows to be matched.

For future work, we see the present algorithm as a representative of a whole class for which an edge-based approach will be suitable and a relation with sparse matrix–vector multiplication can be established. We intend to generalize our approach in this direction.

Acknowledgement: This work has been performed under the HPC-Europa2 Transnational Access programme, in FP7, grant agreement nr. 228398.

References

[1] J. Aronson, A. Frieze, and B. G. Pittel, *Maximum matchings in sparse random graphs: Karp-Sipser revisited*, Random Structures & Algorithms, 12 (1998), pp. 111–177.

Table 8: Speedup (*Su*) and matching quality in % (*Ql*) using edge (2D) partitioning.

	Seq	$p = 2$		$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 64$	
	Ql	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*	*Su*	*Ql*
rw1	100.00	0.18	99.84	0.18	98.79	0.34	98.86	0.53	98.54	0.74	98.14	**0.92**	98.02
rw2	99.93	0.11	96.95	0.17	96.50	0.30	96.83	0.47	96.99	0.72	96.69	**0.83**	96.62
rw3	99.59	1.60	99.59	3.42	99.57	6.49	99.39	12.57	99.12	**15.13**	99.13	10.45	98.92
rw4	91.54	0.49	88.09	0.58	88.15	0.80	88.19	1.31	88.44	**1.55**	88.55	1.36	88.55
rw5	99.60	1.78	99.61	3.49	99.55	6.76	99.47	13.97	99.35	**21.09**	99.20	13.93	98.78
rw6	99.99	1.57	99.98	2.77	99.97	4.68	99.95	8.36	99.95	14.03	99.93	**15.13**	99.89
rw7	99.62	1.24	99.56	1.84	99.58	3.13	99.55	6.10	99.54	12.88	99.53	**25.75**	99.47
rw8	98.53	1.29	98.72	2.11	98.72	3.32	98.74	5.59	98.64	9.25	98.58	**14.01**	98.68
rw9	99.99	1.64	99.99	2.97	99.98	5.20	99.97	9.13	99.96	13.08	99.94	**17.97**	99.93
rw10	99.70	1.37	99.67	2.33	99.62	3.77	99.62	6.04	99.59	8.82	99.55	**11.70**	99.51
rw11	74.26	0.95	71.99	1.63	71.96	2.37	71.73	4.57	71.21	**6.45**	71.11	5.45	70.28
rw12	99.06	2.65	99.35	6.50	99.28	14.23	97.67	25.44	97.96	**30.55**	97.35	23.39	90.98
rw13	68.56	0.65	66.30	0.81	66.31	1.01	67.18	1.31	65.85	1.85	65.99	**2.57**	65.73
rw14	99.65	1.49	99.61	2.79	99.57	5.05	99.57	11.38	99.53	24.23	99.49	**37.74**	99.40
sw1	82.77	2.28	82.56	4.82	82.44	9.36	82.28	19.74	80.31	34.63	77.06	**41.55**	73.33
sw2	93.68	2.20	93.33	4.69	92.94	8.96	92.36	19.10	89.64	33.91	84.09	**55.39**	78.38
sw3	82.76	2.27	82.65	5.23	82.47	9.92	82.01	21.02	79.91	44.64	77.29	**72.16**	73.80
er1	99.99	1.34	98.87	2.27	98.65	4.35	97.43	7.13	83.71	9.51	74.13	**13.14**	58.46
er2	99.99	1.48	99.16	2.98	99.23	5.71	97.89	9.83	86.62	13.63	76.07	**21.12**	64.23
er3	100.00	1.48	99.39	3.10	99.39	5.66	99.15	11.89	92.63	21.55	77.59	**29.96**	67.43

[2] D. A. Bader and K. Madduri, *GTGraph: A synthetic graph generator suite.* http://www.cc.gatech.edu/˜kamesh/GTgraph, 2006.

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

[4] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. Boman, and Ümit V. Çatalyürek, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.

[5] P. Chebolu, A. Frieze, and P. Melsted, *Finding a maximum matching in a sparse random graph in $O(n)$ expected time*, in Automata, Languages and Programming, L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, eds., vol. 5125 of LNCS, Springer Verlag, 2008, pp. 161–172.

[6] T. A. Davis, *The university of florida sparse matrix collection*, IEEE Transactions on Circuits and Systems. To appear.

[7] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, 1986.

[8] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[9] R. M. Karp and M. Sipser, *Maximum matching in sparse random graphs*, Annual IEEE Symposium on Foundations of Computer Science, 0 (1981), pp. 364–375.

[10] G. Karypis and V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, 48 (1998), pp. 96–129.

[11] J. Koster, *Parasol matrices.* http://www.parallab.uib.no/projects/, 1999.

[12] J. Langguth, F. Manne, and P. Sanders, *Heuristic initialization for bipartite matching problems*, ACM Journal of Experimental Algorithmics, 15 (2010), pp. 1.3:1–1.3:22.

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing*, in proceedings of the 28th ACM symposium on Principles of distributed computing (PODC 2009), ACM, 2009, pp. 6–6.

[14] F. Manne and R. H. Bisseling, *A parallel approximation algorithm for the weighted maximum matching problem*, in proceedings of the Seventh International Conference on Parallel Processing and Applied Mathmatics (PPAM 2007), vol. 4967 of LNCS, Springer Verlag, 2007, pp. 708–717.

[15] R. H. Möhring and M. Müller-Hannemann, *Cardinality matching: Heuristic search for augmenting paths*, Tech. Rep. 439, Fachbereich Mathematik, Technische Universität Berlin, 1995.

[16] F. Pellegrini and J. Roman, *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in High-Performance Computing and Networking, H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, eds., vol. 1067 of LNCS, Springer Berlin/Heidelberg, 1996, pp. 493–498.

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

[18] Ümit V. Çatalyürek and C. Aykanat, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), 2001, pp. 609–625.

[19] L. G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, 33 (1990), pp. 103–111.

[20] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

IV

Parallel Algorithms for Bipartite Matching Problems on Distributed Memory Computers

Johannes Langguth Md. Mostofa Ali Patwary
Fredrik Manne*

Abstract

We present the first practical parallel algorithm for computing a maximum cardinality matching in a bipartite graph suitable for distributed memory computers.

The presented algorithm is based on the Push-Relabel algorithm which is known to be one of the fastest algorithm for the bipartite matching problem. Previous attempts at developing parallel implementations of it have focused on shared memory computers using a limited number of processors.

We first present a straightforward adaptation of these shared memory algorithms to distributed memory computers. However, this is not a viable approach as it requires too much communication. We then present an efficient algorithm by modifying the previous approach through a sequence of steps with the main goal being to reduce the amount of communication and to increase load balance. The first goal is achieved by changing the algorithm so that many push and relabel operations can be performed locally between communication rounds and also by selecting augmenting paths that cross processor boundaries infrequently. To achieve good load balance, we limit how fast global relabelings traverse the graph. Through a number of experiments on large instances we show the scalability of our algorithm using up to 128 processors.

Keywords: Bipartite graphs, parallel algorithms, matching.

1 Introduction

The bipartite cardinality matching problem is defined as follows:

Given an undirected, bipartite graph $G = (V_1, V_2, E), E \subseteq V_1 \times V_2$, find a maximum subset $M^* \subseteq E$ of pairwise nonadjacent edges. A set M^* is called a

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, {Johannes.Langguth,Mostofa.Patwary,fredrikm}@ii.uib.no

perfect matching iff $|V_1| = |V_2| = |M^*|$. Clearly, not all bipartite graphs have a perfect matching.

Bipartite matching is a classical topic in combinatorial optimization and has been studied for almost a century. It has many applications, but it is especially relevant for numerical computations due to the fact that it can be used to find zero-free diagonals for linear solvers [8, 20], making it an important problem in combinatorial scientific computing.

For a given $m \times n$ matrix A we define $G_A = (V_1, V_2, E)$ where $|V_1| = m$, $|V_2| = n$, and $E = \{\{i, j\} \in V_1 \times V_2 : a_{i,j} \neq 0\}$ as the graph derived from A. Assuming A belongs to a specified systems of linear equations, A is a square matrix having full structural rank [5] and G_A has a perfect matching. As any edge included in a matching on G_A corresponds to an entry of A being permuted to the main diagonal, a perfect matching corresponds to a *transversal* for A.

Many sophisticated sequential implementations solving this problem, such as MC21 [6, 7], are available, but for parallel linear solvers where A is typically distributed among several processors, it is desirable to use a matching algorithm that works directly on the distributed matrix to avoid the memory limitations of a single node and that scales reasonably well with the number of processors.

Bipartite matching is a special case of the maximum flow problem, for which sequential polynomial time algorithms have long been known [9], and many specialized algorithms for finding bipartite matchings have been proposed in the past [1, 13]. Among these, the Push-Relabel algorithm [11] by Goldberg and Tarjan has proven to be one of the fastest sequential algorithms, and it also exhibits a structure that makes it more amenable to parallelization than other matching algorithms. Parallelizations of the Push-Relabel algorithm for maximum flow on shared memory computers have been presented by Bader and Sachdeva [3] and by Anderson and Setubal [2]. The latter work has been adapted to bipartite matching and studied in [22]. Recently, in [21] an implementation of an auction based algorithm for finding a perfect matching of maximum weight on a distributed parallel computer was presented. This algorithm is used as a subroutine for the Pspike [18] hybrid linear solver.

In this paper we show how an efficient parallel algorithm suitable for distributed memory can be obtained from the parallel Push-Relabel algorithm suggested in [11]. To do so, we first adapt ideas for shared memory computation from [2] to the distributed memory model, thereby obtaining a simple parallel bipartite matching algorithm labeled Algorithm 1. It is presented in Section 4. Since analysis and preliminary experiments showed that the performance of Algorithm 1 was not competitive, detailed experimental results are not presented. Instead, in Section 5 we show how to modify Algorithm 1 in order to obtain the competitive Algorithm 2. The improvement is achieved by relaxing the requirement for labels to constitute a global lower bound on the distance to a free vertex, a central invariant of the Push-Relabel algorithm. This allows us to execute a large number of push and relabel operations that were previously locked in successive communication phases outside of their standard order, thereby creating large batches of operations that can be processed locally. To modify the communication to computation ratio, the size of these batches is adjusted depending on the number of available processors. This allows us to control, and thereby optimize load-balancing. In Section 5.4 we show that the algorithm remains correct under this relaxation.

Section 6 describes the experimental setup and Section 7 the experimental results for Algorithm 2. These results show that for reasonably large instances, the algorithm shows very good scaling behaviour, but we do not consistently obtain speedup compared to an efficient sequential implementation. However the advantage of the sequential algorithm is small for large matrices. Thus, to our best knowledge this paper describes the first efficient parallel algorithm for bipartite matching on distributed memory computers.

2 Preliminaries

In the following, V_1 is designated as the *left side* and V_2 as the *right side.* A vertex $v \in V_1$ is a *left vertex* and $u \in V_2$ is a *right vertex.* The matching is denoted by the set of matched vertices M. In addition, if $v \in M$, $M(v)$ is the vertex matched to v. An unmatched vertex $w \notin M$ is called a *free vertex* with $M(w) = \emptyset$. In general, we will refer to *free left vertices* as *active vertices.* If $M(v) = u$ and $M(u) = v$ then the edge $e = \{u, v\}$ is a *matched edge.* By the definition of a matching, any other edge incident on either u or v cannot be a matched edge, and is thus referred to as *unmatched.* A path P that alternates between matched and unmatched edges is called an *alternating path.* If both endpoints of P are free, it is also an *augmenting path* for M because switching all matched edges of P to unmatched and vice versa results in a matching of cardinality $|M| + 1$. For a vertex $v \in V_1$, let $\Gamma(v) \subseteq V_2$ be the neighborhood of v, i.e., the set of all vertices adjacent to v. For $u \in V_2$, define $\Gamma(u) \subseteq V_1$ analogously.

Edge-based partitioning

Both our algorithms assume that the input matrices are partitioned using a 2-D partitioner, such as Mondriaan [24] or Zoltan [23]. They allow a more fine-grained partitioning than 1-D, i.e., column or row-based partitioning and have proven useful for reducing communication [19]. In graph terms, 2-D matrix partitioning amounts to edge-based partitioning, which means that a single edge is owned by only one processor, but a vertex v may be shared among multiple processors. In this case, the processor on which v has maximum degree treats v as an *original vertex*, while all other processors that have an instance of v treat it as a *ghost vertex.* We will refer to such a split vertex, along with all its ghost instances as a *connector*, denoted as $C(v)$. For a connector that is split among

k vertices, $C_0(v)$ will refer to the original vertex in the connector, i.e., v, while $C_1(v), ..., C_{k-1}(v)$ refers to the $k-1$ ghost instances that are part of the connector. Communication during the algorithms is performed only between processors that share connectors. We will distinguish between *left connectors*, i.e., vertices v such that $C_0(v) \in V_1$ and *right connectors*, i.e., vertices u such that $C_0(u) \in V_2$. To simplify notation, for a vertex v that is not part of a connector, let $C_0(v) = v$. Such a vertex is designated as *local*. An edge $\{C_0(v), C_{i>0}(u)\}$ is called a *crossing edge*, while an edge $\{C_0(v), C_0(u)\}$ is called *local*.

3 The Push-Relabel Algorithm for Bipartite Matching

The Push-Relabel algorithm by Goldberg and Tarjan [11] was originally designed for the maximum flow problem. Since bipartite matching is a special case of maximum flow, it can be solved using the Push-Relabel algorithm. In fact, it is one of the fastest know algorithms for bipartite matching, as was shown in [4]. The algorithm works by defining a distance labeling $\psi : V_1 \cup V_2 \to \mathbb{N}$ which constitutes a lower bound on the length of an alternating path from a vertex v to the next free right vertex. Note that if v is a free left vertex, such a path is also an augmenting path. We initialize $\psi(v) = 0 \; \forall v \in V_1 \cup V_2$. Now, as long as there are free left vertices, we pick one of these v and search its neighbourhood for a vertex u with minimum $\psi(u)$. If $\psi(u) = 0$ then u must be unmatched, and matching it to v increases the size of M by one. Otherwise, let u be matched to w. In that case we match v and u, rendering w unmatched. Because now any augmenting path from a free right vertex to u must contain v, we update the distance labels by setting $\psi(v) = \psi(u) + 1$ and increasing $\psi(u)$ by 2. If $\psi(v) > 2n$, we instead mark v as unmatchable and cease to consider it any further.

In maximum flow terms, during this operation a unit of flow entering v from the source has been transferred from v to u. If u was unmatched, it now has one unit of incoming flow which is routed to the sink. Otherwise, u is left with one unit of excess flow which has to be pushed back to a left vertex from which it currently receives flow, i.e., v or w. Since the algorithm cannot progress by immediately undoing the push from v to u, flow is pushed back from u to w, making w active immediately. The entire operation is referred to as a double push if $M(u) \neq \emptyset$, otherwise it is called a single push. Since G is bipartite, this technique ensures that matched right vertices can never become active again because excess flow will immediately be transferred back due to the double push [4]. Therefore, implementation of the Push-Relabel algorithm for bipartite matching is significantly easier and more efficient than using the standard Push-Relabel algorithm for maximum flow.

In the bipartite matching context, a push from v to u means matching the edge $\{v, u\}$. If $M(u) \neq \emptyset$ was true prior to the push, for $M(u) = w$ we also unmatch the edge $\{w, u\}$. Thus, u is added to M while w is removed. We also set $M(v) = u$, $M(u) = v$, and $M(w) = \emptyset$ thereby making w active instead of v. Updating ψ ensures that the now free vertex w does not push to u again unless there is no better alternative.

The push operation is repeated until there are no further active vertices. It is easy to show that in this case M is a maximum matching. As observed in [11] pushes can be performed in arbitrary order, making the algorithm a prime candidate for parallelization. In the sequential case the order of pushes seems to influence performance strongly [4]. However, the algorithm can be initialized by starting with a matching M of high cardinality which can be obtained easily using various heuristics [16]. Doing so can level the difference between various orders of operations.

Although not necessary for the algorithm, performance is enhanced greatly by periodic updates of the distance labels. This is achieved by starting a breadth first search along alternating paths from each unmatched right vertex to set the labels ψ to the actual alternating path distance from the next free right vertex. This operation is called a *global relabeling*. After performing such a global relabeling, augmenting paths consisting of edges $\{u, w\}$ with $\psi(w) \geq \psi(u) + 1$ from the free right to all reachable free left vertices exist. These edges are called *admissible*, since they are correctly aligned with the distance labeling. Assuming ψ is a valid distance labeling, $\psi(w) > \psi(u) + 1$ cannot occur. In contrast to the original Push-Relabel algorithm for the general maximum flow problem, the behaviour of the above algorithm for bipartite matching does not depend on the labels of left vertices since a push can only reach them through a matched edge which will always be admissible.

Like other matching algorithms, the Push-Relabel algorithm with global relabeling searches for augmenting paths. However, it does not augment the matching immediately along paths so discovered. Instead, it uses the distance labels to simultaneously guide the “flow”, i.e., the unmatchedness of all free left vertices towards the unmatched right vertices, increasing the size of the matching when a free left vertex and an adjacent free right vertex are matched.

4 Adaptation of the Push-Relabel Algorithm for Distributed Memory

In [11] the framework for a synchronous parallel algorithm relying on repeated application of a pulse operation was given. Such a pulse operation consists of each active vertex v attempting to push to a neighbour u with minimum $\psi(u)$. Due to the distributed nature of the algorithm, this can result in multiple active vertices pushing to the same right vertex. If this happens, only one of these pushes is considered successful, and all other pushes are considered to have failed. Thus the active vertices that initiated the failed pushes remain active and must try to push again during the next pulse, possibly to a different vertex. Similar to the sequential Push-Relabel algorithm, the pulse operation is repeated until no active vertices remain.

From this framework, we derive a simple parallel distributed memory algorithm called Algorithm 1. In [11], following the PRAM model, one processor per vertex is available. In the distributed memory setting, we instead assume an edge based partitioning of G among p processors, where p is several orders of magnitude smaller than n.

As was suggested in [11], Algorithm 1 works in rounds. During each round, for each active vertex v, the algorithm queries for the lowest labeled neighbour u of v, attempts to match v with u and, if successful, updates labels and matching edges. Rounds are repeated until a maximum matching is found. We refer to the two parallel operations in this algorithm as Query and Pulse. In shared memory models such as PRAM, the procedure Query does not appear explicitly since data access is trivial.

Algorithm 1 Procedure QUERY

```
for each active vertex $v$ do
    Find local neighbour $u_0(v)$ of minimum $\psi(u_0)$ where $\omega(u_0) = \omega(v)$
    Send Request signal to each ghost vertex $C_{k>0}(v)$
Exchange Request messages with other processors
for each ghost vertex $C_{k>0}(v)$ receiving a Request signal do
    Find local neighbour $u_k(v)$ of minimum $\psi(u_k)$ where $\omega(u_k) = \omega(v)$
    Send Response signal $(u_k(v), \psi(u_k)(v))$ to $C_0(v)$
Exchange Response messages with other processors
for each active vertex $v$ do
    Set $u^*(v) = \operatorname{argmin}_{\{u_i|0\leq i\leq p\}} \psi(u_i(v))$
```

To find a neighbour of minimum label, every original left vertex in a left connector needs to query all its neighbours, including those on other processors. Only vertices in the same stage of a global relabeling can be considered as possible matching partners (See *Global Relabeling* below for more information on wave numbers ω). Queries to other processors are initiated by sending request signals. For efficient communication, procedure Query gathers all request signals that refer to ghost vertices and thus require communication. It then transfers the request signals using point to point messages. Thus, in every round all signals from processor p_i to p_j are bundled in one message for every pair of processors between which communication takes place. Throughout this paper, all communication will follow this paradigm. Conceptually, this resembles the BSP model (see [12] for further details), although the implementation is geared towards higher flexibility in superimposing communication and computation.

Requests are answered by returning a local neighbour of minimum label. All such responses are bundled and transferred in the same manner as the queries. After receiving the responses, each active vertex v selects a neighbour u with minimum $\psi(u)$ among all the responses received. Clearly, u is a minimum labeled neighbour of the entire connector $C(v)$. Vertex u is then designated as the *push target* u^* for v.

For a left connector $C(v)$, only the original vertex $C_0(v)$ carries a label $\psi(v)$. The ghost vertices $C_{k>0}(v)$ in a left connector do not carry any information except for a link to $C_0(v)$ and their respective adjacency lists. In a right connector $C(u)$, all vertices $C_{k\geq 0}(u)$ carry a label $\psi(C_{k\geq 0}(u))$ which is updated along with other information as described below. Thus, every vertex in a left connector can obtain the minimum label in its local neighborhood without further communication. Figure 1 illustrates this structure.

In procedure Pulse every active vertex v sends a match signal to its push target u^* with messages exchanged as described above. A push target u will always accept the first push received, sending a success signal to the source and, if necessary an unmatch signal to its previous matching partner. Following a successful push, the processor owning vertex v sets $\psi(v) = \psi(u^*)+1$, where $\psi(u^*)$ is the local value of the minimum neighbouring label determined in procedure Query. Furthermore, the owner of a push target u locally sets $\psi(u) \leftarrow \psi(u) + 2$ and, assuming u is part of a connector $C(u)$, transfers the updated $\psi(u)$ to all vertices $C_k(u)$ in the connector.

If further push requests for u^* arrive during the same round, they are denied and a reject signal is sent back to the source of the push. Again, all these signals are gathered so that at most one message is exchanged between each pair of processors.

In the next round, previously active vertices whose pushes were rejected, along with vertices that received unmatch signals, become active, while those that pushed successfully become inactive. If no active vertices remain on any processor, the algorithm terminates, returning a perfect matching. To terminate the algorithm in graphs without a perfect matching, we remove any left vertex v where $\psi(u) > 2n \; \forall u \in \Gamma(v)$, from the set of active vertices as there is no way to increase the size of the matching using v [11]. Thus, the end result will always be a maximum bipartite matching. This technicality is omitted in the pseudocode.

Assuming G is edge-partitioned and distributed among p processors, we initialize $\psi(v) := 0 \; \forall \; v \in V_1 \cup V_2$ and make all $v \in V_1$ active. Repeatedly calling the procedure Query and then Pulse on each processor until no processor has any active vertices left yields a maximum matching. However, the performance is weak. Even in the sequential case, the Push-Relabel algorithm requires global relabelings in order to make it competitive with other sequential matching algorithms,and the parallel case is no different.

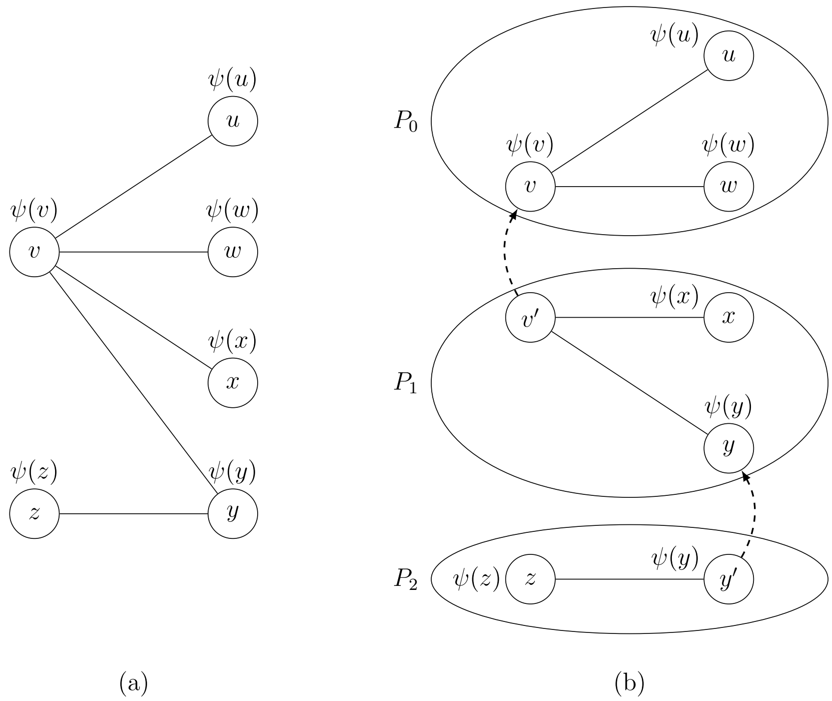


Figure 1: Sequential and Distributed Graph. In the sequential graph (a) each vertex v carries its own label $\psi(v)$. In the distributed graph (b), only the original vertex v of a left connector carries a label. The ghost vertex v' merely queries its neighbours labels $\psi(x)$ and $\psi(y)$ and reports them to v. In a right connector, both the original vertex y and the ghost vertex y' carry the label $\psi(y)$ which is updated every time its value changes.

Global Relabeling

The application of parallel global relabelings is necessary in order to obtain a scalable parallel algorithm as described in [2]. Performing such a relabeling in parallel is relatively simple. We start from each free right vertex u and run a parallel *alternating breadth first search*, i.e., a BFS which alternately uses unmatched and matched edges, through the entire graph. Each time we follow an edge $\{u, v\}$ to an unvisited vertex, its label $\psi(v)$ is set to $\psi(u) + 1$, the minimum BFS distance from a free right vertex. Because the graph is distributed among the processors, every time the search reaches a connector $C(w)$, a signal carrying the label $\psi(w)$ is sent to other vertices in $C(w)$. This communication is done in

Algorithm 1 Procedure PULSE

```
for each active vertex $v$ do
    Send Match signal $(v)$ to $u^*(v)$
    Make $v$ inactive
Exchange Match messages with other processors
for each vertex $u$ receiving a Match signal do
    if $u$ has not received a push in the current round then
        if $M(u) \neq \emptyset$ then
            Send Unmatch signal to $M(u)$
        $M(u) \leftarrow v$
        Send Accept signal to $v$
        $\psi(u) \leftarrow \psi(u) + 2$
        Set a flag indicating that $u$ received a push in the current round
    else
        Send Reject signal to $v$
Exchange Response messages with other processors
for each vertex $v$ receiving Accept signal do
    $M(v) \leftarrow u^*(v)$
    $\psi(v) \leftarrow \psi(u^*(v)) + 1$
for each vertex $v$ receiving a Reject or an Unmatch signal do
    Make $v$ active
Remote update $C_{i>0}(u) = C_0(u)$ for all $u \in V_2$
```

a manner similar to the communication in Query and Pulse. To ensure that $\psi(v)$ is actually a lower bound on the distance between v and the nearest free right vertex, the relabeling must progress at constant speed in all directions in the same way any BFS does.

While it is possible to mimic the sequential algorithm, i.e., stop the Query and Pulse rounds, perform such a global relabeling and then resume, it was pointed out in [2] that in order to obtain good scaling in the shared memory model, global relabeling should be interleaved with pushes. In the distributed memory model we face similar challenges. Stopping the Query and Pulse rounds to perform a complete global relabeling yields only mediocre load balance and poor scaling. There are two main reasons for this. The first is due to the fact that communication costs are likely to increase with the number of processors. The other reason is that all processors that have relabeled their vertices, as well as those that have not been reached by the relabeling yet, are necessarily idle and no processor can resume Query and Pulse rounds until all vertices have been relabeled. Thus, we implement global relabeling in such a way that it can be processed interleaved with the local computation of the Query and Pulse procedures.

This strategy is implemented using two new procedures, RelabelWave and Propagate. The procedure RelabelWave is called periodically. It increments the current wave number $\omega(u)$ for each free right vertex u and puts these vertices in a local propagate queue W. When starting the algorithm, all values of ω are initialized to 0.

The procedure Propagate is called every round following Query and Pulse. For every vertex u in the relabeling queue it relabels each neighbour $v \in \Gamma(u)$ with $\omega(v) < \omega(u)$, setting $\psi(v) = \psi(u) + 1$. For matched left vertices v, $\psi(M(v))$ is set to $\psi(u) + 2$ and $M(v)$ is put into a temporary queue. When the propagate queue is empty, the current relabeling process stops and vertices in the temporary queue are transferred to the propagate queue for use in the next round. If a left ghost vertex $C_j(v)$ would be relabeled from some vertex u, a Propagate signal with the wave number $\omega(u)$ and the new label $\psi(v)$ is sent to its owner $C_0(v)$ instead. If $\omega(u) > \omega(v)$, $C_0(v)$ is relabeled and $M(v)$, assuming it exists, is relabeled as described above and enqueued in the propagate queue by the owner of $C_0(v)$.

If a member of a right connector $C_j(u)$ is relabeled, again a Propagate signal with the new label $\psi(C_j(u))$ and wave number is sent to its owner $C_0(u)$, which, assuming it has not been relabeled in the current wave, broadcasts the new label to all ghost vertices $C_{k>0}(u)$. Their labels $\psi(C_k(u))$ are updated, and they are enqueued in the propagate queue. In both cases, relabeling from the remotely relabeled right vertices starts in the same manner as that from the locally relabeled right vertices in the next round when Propagate is called again. In any case, a vertex is never relabeled twice during the same relabeling wave. Correctness of the algorithm is ensured by restricting pushes in Pulse to edges (v, u) where $\omega(v) = \omega(u)$. See [2] for a proof which also applies to Algorithm 1 since the push and relabel strategies in both algorithms are essentially equivalent.

Algorithm 1 Procedure RelabelWave

```
Initialize local queue of vertices to relabel $W \leftarrow \emptyset$ when first called
for each free right vertex $w$ do
    $\omega(w) \leftarrow \omega(w) + 1$
    push $w$ onto $W$
```

The sequential algorithm executes a global relabeling after performing $\Theta(n)$ local relabelings. This turns out to be inadequate for the parallel implementation. Instead, RelabelWave is called after every r rounds of Query and Pulse to start a new global relabeling wave while Propagate is called every round to spread out existing waves. This allows the algorithm to interleave push operations with relabelings, although in any given round for some processors there will not be any vertices to push or relabel. Combining procedures Query, Pulse, RelabelWave, and Propagate we obtain Algorithm 1. Note that we do not

Algorithm 1 Procedure Propagate

```
Initialize temporary local queue $Q \leftarrow \emptyset$
while $W$ not empty do
    $w \leftarrow pop(W)$
    for each $v$ in $\Gamma(w)$ with $\omega(v) < \omega(w)$ do
        Send Left Propagate signal $(v, \psi(w), \omega(w))$ to $C_0(v)$
    for each remote processor $k$ where a $C_i(w)$ exists do
        Send a Right Propagate signal $(w, \psi(C_i(w)), \omega(C_i(w)))$ to $k$
Exchange Left Propagate messages with other processors
for each Left Propagate signal $(v, \psi(w), \omega(w))$ received do
    $\psi(v) \leftarrow \psi(w) + 1$
    $\psi(M(v)) \leftarrow \psi(v) + 1$
    $\omega(v) \leftarrow \omega(w) + 1$
    $\omega(M(v)) \leftarrow \omega(v) + 1$
    $push$ $M(v)$ onto $Q$
Exchange Right Propagate messages with other processors
for each Right Propagate signal $(w, \psi(C_i(w)), \omega(C_i(w)))$ received do
    if $\omega(w) < \omega(C_i(w))$ then
        $push$ $w$ onto $Q$
        $\psi(w) \leftarrow \psi(C_i(w))$
        $\omega(w) \leftarrow \omega(C_i(w))$
$W \leftarrow Q$
```

use the gap-relabeling heuristic described in [3], as this requires maintaining a global structure which is unsuitable for distributed memory computations.

5 A New Algorithm

As described above, Algorithm 1 is a distributed memory version of the parallel algorithm described in [2]. However, its performance is not competitive with sequential or shared memory parallel code. The reason for this lies in the fact that in most graphs, after matching the majority of vertices, the few remaining unmatched vertices are connected by relatively long augmenting paths. Even after suitable labels for such a path have been assigned, i.e., all edges on the path are admissible, augmenting along this path still takes a large number of consecutive pushes. On a sequential or shared memory machine, a single push can be performed quickly, but in Algorithm 1 procedures Query and Pulse must be called once per consecutive push. Since these procedures require communication, they are significantly slower than a simple push operation. This, along with an equally slow global relabeling routine, makes Algorithm 1 noncompetitive.

In this section we describe how some of these weaknesses of Algorithm 1 can be overcome by introducing the following three modifications. First by performing many push and relabel operations locally between communication rounds, then by routing augmenting paths in such a way that the number of processor jumps is minimized, and finally by balancing the amount of work across processors. In the remainder of this section, we show how to use these techniques in order to derive a new Algorithm 2 from Algorithm 1.

5.1 Local Work

One of the main drawbacks of Algorithm 1 is the high number of pulse operations. However, assuming the input graph is well partitioned, almost all of these pulses move a vertex on the same processor which suggests that communication is not required. Thus, by processing these pushes locally, it should be possible to speed up the algorithm significantly. However, in order to preserve correctness of the algorithm in the Push-Relabel scheme, one must ensure that one always pushes to a neighbour with a globally minimum label. To achieve this without resorting to communication for querying all neighbours, we only perform local push operations along admissible edges, i.e., edges $\{v, u\}$ where $\psi(v) \geq \psi(u) + 1$ (see Section 5.2). Thus, unlike in the Pulse procedure where an active vertex v simply sets its label $\psi(v)$ to $\psi(u) + 1$, we only push locally if $\psi(v) \geq \psi(u) + 1$ was already the case before the push, thereby following an assumed alternating path towards a free right vertex.

Assuming ψ is a valid distance labeling, we have $\psi(v) \leq \psi(u) + 1 \; \forall v \in V_1$, $u \in \Gamma(v)$ (see [15] for a complete proof). Thus, if an edge (v, u) is admissible, u is always a neighbour of minimum $\psi(u)$ for v. Therefore a push along an admissible edge is always legal for the Push-Relabel algorithm. Since the Push-Relabel algorithm remains correct for any ordering of legal pushes, it follows that we preserve a correct Push-Relabel strategy by applying local pushes. However, if such a push targets the original vertex in a right connector $C_0(u)$, $\psi(C_0(u))$ is increased. Consequently, labels of the ghost vertices $\psi(C_{i>0}(u))$ need to be updated. But since ghost vertices cannot be the target of a local push, it is not necessary to perform these remote update until after all local pushes have been executed.

Procedure Localwork implements this idea. It is called during each round before Query and Pulse. Even though we will relax the notion of a valid relabeling in the next section, Localwork will remain applicable.

5.2 Fast Relabeling

Performance can be increased by the addition of local work as described above, but the global relabeling as described in Section 4 is still very slow because it progresses by a single vertex per communication round. Also, it finds augmenting

Algorithm 2 Procedure LOCALWORK

```
while $\exists$ an active vertex $v$ that has an edge $\{v, C_0(u)\}$
with $\psi(C_0(u)) < \psi(v)$ and $\omega(v) \leq \omega(C_0(u))$ do
    if $M(u) \neq \emptyset$ then
        Make $M(u)$ active
        $M(M(u)) \leftarrow \emptyset$
    $M(u) \leftarrow v$
    $M(v) \leftarrow u$
    Make $v$ inactive
    $\psi(v) \leftarrow \psi(C_0(u)) + 1$
    $\psi(C_0(u)) \leftarrow \psi(C_0(u)) + 2$
Remote update $C_{i>0}(u) = C_0(u)$ for all $u \in V_2$
```

paths having a minimum number of edges, but with the introduction of Localwork such paths are no longer necessarily optimal as it is now desirable to have augmenting paths containing a minimum number of processor crossing edges. As Localwork allows pushes along paths on a single processor to be performed quickly, while pushes in Pulse that cross processors remain expensive, optimal augmenting paths will cross processors as infrequently as possible to minimize the number of communication rounds required for augmentation. Even on a distributed memory machine with fast interconnects, we determined the actual difference in cost between local and processor crossing pushes to be at least a factor of 10^3. Thus, in the following we assume that an optimal augmenting path has a minimum number of edges incident to at least one ghost vertex. We call this number the *processor distance* of an augmenting path.

We now show how to modify the global relabeling presented in Section 4 to deal with both of the above issues. We do this by changing the Propagate procedure so that all right vertices that have been relabeled are immediately enqueued in the local propagate queue W again which renders the temporary queue Q obsolete. This allows relabeling of all reachable vertices on a given processor in a single round without further communication. The transmission of Propagate signals is performed as described in Section 4, but not until after all local propagate queues are empty. Thus, in every round a global relabeling wave can progress by one processor, while relabeling all vertices it can reach on that processor that have not yet been relabeled in the current wave. As before, no vertex is relabeled more than once by the same wave.

In effect, the global relabeling has now become a parallel BFS traversal of a graph G' consisting of metanodes of vertices from G. All vertices reachable by local alternating paths from the unmatched right vertices on a processor form a single metanode, and all vertices on one processor that are reachable via processor crossing edges from an existing metanode V are assigned to a new metanode U.

The metanodes V and U are connected by an edge in G'. There are no edges between metanodes on the same processor. Once assigned to a metanode, a vertex is not assigned again. Note that given G, G' is not unique.

Clearly, such a traversal generates paths of minimum processor length, i.e., containing a minimum number of nonlocal edges. Since all vertices in a metanode are relabeled during the same communication round, this relabeling can be performed much faster as long as metanodes contain a large number of vertices.

As a side effect of this technique, we have to account for the possibility that the relabeling may not be valid anymore, i.e., there could be unmatched edges (v, u) with $v \in V_1$ such that $\psi(v) > \psi(u) + 1$. These edges are treated as *admissible* edges. We refer to a directed path from an active vertex consisting entirely of edges that are admissible using the labels assigned by relabeling wave k as a $k-$admissible path. PULSE and LOCALWORK can push along such edges, but labels of left vertices are never reduced as a result of such a push, since this might cause cyclic behaviour in the algorithm. The existence of edges where $\psi(v) > \psi(u) + 1$ implies that a push in LOCALWORK might no longer target a neighbour of minimum label. However, as we show in Section 5.4 we still retain correctness of the algorithm.

As a further modification, we add the setting of backpointers in the global relabeling. This means every time a vertex v is relabeled, we store the vertex from which v was reached in the variable $b(v)$. We refer to the edge $(v, b(v))$ as the back edge of v. In procedure LOCALWORK, an active vertex v will now always push to $b(v)$ as long as $\psi(v) > \psi(b(v))$ and $\omega(v) \leq \omega(b(v))$. Doing so avoids the search for a neighbour of minimum label (See Section 5.3 for modified wave restrictions). After a push from v, we set $b(v) = \emptyset$.

We now show that this modified relabeling procedure indeed finds augmenting paths of minimum processor distance. To do so, let $b(v, k)$ denote the vertex from which relabeling wave k reached v and let $f(v)$ be the free right vertex from which v was relabeled. We then call a path $P = (v_0, v_1, v_2, ..., v_l)$, where $v_0 = v$, $v_{i+1} = b(v_i, k)$, and $v_l = f(v)$, i.e., the path actually taken by wave k from $f(v)$ to v, a *k-relabeling path.*

Also, let $|C|$ be the number of connectors in the partitioned graph G. Define $d(v, k) = \infty$ for all vertices v that have not been relabeled by wave k. For a given set of free right vertices, the minimum processor distance of v is the minimum processor length of all alternating paths that connect v to some free right vertex.

Lemma 5.1. *For each vertex v relabeled by global relabeling wave k, there is no $k-$admissible path $P'(v, u)$ for any free right vertex u with lower processor distance than the k-relabeling path.*

Proof. The proof is by induction on the number of rounds l following the start of wave k.

If $l = 0$ then every vertex v of minimum processor distance 0 will be relabeled during the first call to PROPAGATE after the start of wave k, thus producing an augmenting path $P(v, f(v))$ consisting entirely of local edges. It also follows that no vertex w of minimum processor distance greater than 0 can be relabeled during the current call to PROPAGATE as this would require a Propagate signal to be sent to the processor owning w.

Now assume that $l > 0$ and that each vertex w of processor distance below l has been relabeled correctly by wave k while no vertex of processor distance at least l has been relabeled so far. Let v be a vertex on processor a with a processor distance of l. There must exist a connector $C(u)$ on a minimum processor distance path from v such that an instance $C_b(u)$ of $C(u)$ has minimum processor distance $l-1$ on some processor b. In slight abuse of notation, let $C_a(u)$ be the instance of $C(u)$ on processor a. By induction it then follows that $C_b(u)$ has been relabeled during round $l-1$ and that prior to round l a Propagate signal is sent to $C_a(u)$.

Thus since there exists an augmenting path from $C_a(u)$ to v it follows that v will be relabeled during the l'th round. Moreover, no vertex of processor length greater than l can have received a Propagate signal from wave k during round l as this would have required that wave k had already reached some vertex of minimum processor distance at least l prior to round l. □

Since a vertex is only relabeled once per wave, even if a new relabeling would reduce its label, the global relabeling assigns an alternating path of minimum processor length, but not necessarily of minimum edge length to each vertex. As the relabeling is interleaved with pushes, the path so created might be interrupted by pushes and their respective updates of ψ before it is complete.

5.3 Load Balance

The introduction of LOCALWORK and the modified global relabeling increase performance by performing as much work as possible locally and without communication, but it might lead to poor load balance when only few processors are able to perform local computations. However, the load balance of the interleaved global relabeling presented in Section 5.2 can be improved by modifying the ratio between communication and local relabeling work.

To do so, we introduce a relabeling propagation speed s. Again every r rounds, a new wave is sent out from every free right vertex, but every processor relabels only a $\frac{1}{s}$ fraction of local vertices per round. It is easy to see that this strategy allows far better load distribution. However, for values of $s > 1$, this might require an increased number of communication rounds.

Also, the paths found by the global relabeling are no longer guaranteed to be of minimum processor length because only part of a processor's vertices can be relabeled. Thus the algorithm faces a tradeoff here. Effects of various values of s on performance are studied in Section 7.

Algorithm 2 Procedure Propagate

```
while $W$ not empty do
    $w \leftarrow pop(W)$
    for each $v$ in $\Gamma(w)$ with $\omega(v) < \omega(w)$ do
        if $v$ is original vertex $C_0(v)$ then
            $\psi(v) \leftarrow \psi(w) + 1$
            $\psi(M(v)) \leftarrow \psi(v) + 1$
            $\omega(v) \leftarrow \omega(w) + 1$
            $\omega(M(v)) \leftarrow \omega(v) + 1$
            push $M(v)$ onto $W$
        else
            Send Left Propagate signal $(v, \psi(w), \omega(w))$ to $C_0(v)$
    for each remote processor $k$ where a $C_i(w)$ exists do
        Send a Right Propagate signal $(w, \psi(C_i(w)), \omega(C_i(w)))$ to $k$
Exchange Left Propagate messages with other processors
for each Left Propagate signal $(v, \psi(w), \omega(w))$ received do
    $\psi(v) \leftarrow \psi(w) + 1$
    $\psi(M(v)) \leftarrow \psi(v) + 1$
    $\omega(v) \leftarrow \omega(w) + 1$
    $\omega(M(v)) \leftarrow \omega(v) + 1$
    push $M(v)$ onto $W$
Exchange Right Propagate messages with other processors
for each Right Propagate signal $(w, \psi(C_i(w)), \omega(C_i(w)))$ received do
    if $\omega(w) < \omega(C_i(w))$ then
        push $w$ onto $W$
        $\psi(w) \leftarrow \psi(C_i(w))$
        $\omega(w) \leftarrow \omega(C_i(w))$
```

To prevent relabeling waves from locking out pushes towards the free right vertices, we allow a push along an edge $\{v, u\}$ if $\omega(v) \leq \omega(u)$. The reason for this is that because wave $\omega(u)$ is more recent than $\omega(v)$, it follows that u and $M(u)$ are likely to be closer to a free right vertex because otherwise v would have been relabeled by wave $\omega(u)$ already. In Section 5.4 we show that the algorithm remains correct nonetheless.

Algorithm 2 implements the presented ideas. It calls Localwork, Query, Pulse, and Propagate every round, and RelabelWave every r rounds. Procedure Query now allows pushes along edges $\{v, u\}$ with $\omega(v) \leq \omega(u)$, and Propagate was modified as described in Section 5.2. Just like in the sequential Push-Relabel algorithm where finding a good relabeling frequency was crucial, setting a correct propagation speed s and relabeling frequency r is paramount for performance. See Section 7 for a detailed evaluation of the effects the parameters have on performance.

5.4 Correctness of Algorithm 2

It remains to show that Algorithm 2 remains correct, even though the main invariant of Push-Relabel algorithms, i.e., maintaining a valid relabeling, can be violated. In the following we show that Algorithm 2 in fact terminates. The worst case running time we obtain in this way is weak compared to other algorithms because the proof only considers augmenting paths found directly by global relabelings, not those found by labels alone. To show correctness, we first consider the effect of pushes on augmenting paths. For simplicity, we assume $s = 1$, i.e., no special load balancing. Our first step is to show that once we have an augmenting path, pushes and relabels to the vertices on this path will always result in a new and possibly shorter augmenting path. To count processor jumps, let $d(v, k)$ be the processor distance between v when it is relabeled by wave k and the source of the relabeling in wave k, and let $d(v) = d(v, k^*)$ for the latest relabeling wave k^* that relabeled v.

Lemma 5.2. *Let v be an active vertex and let $P(v, f(v))$ be an augmenting path connecting v to some free right vertex $f(v)$. Then if $u \in P$, $u \neq f(v)$ is the target of a successful push we obtain a suffix path $P'(v', f(v)) \subset P$ where v' is active.*

Proof. Since P can span multiple processors, several such pushes can happen at the same time. Given multiple possibilities, let u be the the vertex closest to $f(v)$ on P. Since u is the target of a push originating at some active vertex w, u is a right vertex and because we assumed $u \neq f(v)$ by the definition of an augmenting path, u is matched to some $M(u) \in P$. Let $M(u) = v'$. Since v' is a left vertex, the matching edge (u, v') enters v' when traversing P from v to $f(v)$. Since the push operation changes $M(u)$ to w, it renders v' unmatched. Now, as the only edges affected by the push are (w, u) and (v', u) all edges of the suffix path $P'(v', f(v))$ remain unaffected by the push. Thus, since v' is active and $f(v)$ is a free right vertex, P' is an augmenting path. □

The next step is to show that for $k-$admissible paths, the suffix path cannot increase in processor length.

Lemma 5.3. *Let v be some active vertex, and let $P(v, f(v))$ be a $k-$relabeling path connecting v to some free right vertex $f(v)$, and let $u \in P$, $u \neq f(v)$ be the target of a successful push. Then we obtain a suffix path $P'(v', f(v)) \subset P$ with $d(v', k) \leq d(v, k)$ with a processor length smaller or equal to that of P.*

Proof. Since the global relabeling progresses only along alternating paths, $P(v, f(v))$ must be an augmenting path and thus we invoke Lemma 5.2 to show existence. From Lemma 5.1 it is clear that there is no shorter $k-$admissible path from v to

$f(v)$ w.r.t. processor distance than P. And as processor distance cannot increase when following P from v to $f(v)$, we obtain $d(v', k) \leq d(v, k)$. □

The third step establishes existence and minimality of such paths during the algorithm:

Lemma 5.4. *Let v be the first active vertex to be relabeled by relabeling wave k, and let round R_1 be the round in which v is so relabeled. Also, let R_2 be the round in which the first free right vertex has been matched after round R_1. At any time during the algorithm between rounds R_1 and R_2, there is an active vertex v' of minimum $d(v')$ in G that is the endpoint of an admissible augmenting path.*

Proof. Consider a global relabeling wave k after it relabels v in round R_1. It has created a $k-$relabeling path that links the active vertex v to some free right vertex $f(v)$. Note that v has minimum $d(v, k)$ among all active vertices.

Now, if the $k-$relabeling path to v is an augmenting path we only need to show that it remains admissible. In that case, let $v' = v$. Otherwise, by Lemma 5.3, there is some v' on an augmenting suffix path $P'(v', f(v))$ with $d(v', k) \leq d(v, k)$.

To see that there is a $P'(v', f(v))$ which is admissible, remember that P' was a part of the $k-$relabeling path to v'. Since no pushes can have affected vertices on $P'(v', f(v))$ and it was marked by relabeling wave k with backpointers, it must be admissible unless the backpointers were changed by some global relabeling wave $k' > k$. But if this has happened, v' must be reachable via a $k'-$relabeling path which again is admissible.

Since $f(v)$ was not matched, $d(v', k') \leq d(v', k)$ because otherwise the k'-relabeling path to v' would have started from $f(v)$. □

Now we need to bound the time between R_1 and R_2.

Lemma 5.5. *Let v be the first active vertex to be relabeled by relabeling wave k, and let round R_1 be the round in which v is so relabeled. After round $R_1 + d(v, k) + 1$, at least one free right vertex must have been matched since the start of relabeling wave k.*

Proof. By Lemma 5.4, at any time between R_1 and R_2, there is some active vertex v' of minimum $d(v', k)$ that is the endpoint of an augmenting path consisting entirely of admissible edges marked with backpointers.

During every round, procedure Localwork will perform a series of pushes along such a path until it reaches a connector. Thus, after Localwork finishes, there is some active vertex w that is a member of a left connector. It will either be pushed along its backedge in procedure Pulse, or $b(w)$ was the target of a successful push after it was relabeled by wave k. In both cases, there must now be some active vertex $v' \in P'$ with $d(v', k) < d(w, k)$.

Again v' will be the endpoint of an augmenting path marked with backpointers. Thus, after at most $d(v, k)$ rounds, there will be such a vertex v' on an augmenting admissible path $P'(v', f(v'))$ with $d(v', k) = 0$. The next time Localwork is called, augmentation along P' causes v' and $f(v')$ to be matched. Since $f(v')$ was a free right vertex, this concludes the proof of the lemma. □

We call a round in which $|M|$ has increased by at least one $|M|$-*increasing.* Let k be the first global relabeling wave after some $|M|$-increasing round, and let $k' > k$ be the first global relabeling wave following the first $|M|$-increasing round after the start of k. We call the rounds between k and k' a phase. Clearly, the number of phases is bounded by n. Using Lemma 5.5, it is easy to show that each phase terminates.

Lemma 5.6. *A phase takes at most* $2|C| + 2 + r$ *rounds.*

Proof. Global relabel wave k takes at most $|C|+1$ rounds to relabel its first active vertex v, since $|C|$ is an upper bound on all finite processor distances. By Lemma 5.5, at most $d(v, k)+1 < |C|+1$ rounds later a free right vertex is matched, thus increasing $|M|$. A new global relabel wave happens at most r rounds after $|M|$ is increased. Thus, a phase takes at most $2|C| + 2 + r$ rounds.

□

Lemma 5.6 shows that the modified algorithm is guaranteed to terminate, and since moving free vertices along paths of minimum processor-length maximizes the opportunity for local work while minimizing the necessity for communication, the modified wave based relabeling should improve performance. Note that the relabeling might jump back to unreached vertices on a processor that was reached by the same wave earlier, possibly resulting in paths of processor length up to $|C|$. If a perfect matching does not exist, the algorithm terminates after a complete wave failed to relabel any active vertex. Clearly, this cannot take more time than a single phase.

For values of $s > 1$, the proof has to be modified slightly. Since it is not specified which $\frac{1}{s}$ fraction of the vertices of a given processor are relabeled each round, a global relabeling can take up to s rounds to traverse one processor. Thus, processor distance of the $k-$relabeling path $P(v, f(v))$ can be s times greater than that of a minimum augmenting path from v to $f(v)$. However, with slight modification Lemma 5.4 still applies since an augmenting path exists. Lemma 5.5 can be extended to show that longer paths are also shortened by Localwork. Thus, it is possible to derive a $2s(|C| + 1) + r$ time bound for each phase. Note that Lemma 5.5 holds even if free vertices can decrease in processor distance due to a new relabeling wave, which is likely to happen for $s > 1$.

6 Experimental Setup

In the following we describe the experiments that were used to test the performance of Algorithm 2, to compare it to sequential performance, and to measure performance scaling for increasing number of processors.

All experiments were performed on a Cray XT4 equipped with AMD Opteron quad-core 2.3 GHz processors. Codes are written in C++ with MPI using the MPICH2 based xt-mpt module version 5.0.2 and the PathScale Compiler Suite 3.2.99. Tests were performed using configurations of $1, 2, 4, 8, 16, 32, 64$, and 128 processors.

The test consists of a group of large square matrices from the University of Florida Sparse Matrix Collection [5]. Matrices were partitioned using the Mondriaan graph partitioning package version 2.0 [24]. The main program reads the partitioned matrix as input and distributes it among the processors to build the parallel data structures. When measuring running time, this part was not included in the measurement since the algorithm is intended for inputs that are distributed to begin with.

The graphs are derived from the matrices as follows: For a given square matrix A of size $n \times n$ its bipartite graph $G = (V_1 \cup V_2, E)$ is obtained by setting $|V_1| = |V_2| = n$ and $E = \{\{i, j\} \in V_1 \times V_2 : a_{i,j} \neq 0\}$.

The parallel computation starts by initializing the matching using a sequential Karp–Sipser style heuristic [14] locally on each processor. The heuristic maintains a priority queue of local vertices, sorted by degree. Vertices of degree one are matched immediately. Each time a vertex is matched, it is removed along with its incident edges, thereby reducing the degrees of some remaining vertices and thus changing their position in the priority queue. If no vertices of degree one are available, the heuristic selects a random local edge and matches the incident vertices. If no edge remains, the heuristic terminates. Ghost vertices and their incident edges are not considered in the heuristic.

For sequential computations, this heuristic can speed up matching algorithms greatly [16], usually producing a matching within 99.5% of the optimum. However, performing an equivalent parallel version of this heuristic would incur additional communication cost. Thus, in the parallel initialization, ghost vertices are not considered, which means that the solution quality is necessarily lower than for the sequential algorithm, and it also decreases with an increasing number of processors. After the initialization, Algorithm 2 is executed as described in Section 5.

Optimal values for the two relabeling parameters, i.e., the frequency of global relabelings r and s, the fraction of local vertices that can be relabeled in each round, are not known. In a first set of experiments we determine good values for r and s and use the values so obtained in a second set of experiments to evaluate performance and scaling of Algorithm 2. For comparison with sequential performance, we implemented a Push-Relabel algorithm as described in Section 3. The implementation uses a FIFO order of pushes, and a Karp–Sipser heuristic initialization as described above. We also ran Algorithm 2 using only one processor. This is essentially a sequential computation, but our code is not designed to adapt to this. Therefore, it still calls all the normal communication routines, making it inferior to the purely sequential code. It does keep the advantage derived from the sequential initialization though.

7 Experimental Results

Our first set of experiments aims to explore the effect relabeling frequency r and relabeling speed s (see Section 5.3) have on the overall performance. To do so, we selected the seven matrices *hamrle3, cage14, ldoor, kkt_power, parabolic_fem, av41092 and rajat31* from the University of Florida Sparse Matrix Collection [5]. The sizes of these matrices are listed in Table 3. We then ran the algorithm on each matrix for all combinations $(s, r) \in \{1, 2, 4, 8, 16, 32, 64, 128\}^2$ and all configurations $p \in \{1, 2, 4, 8, 16, 32, 64, 128\}$, obtaining $7 * 8^3$ individual timings. Since presenting $3,584$ numbers would yield little insight, we aggregate these results in order to present our conclusions.

We first note that the matrix *rajat31* behaved in a different way from the other instances. Here the KARP–SIPSER initialization matched either all or, for values of p above 4 almost all vertices. Thus, running times were very low and completely independent of the relabeling parameters. Therefore, *rajat31* is not considered in the analysis of parameter effects. When using a single processor, *ldoor* and *cage14* showed similar behaviour.

Since running times vary widely over instances and processor configurations, they need to be normalized for comparison. For each of the 48 combinations of the 6 remaining instances and the 8 different p values, we do this by dividing the running time of each (r, s) combination by the best running time for this $(instance, p)$ combination. This gives the best settings of r and s in each $(instance, p)$ combination a performance of 1. For other (r, s) combinations, the normalized performance is a number between 0 and 1. Experiments that failed because of memory shortage were assigned a performance of 0.

To obtain parameter dependent performance over multiple test instances, we averaged the performance values for each combination of r, s, and p over all matrices except *rajat31*. The results are shown in Figure 2. Each chart in the figure is a map of the effect of s and r for one processor configuration. We observe that high performance, i.e., dark red areas can usually be found along and above the $s = r$ line, with optimal values of r decreasing with increasing processor numbers. It is easy to see that setting $s > r$ yields weak performance.

Thus, to obtain maximum performance when increasing p, r should be halved for each doubling of p, and s should be modified accordingly. Doing so will ensure that the relabel work per processor per round remains roughly constant, since the partition of the input assigned to a single processor decreases. This allows the algorithm to maintain a good computation to communication ratio.

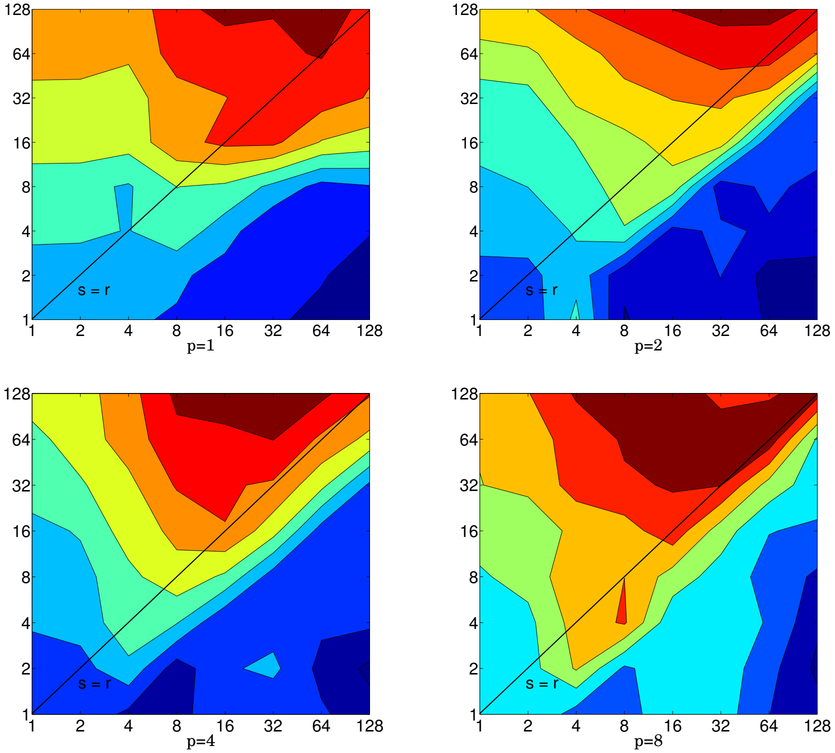


Figure 2: Relative performance for varying parameters and processor numbers. High performance is shown in red. Colder colors indicate lower performance. r is shown on the $X-$axis, s on the $Y-$axis. With increasing values of p, high performance moves from high to low values of s and r. Figure continues on page 23. For all plots except $p = 128$ a sharp drop in performance below the $s = r$ line is visible.

Since relabeling become more frequent, the expected number of rounds decreases, thereby allowing the algorithm some scaling performance with increasing p. This also suggest a natural scaling limit which is reached at $s = 1$ and $r = 1$. For some matrices this can clearly be observed. For example *parabolic_fem* at $p = 64$ performs best for $s = 1$ and $r = 1$, and does not scale beyond this number of processors.

From these results, we obtained good guidelines for setting parameter values, but the actual optimal values for a given instance are not clear. For the second

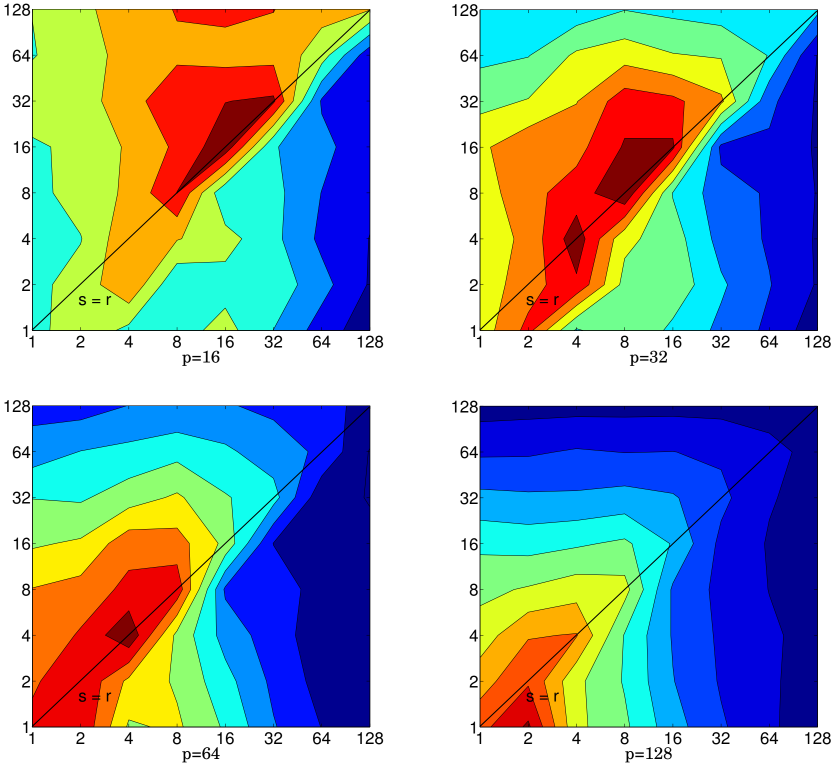


Figure 2: continued. Relative performance for 16, 32, 64, and 128 processors.

experiment, we used the estimated optimum values r and s for the average over the instances and ran Algorithm 2 without further input parameters. The *average optimum* values for (r, s) are shown in Table 1.

Table 1 also shows the optimal values of r and s for each individual $(instance, p)$ combination in Experiment 1, while Table 2 shows the timing for these (r, s) combinations compared to non-parameterized running times from experiment 2. On average, the optimal running times are 83% of the non-parameterized running times. From these values we computed the *Parameter influence ratio* which is also shown in Table 1. It is the performance averaged over all (r, s) combinations divided by the performance of the average optimum. It measures the inverse of the influence parameter settings have on performance. For $p = 1$ the influence is quite low, but it gradually grows and for $p = 128$ it is very high.

Table 1: Optimum parameters by number of processors and instance. The table shows the combinations of (r, s) at which performance in the individual experiments was maximum. Average optimum shows the optimum (r, s) values for normalized running that were averaged over the test instances. An entry of *all* indicates that the parameter is irrelevant for performance. Parameter influence ratio is the performance averaged over all (r, s) combinations divided by the performance of the average optimum.

Matrix	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
Optimum (r, s) by instance								
hamrle3	128,16	128,128	128,8	64,64	16,16	2,4	1,2	1,2
ldoor	all,all	64,1	128,8	128,4	64,8	16,16	2,2	2,2
cage14	all,all	128,128	128,64	64,16	64,32	32,8	16,4	1,2
kkt_power	4,128	128,64	128,32	64,32	1,64	16,8	1,16	1,2
parabolic_fem	128,16	64,32	128,16	2,4	2,4	1,2	1,1	1,1
av41092	128,1	64,64	4,64	2,16	16,16	8,8	4,2	4,2
Average optimum (r, s)	128,64	128,128	64,64	32,32	16,16	8,8	4 ,4	2,2
Parameter influence ratio	0.867	0.540	0.533	0.552	0.509	0.441	0.361	0.270

Table 2: Best running times over all values of s and r in the parameterized experiment compared to running times in the non-parameterized experiment with fixed s and r for each instance and processor number. Times are in seconds.

Matrix	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
hamrle3	4.55 · 4.61	13.1 · 13.1	8.01 · 12.8	6.96 · 7.32	5.88 · 5.88	2.87 · 3.02	3.21 · 3.43	2.75 · 3.55
cage14	2.68 · 2.68	7.76 · 7.76	6.18 · 9.40	5.03 · 5.91	3.20 · 4.31	2.39 · 3.32	2.20 · 2.34	1.49 · 2.19
ldoor	1.98 · 1.98	8.09 · 8.99	4.75 · 5.97	2.82 · 3.68	1.96 · 2.39	1.33 · 1.33	1.13 · 1.13	1.17 · 1.35
kkt_power	4.08 · 4.51	7.43 · 8.52	6.49 · 8.93	3.47 · 6.87	3.16 · 5.23	1.69 · 2.10	1.53 · 1.98	1.49 · 1.85
parabolic_fem	0.59 · 0.59	3.26 · 3.73	1.72 · 2.30	1.04 · 1.48	0.84 · 1.14	0.38 · 0.74	0.25 · 0.59	0.37 · 0.52
av41092	0.25 · 0.25	0.47 · 0.62	0.46 · 0.56	0.34 · 0.43	0.41 · 0.41	0.57 · 0.57	0.90 · 0.92	1.50 · 1.76

For the second experiment the test set consisted of 22 matrices from the University of Florida Sparse Matrix Collection [5]. Table 3 gives sequential and parallel running times. The instances are grouped by scaling behaviour.

The first group consists mostly of smaller instances with up to 10^6 nonzeros. The algorithm shows little or no scaling here. Matrix *av41092*, the largest in this group, reacts poorly to increasing values of p. This is consistent with the behaviour of other parallel algorithms [21], although it is far less drastic here.

The second group contains medium sized instances with more than 10^6 nonzeros. In this group most instances show reasonable scaling which usually peaks at $p = 16$ or $p = 32$. Running times increase noticeably at $p = 64$. The largest matrix in this group, *Hamrle3*, is a very difficult instance even for sequential algorithms. Unlike other instances in this group, it shows very good scaling that peaks at $p = 32$.

For the two remaining groups, scaling behaviour can clearly be observed. In the last group, running time drops by approximately 25% on average for each doubling of p. From the size of these instances, we expect Algorithm 2 to scale well on average matrices of at least 10^7 nonzeros. However, since scaling already declines at $p = 128$ we believe that even larger matrices are necessary to continue scaling beyond 512 processors. The largest matrix in the fourth group, *Audikw_1*, could not be partitioned using Mondriaan due to lack of memory and is therefore only block partitioned. Performance compared to the sequential algorithm would most likely be better if it was properly partitioned.

We observe that the sequential algorithm is about three times faster than Algorithm 2 using one processor. This difference is due to the far simpler structure of the sequential algorithm, and to the fact that the sequential algorithm can automatically adapt the relabeling frequency. Also, using a single processor Algorithm 2 is again approximately twice as fast as using two processors. The reason for this lies in the fact that the sequential initialization is far superior. As shown in [16], initialization has a strong effect on running time. Good initialization also tends to make the FIFO order of pushes more competitive.

However, the difference in sequential and parallel performance decreases for larger matrices, but, due to the inherent sequentiality of matching a small number of unmatched vertices, scaling will most likely be limited for any parallel algorithm. For the test matrices *ASIC_680ks*, *kkt_power*, and *rajat31*, the parallel algorithm delivers performance that is superior to that of the sequential algorithm. For *rajat31*, this happens because even for the parallel case the initialization matches most vertices, and the remaining vertices are matched without starting a global relabeling.

It is interesting to note that for difficult matrices such as *Hamrle3*, i.e., matrices that require high running times relative to their size, the performance of the sequential algorithm is only slightly better than that of Algorithm 2 using at least 32 processors. This is most likely due to the fact that both algorithms perform a large number of global relabelings here, but the parallel algorithm can perform a single relabeling faster. Thus, Algorithm 2 becomes more competitive for very easy and for very difficult instances.

From the results of the experiments, we can conclude that:

- To some degree, the algorithm scales for moderately sized instances (about 10^6 nonzeros). For larger instances, it continued to scale at $p = 128$.

- For low values of p, global relabels should be infrequent. Since the processor length of augmenting paths will be low, the need for global relabelings is likely to be lower compared to experiments with higher processor count. Also, the number of operations per processor in a global relabeling is high, thus making it costly. For high p, the opposite is the case, which explains the better performance of low values of r there.

Table 3: Results for the performance experiment. Instances are grouped by scaling behaviour and ordered by number of nonzeros. Running times are in seconds.

Matrix	#Rows	#Nonzeros	Seq.	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=128
No scaling											
Zhao2	33,861	166,453	0.01	0.03	0.09	0.12	0.13	0.15	0.16	0.21	0.40
ncvxqp5	62,500	237,483	0.03	0.08	0.26	0.21	0.22	0.22	0.23	0.22	0.48
c-71	76,638	859,554	0.03	0.08	0.25	0.27	0.27	0.25	0.24	0.27	0.42
kim1	38,415	933,195	0.01	0.09	0.05	0.07	0.09	0.12	0.24	0.16	0.40
twotone	120,750	1,224,224	0.05	0.11	0.29	0.32	0.24	0.21	0.24	0.23	0.55
av41092	41,092	1,683,902	0.10	0.25	0.62	0.56	0.43	0.41	0.57	0.92	1.76
Scaling up to 32											
scircuit	170,998	958,936	0.04	0.14	0.44	0.29	0.25	0.15	0.17	0.23	0.51
ibm_matrix_2	51,448	1,056,610	0.02	0.06	0.38	0.33	0.33	0.25	0.27	0.46	0.53
crashbasis	160,000	1,750,416	0.04	0.16	0.31	0.14	0.1	0.04	0.05	0.23	0.17
matrix_9	103,430	2,121,550	0.03	0.13	1.13	0.38	0.41	0.33	0.30	0.41	0.61
ASIC_680ks	682,712	2,329,176	0.21	0.41	0.21	0.14	0.15	0.09	0.06	0.08	0.17
poisson3Db	85,623	2,374,949	0.09	0.27	0.36	0.37	0.26	0.19	0.19	0.26	0.48
barrier2-4	113,076	3,805,068	0.05	0.16	0.56	0.46	0.35	0.41	0.30	0.52	0.76
Hamrle3	1,347,360	5,514,242	2.56	4.61	13.12	12.81	7.32	5.88	3.02	3.43	3.55
Scaling up to 64											
bone010_M	986,703	23,888,775	0.43	1.61	10.5	6.32	3.68	2.53	1.23	0.95	1.56
ldoor	952,202	42,493,817	0.73	1.98	9.38	5.97	3.68	2.39	1.33	1.13	1.35
Scaling at 128											
parabolic_fem	525,825	3,674,625	0.16	0.59	3.73	2.30	1.48	1.14	0.74	0.59	0.52
kkt_power	2,063,494	12,771,361	1.95	4.51	8.52	8.93	6.87	5.23	2.10	1.98	1.85
af_shell2	504,855	17,588,875	0.27	0.77	5.00	3.12	2.06	2.07	1.10	0.95	0.78
rajat31	4,690,002	20,316,253	1.06	2.96	1.54	0.85	0.99	0.57	0.30	0.18	0.12
cage14	1,505,785	27,130,349	0.69	2.68	7.76	9.40	5.91	4.31	3.32	2.34	2.19
Audikw_1	943,695	39,297,771	1.18	2.82	14.24	8.95	4.11	3.90	3.73	2.64	1.96

- Using low relabeling speeds, i.e., high values of s generally increases performance by improving load balancing. However, setting $s > r$ yields weak performance. This is to be expected because in this case a relabeling wave might not be able to relabel all vertices on a processor before a new wave starts. For $s = 128$, running times were in many instances slower than the best running times by three orders of magnitude.
  Thus, setting $s = r$ becomes optimal since this provides the strongest load balance possible that avoids the above problem. However, this is somewhat instance dependent. Often, lower values of s provide performance similar to that for $s = r$, but the performance when setting $s = r$ is rarely exceeded significantly, which suggests using this relation for further experiments.

- The optimum number of global relabels depends to some degree on the instance, but this cannot be analysed beforehand and thus Algorithm 2 cannot take this into account. However, on difficult instances Algorithm 2 scales well and shows performance competitive with the sequential algorithm.

- As a rule of thumb, doubling the processor number halves the optimal values of r and s.

- With a larger number of processors the effect of parameters on performance increases (see Table 1). For $p = 128$, average performance over all values of s and r examined is only 27% of the optimum performance.

- Performance of the sequential algorithm is usually superior to performance of Algorithm 2, unless instances are very large, very difficult or very easy.

- The performance of the algorithm could be improved by about 20% by automatically adapting s and r towards the optimum for the current instance. The sequential algorithm does this by starting a global relabeling after $O(n)$ local relabels. Attempts to introduce a similar mechanism on the parallel algorithm were not successful. (see Table 2) It is possible that varying s and r during a run of the algorithm results in even larger performance gains.

8 Conclusions and Further Work

Although the execution time of Algorithm 2 in most cases does not improve on the running time of the sequential Push-Relabel algorithm it is still the first scalable parallel algorithm for the bipartite matching problem. The main usefulness of the algorithm is in parallel applications where the data is already partitioned and distributed on the processors. In this setting the alternative to a parallel algorithm would be to gather the data on one processor before applying a sequential algorithm and then again distributing the solution. This might be too costly in terms of time and it might also not be possible because of memory limitations on a single compute node.

When comparing experimental results for Algorithm 2 to those for the shared memory maximum flow algorithms that Algorithm 2 is based on, we note that relative to processor speed, memory access on the shared memory machines is still faster than using the interconnects on a distributed memory supercomputer. Furthermore, bipartite matching tends to exhibit a smaller amount of parallelism than maximum flow [2]. Still, with the exception of the one processor case, the scaling behaviour of Algorithm 2 compares favourably to the results presented in [3]. However, Algorithm 2 was unable to provide a speedup comparable that reported in [22], but this might be due to the fact that sequential algorithms profit disproportionately from recent advances in processor technology.

If one is to improve further on the parallel running time of Algorithm 2 there are different options that one could pursue. A fairly simple one would be to use a parallel version of the Karp–Sipser algorithm for initialization [19]. This would probably be most helpful for instances using a high number of processors. However, the main problem with Algorithm 2 is that a significant amount of the execution time is spent on the relabeling. A relabeling wave touches every reachable vertex independent of whether it is on an alternating path to an active vertex or not. Also, a vertex will be continuously relabeled even if its path to a free right vertex has not changed. One way to speed up global relabelings might be to use local graph compression techniques such as those discussed in [10, 16, 17].

We note that the ideas of local work leading up to Algorithm 2 might also be applied in the shared memory model giving an algorithm where more work could be performed by each processor in between synchronizations.

A closely related problem in combinatorial scientific computing is finding a matching of maximum weight in addition to maximum cardinality. One way to obtain an approximation for this problem would be to first apply Algorithm 2 to compute a maximum matching without regards to the weights of the edges. Based on this solution one can then search for weight augmenting cycles and augment along these. Depending on the length of the longest cycle one searches for and whether such cycles can span several processors or not one obtains algorithms with different solution quality and timing properties. Exploring such strategies is a topic for further study but we note that preliminary results indicate that this is a promising approach.

References

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, *Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/logn})$*, Information Processing Letters, 37 (1991), pp. 237–240.

[2] R. Anderson and J. C. Setubal, *A parallel implementation of the push-relabel algorithm for the maximum flow problem*, Journal of Parallel and Distributed Computing, 29 (1995), pp. 17–26.

[3] D. A. Bader and V. Sachdeva, *A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic*, in proceedings of the 18th International Conference on Parallel and Distributed Computing Systems, 2005, pp. 41–48.

[4] B. V. Cherkassky, A. V. Goldberg, P. Martin, J. C. Setubal, and J. Stolfi, *Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms*, ACM Journal on Experimental Algorithmics, 3 (1998).

[5] T. A. Davis, *The university of florida sparse matrix collection*, IEEE Transactions on Circuits and Systems. To appear.

[6] I. S. Duff, *Algorithm 575: Permutations for a zero-free diagonal [F1]*, ACM Transactions on Mathematical Software, 7 (1981), pp. 387–390.

[7] ——, *On algorithms for obtaining a maximum transversal*, ACM Transactions on Mathematical Software, 7 (1981), pp. 315–330.

[8] I. S. Duff and B. Uçar, *Combinatorial problems in solving linear systems*, in proceeding of the Dagstuhl Seminar on Combinatorial Scientific Computing, U. Naumann, O. Schenk, H. D. Simon, and S. Toledo, eds., no. 09061, Dagstuhl, Germany, 2009.

[9] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, Journal of the ACM, 19 (1972), pp. 248–264.

[10] T. Feder and R. Motwani, *Clique partitions, graph compression and speeding-up algorithms*, Journal of Computer and System Sciences, 51 (1995), pp. 261–272.

[11] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem*, in proceedings of the 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 136–146.

[12] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, *BSPlib: The BSP programming library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[13] J. E. Hopcroft and R. M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM Journal on Computing, 2 (1973), pp. 225–231.

[14] R. M. Karp and M. Sipser, *Maximum matchings in sparse random graphs*, in proceedings of the 22nd Annual Symposium on Foundations of Computer Science (FOCS 1981), IEEE, 1981, pp. 364–375.

[15] B. H. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, Birkhäuser, 2006.

[16] J. Langguth, F. Manne, and P. Sanders, *Heuristic initialization for bipartite matching problems*, ACM Journal of Experimental Algorithmics, 15 (2010), pp. 1.3:1–1.3:22.

[17] M. Löhnertz, *Algorithmen für Matchingprobleme in speziellen Graphklassen*, PhD thesis, Universität Bonn, 2010.

[18] M. Manguoglu, A. H. Sameh, and O. Schenk, *PSPIKE: A parallel hybrid sparse linear system solver*, in proceedings of the 15th International

European Conference on Parallel Processing (Euro-Par 2009), Delft, The Netherlands, 2009, Springer-Verlag, pp. 797–808.

[19] M. M. A. Patwary, R. H. Bisseling, and F. Manne, *Parallel greedy graph matching using an edge partitioning approach*, in proceedings of the Fourth ACM SIGPLAN Workshop on High-level Parallel Programming and Applications (HLPP 2010), 2010, pp. 45–54.

[20] A. Sangiovanni-Vincentelli, *A note on bipartite graphs and pivot selection in sparse matrices*, IEEE Transactions on Circuits and Systems, 23 (1976), pp. 817–821.

[21] M. Sathe, O. Schenk, F. Müller, Y. Zhao, and H. Burkhart, *Accelerating maximum weighted matching algorithms in massive graph analysis*, in the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), 2011. Submitted.

[22] J. C. Setubal, *New experimental results for bipartite matching*, in proceedings of the Network Optimization, Theory and Practice (NETFLOW 1993), 1992.

[23] Ümit V. Çatalyürek, E. Boman, K. Devine, D. Bozdağ, R. Heaphy, and L. Riesen, *Hypergraph-based dynamic load balancing for adaptive scientific computations*, in proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), IEEE, 2007.

[24] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

V

New Multithreaded Ordering and Coloring Algorithms for Multicore Architectures

Md. Mostofa Ali Patwary* Assefaw H. Gebremedhin**
Alex Pothen**

Abstract

We present new multithreaded vertex *ordering* and *distance-k graph coloring* algorithms that are well-suited for the emerging and rapidly growing multicore platforms. The vertex ordering techniques rely on various notions of "degree", are known to be effective in reducing the number of colors used by a *greedy* coloring algorithm, and are generic enough to be applicable to contexts other than coloring. We employ *approximate degree* computation in the ordering algorithms and *speculation* and *iteration* in the coloring algorithms as our primary remedies for breaking sequentiality and achieving effective parallelization. The algorithms have been implemented using OpenMP, and experiments run on Intel Nehalem and other multi-core machines using a set of carefully designed synthetic graphs and real-world graphs attest that the algorithms provide scalable runtime performance. The number of colors the algorithms use is nearly the same as in the serial case, which in turn is often very close to optimal.

1 Introduction

Graph algorithms in general are challenging to parallelize, when high performance and good scalability are primary design goals. Among the reasons causing the difficulty are: low available concurrency, poor data locality, irregular memory access pattern, and high data access to computation ratio. For these reasons graph algorithms are rather ill-suited for distributed memory machines. Shared-memory, multi-core architectures supporting *multithreading* provide a better environment for performance and ease of programming.

In this paper, we present new multithreaded vertex *ordering* and *distance-k coloring* algorithms that are well-suited for the emerging and rapidly growing

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, Mostofa.Patwary@ii.uib.no

**Purdue University, West Lafayette, IN, USA, {agebreme,apothen}@purdue.edu

multicore platforms. Distance-1 coloring is used (among many others) in parallel scientific computing to discover tasks that can be carried out concurrently or data elements that can be updated concurrently [7, 8]. Distance-2 coloring is an archetypal model used in the efficient computation of sparse Jacobian and Hessian matrices using Automatic Differentiation [4]. To solve the coloring problems, we rely on a *greedy* algorithm that incorporates a vertex *ordering* stage. The vertex ordering techniques we consider are formulated in a manner independent of a coloring algorithm using various notions of *degrees*. They are known to be effective in reducing the number of colors used by a greedy coloring algorithm in the serial setting, and are of interest in their own right [5].

The ordering and coloring algorithms we consider are extremely challenging to parallelize as the computation involved in both is inherently sequential. We employ *approximate degree* computation in the ordering algorithms and *speculation* and *iteration* in the coloring algorithm as our primary ingredients for overcoming the hurdle and achieving good parallel performance. Additional techniques we explore to enhance concurrency include *randomization* and various conflict-reducing *color choice* strategies. The algorithms have been implemented using OpenMP. Experiments we have run on Intel Nehalem and other multi-core machines using a set of carefully designed synthetic graphs as well as real-world graphs drawn from various application areas attest that the algorithms provide scalable runtime performance. The number of colors the algorithms use is nearly the same as in the serial case, which in turn is often very close to optimal.

Preliminaries, Related Work, and Organization

A distance-k coloring of a graph $G = (V, E)$ is an assignment of positive integers, called *colors*, to vertices such that any two vertices connected by a path consisting of at most k edges receive different colors. Thus, in a distance-1 coloring, every pair of adjacent vertices receives two different colors, and in a distance-2 coloring, every path on three vertices uses three different colors. The objective in the distance-k coloring problem is to minimize the number of colors used, and the problem is known to be NP-hard for every fixed integer $k \geq 1$ [4].

Previous work has shown that a *greedy* coloring algorithm—an algorithm that visits vertices sequentially in some *order* in each step assigning a vertex the *smallest* permissible color—is quite effective in practice. The order in which vertices are processed determines the number of colors used by the algorithm. In an earlier work [5], we identified three ordering techniques, called *Smallest Last* (SL), *Dynamic Largest First* (DLF), and *Incidence Degree* (ID) that are particularly effective in reducing the number of colors used by a greedy coloring algorithm and are generic enough to be useful in other contexts. In particular, the three ordering techniques are characterized (as shown in [5]) purely in terms of *relative* vertex degrees, in a manner decoupled from the coloring algorithm that could use them. This makes the ordering techniques of interest in their own right. Examples of non-coloring applications in which, for instance, SL ordering is useful are discussed in [4, 5]. In this paper, we present algorithms—which are the first to the best of our knowledge—for *parallelizing* these ordering techniques on multithreaded, shared-memory architectures. The algorithms are discussed in Sect. 2.

For a given vertex ordering, a greedy algorithm for distance-k coloring can be implemented such that its runtime is $O(|V| \cdot \overline{d}_k)$, where d_k denotes the average degree-k, the number of distinct paths of length at most k edges leaving a vertex [4, 6]. Note that, in the distance-1 coloring case, $\overline{d}_1$ is simply the average number of adjacent vertices a vertex has in the graph, and the complexity thus reduces to $(O|E|)$.

Using *speculation* and *iteration* as basic ingredients, a framework for effective parallelization of greedy distance-1 coloring on *distributed-memory* architectures was developed in [1]. The framework was extended to distance-2 coloring and related problems in [2]. Recently, a multithreaded algorithm derived from the framework in [1] and tailored for *shared-memory* architectures has been developed for the distance-1 coloring problem [9]. We present in this paper a similar algorithm for distance-2 coloring that additionally employs a number of new ingredients. The algorithm—along with its variations—is described in Sect. 3. In Sect. 4 we present experimental results on the parallel ordering and coloring algorithms.

2 Vertex Ordering

2.1 The Serial Framework

We give in Algorithm 1 a succinct summary of a *template* for the ordering techniques SL, DLF and ID in the serial setting. Table 1 shows how the template is *specialized* in the three cases. The key idea in the definition (and computation) of these orderings is the use of a *dynamically* changing quantity, the *back* or *forward degree* of vertices. The back degree of a vertex v is the number of vertices that are adjacent to v in G and appear *before* v in the ordering, and the forward degree of v is the number of vertices that are adjacent to v in G and appear *after* v in the ordering. The dynamic degree (back or forward) of a vertex v is denoted by $d(v)$ in Algorithm 1, while the *static* degree of the vertex in the input graph G is denoted by $d(v, G)$.

To arrive at an efficient implementation, a two-dimensional array B is used in Algorithm 1 to maintain vertices that are not yet ordered in *bins* according to their dynamic degrees. Specifically $B[j]$ stores a set of vertices where each member vertex u has a current dynamic degree $d(u)$ equal to j. The output of Algorithm 1 is given by the ordered list W of the vertices where $W[i]$ stores

Algorithm 1 Template for serial ordering (SL, DLF, ID). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V. B is a two-dimensional array used for maintaining unordered vertices binned according to their “degrees”.

```
 1: procedure OrderingTemplate($G = (V, E)$)
 2:     for each vertex $v \in V$ do
 3:         init $d(v)$
 4:         $B[d(v)] \leftarrow B[d(v)] \cup \{v\}$
 5:     init $i$                ▷ $i$ is position in $W$ where next vertex in the order is placed
 6:     while check $i$ do                           ▷ there remain vertices to order
 7:         locate $j^*$, an appropriate extreme index $j$ where $B[j]$ is non-empty
 8:         Let $v$ be a vertex drawn from $B[j^*]$
 9:         $B[j^*] \leftarrow B[j^*] \setminus \{v\}$
10:         for each vertex $w \in adj(v)$ do
11:             $B[d(w)] \leftarrow B[d(w)] \setminus \{w\}$
12:             update $d(w)$
13:             $B[d(w)] \leftarrow B[d(w)] \cup \{w\}$
14:         $W[i] \leftarrow v$
15:         update $i$
```

Table 1: Table accompanying the ordering template in Algorithm 1

	SL	DLF	ID
L 3: `init` $d(v)$	$d(v) \leftarrow d(v, G)$	$d(v) \leftarrow d(v, G)$	$d(v) \leftarrow 0$
L 5: `init` i	$i \leftarrow \|V\| - 1$	$i \leftarrow 0$	$i \leftarrow 0$
L 6 `check` i	$i \geq 0$	$i \leq \|V\| - 1$	$i \leq \|V\| - 1$
L 7: `locate` j^*	$j^* = \min_j\{B[j] \neq \emptyset\}$	$j^* = \max_j\{B[j] \neq \emptyset\}$	$j^* = \max_j\{B[j] \neq \emptyset\}$
L 12: `update` $d(w)$	$d(w) \leftarrow d(w) - 1$	$d(w) \leftarrow d(w) - 1$	$d(w) \leftarrow d(w) + 1$
L 15: `update` i	$i \leftarrow i - 1$	$i \leftarrow i + 1$	$i \leftarrow i + 1$

the ith vertex in the ordering. In SL, the ordering W is computed right-to-left ($i = |V| - 1$ down to $i = 0$), whereas the ordering in DLF and ID is computed left-to-right ($i = 0$ up to $i = |V| - 1$). The ith vertex in SL ordering is a vertex with the *smallest* back degree among the vertices not yet ordered, in a DLF ordering it is a vertex with the *largest* forward degree among the vertices not yet ordered, and in an ID ordering it is a vertex with the *largest* back degree among the vertices not yet ordered. The rationale behind each of these ordering techniques in the context of a coloring algorithm is to bring vertices that are likely to be highly constrained in choice of colors early in the ordering. In Algorithm 1, once the ith vertex v in the ordering is determined (and removed from B), each vertex w adjacent to v is moved from its current bin in B to an appropriate new bin. With suitable pointer techniques the relocation can be performed in constant time [5]. Thus the work involved in the ith step of Algorithm 1 is proportional to $d(v, G)$, and the overall complexity of the algorithm is $O(|E|)$.

2.2 Parallel Ordering

We parallelized these three ordering techniques employing a common paradigm, but we restrict the presentation here to only the SL ordering case.

We developed two different approaches for the parallelization. The first approach aims at parallelizing the ordering closely maintaining the serial behavior, while the second approach settles for an approximate solution in favor of increased concurrency. In both approaches, we assume p threads are available and utilized, and we denote by $t(v)$ the thread with which the vertex v is initially associated.

Algorithm 2 A parallel SL ordering algorithm using p threads (the Regular variant). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V. The array B is as in Algorithm 1, and the arrays B_t, R_t, and A_t are thread-private arrays; the latter two are used to remove or add vertices from or into the global array B.

```
procedure SmallestLastOrdering-Regular($G = (V, E)$)
    for each vertex $v \in V$ in parallel do
        $d(v) \leftarrow d(v, G)$
        $B_{t(v)}[d(v)] \leftarrow B_{t(v)}[d(v)] \cup \{v\}$
    for each bin $j \in \{\delta(G), \ldots, \Delta(G)\}$ in parallel do
        for $k = 1$ to $p$ do
            for each vertex $v \in B_k[j]$ do
                $B[j] \leftarrow B[j] \cup \{v\}$                ▷ note that $j = d(v)$
    $i \leftarrow |V|$
    while $i \geq 0$ do
        Let $\delta$ denote the smallest index $j$ such that $B[j]$ is non-empty
        for each vertex $v \in B[\delta]$ in parallel do
            for each vertex $w \in adj(v)$ do
                if $w \notin R_{t(v)}$ then
                    $R_{t(v)}[d(w)] \leftarrow R_{t(v)}[d(w)] \cup \{w\}$
                $r(w) \leftarrow r(w) + 1$                ▷ atomic statement
            $W[i] \leftarrow v$; $i \leftarrow i - 1$                ▷ critical statements
        for each bin $j \in \{\delta, \ldots, \Delta(G)\}$ in parallel do
            for $k = 1$ to $p$ do
                for each vertex $v \in R_k[j]$ do
                    if $r(v) > 0$ then
                        $B[j] \leftarrow B[j] \setminus \{v\}$                ▷ note that $j = d(v)$
                        $d(v) \leftarrow d(v) - r(v)$; $r(v) \leftarrow 0$
                        $A_{t(v)}[d(v)] \leftarrow A_{t(v)}[d(v)] \cup \{v\}$
        for each bin $j \in \{\delta, \ldots, \Delta(G)\}$ in parallel do
            for $k = 1$ to $p$ do
                for each vertex $v \in A_k[j]$ do
                    $B[j] \leftarrow B[j] \cup \{v\}$                ▷ note that $j = d(v)$
```

2.2.1 The First Approach—Regular

Algorithm 2 outlines the first approach. The first task Algorithm 2 parallelizes is the population of the global bin array B. To achieve this, with each thread T_k, $1 \leq k \leq p$, a *local* two-dimensional array B_k is associated. The p local arrays are first populated in parallel (the for-loop in Lines 2–4). Then, the contents are gathered into the global array B, where the parallelization is now switched to run over bins, as shown in the for-loop in Lines 5–8. There and elsewhere in this paper, $\delta(G)$ and $\Delta(G)$ denote the minimum and maximum degree in G, respectively.

The remainder of Algorithm 2 mimics the serial algorithm (Algorithm 1). In the serial algorithm, in each step of the while loop, a *single* vertex—a vertex with the *smallest* current dynamic degree δ—is ordered and its neighbors' locations updated in B. However, the bin $B[\delta]$ could contain *multiple* vertices. Algorithm 2 takes advantage of this opportunity and strives to order such vertices and update their neighborhoods in parallel. There are a few potential problems associated with such an attempt.

- **Problem**: A pair of vertices u and v in $B[\delta]$ are adjacent to each other. In such a case, a thread processing one of the vertices, say u, could try to move the vertex v to another bin while another thread at the same time attempts to order v, making the result inconsistent.
  Solution: while ordering the vertex u, we avoid updating the location of the vertex v in B, and instead order v as well in the current step.

- **Problem**: Removal of multiple vertices from the same bin, say $B[j]$. Suppose two vertices u and v from $B[\delta]$ have a common neighbor w in $B[j]$. In the serial case, u and v would be ordered one after another, $d(w)$ would be reduced by 2, and w would be relocated twice. In the parallel case, two threads might try to remove w from $B[j]$ at the same time and the removal of w in constant time will make $B[j]$ inconsistent. Similarly, suppose two vertices u and v in $B[\delta]$ have respective neighbors w and x such that $d(w) = d(x) = j$. In the parallel case, two threads might try to remove w and x from $B[j]$ at the same time while processing u and v in parallel and and the removals of w and x in constant time will also make $B[j]$ inconsistent.
  Solution: We let each thread T_k, $1 \leq k \leq p$, maintain its own two-dimensional *removal* array R_k, where it stores vertices to be removed from B while the parallel ordering of $B[\delta]$ happens (see the for loop in Lines 13–16). The removal from B takes place once the ordering of vertices in $B[\delta]$ is completed. Since for any two bins $B[j]$ and $B[j']$ the removal from $B[j]$ is independent of the removal from $B[j']$, these could be done in parallel, as shown in Lines 18–24.

Algorithm 3 A parallel SL ordering algorithm on p threads (the RELAXED variant). Input: graph $G = (V, E)$. Output: An ordered list W of the vertices in V.

```
procedure SMALLESTLASTORDERING-RELAXED($G = (V, E)$)
    for each vertex $v \in V$ in parallel do
        $d(v) \leftarrow d(v, G)$
        $B_{t(v)}\left[d\left(v\right)\right] \leftarrow B_{t(v)}\left[d\left(v\right)\right] \cup \{v\}$
    $i \leftarrow |V|$
    for $k = 1$ to $p$ in parallel do
        while $i \geq 0$ do
            Let $\delta$ be the smallest index $j$ such that $B_k\left[j\right]$ is non-empty
            Let $v$ be a vertex drawn from $B_k\left[\delta\right]$
            $B_k\left[\delta\right] \leftarrow B_k\left[\delta\right] \setminus \{v\}$
            for each vertex $w \in adj\left(v\right)$ do
                if $w \in B_k$ then
                    $B_k\left[d\left(w\right)\right] \leftarrow B_k\left[d\left(w\right)\right] \setminus \{w\}$
                    $d\left(w\right) \leftarrow d\left(w\right) - 1$
                    $B_k\left[d\left(w\right)\right] \leftarrow B_k\left[d\left(w\right)\right] \cup \{w\}$
            $W\left[i\right] \leftarrow v$; $i \leftarrow i - 1$                    ▷ critical statements
```

- **Problem**: Addition of multiple vertices to the same bin, say $B[j]$.
  Solution: we address this concern by using a similar technique as in the second bullet item. We let each thread maintain its own two-dimensional *addition* array A_k. Again, the addition of vertices to different bins in B can be done in parallel, as shown in Lines 25–28.

2.2.2 The Second Approach—Relaxed

Our second approach for parallelizing the SL ordering algorithm abandons the use of the global array B altogether, and works only with the local arrays B_k associated with each thread T_k. In updating locations of neighbors of a vertex, a thread T_k checks whether or not the vertex w desired to be relocated is in the thread's local array B_k. If w is indeed in B_k it is relocated by the same thread, if not, it is simply ignored. In this manner, only *approximate* dynamic degrees are used while computing the ordering. The approach is formalized in Algorithm 3.

3 Parallel Distance-2 Coloring

Algorithm 4 outlines the parallel distance-2 coloring we developed in this work. The algorithm has two phases, both of which are performed in parallel, and runs in an iterative fashion. In the first phase of each round of the iteration, threads concurrently color their respective vertices in a *speculative* manner (paying atten-

Algorithm 4 An iterative parallel algorithm for distance-2 coloring using p threads. Input: graph $G = (V, E)$. Output: a vertex-indexed array *color*[] indicating colors of vertices. The vertex set V is assumed to be *ordered*.

```
 1: procedure IterativeD2Coloring($G = (V, E)$)
 2:     $U \leftarrow V$
 3:     while $U \neq \emptyset$ do
 4:         for each vertex $v \in U$ in parallel do          ▷ Phase 1: tentative coloring
 5:             for each vertex $w \in adj(v)$ do
 6:                 mark $color[w]$ as forbidden to vertex $v$
 7:                 for each vertex $x \in adj(w)$ and $x \neq v$ do
 8:                     mark $color[x]$ as forbidden to vertex $v$
 9:             Pick a permissible color $c$ for vertex $v$ using some strategy
10:         $R \leftarrow \emptyset$                     ▷ $R$ denotes the set of vertices to be recolored
11:         for each vertex $v \in U$ in parallel do          ▷ Phase 2: conflict detection
12:             $cont \leftarrow true$
13:             for each vertex $w \in adj(v)$ and $cont = true$ do
14:                 if $color[v] = color[w]$ and $f(v) > f(w)$ then
15:                     $R \leftarrow R \cup \{v\}$; break
16:                 for each vertex $x \in adj(w)$ and $v \neq x$ do
17:                     if $color[v] = color[x]$ and $f(v) > f(x)$ then
18:                         $R \leftarrow R \cup \{v\}$; $cont \leftarrow false$; break
19:         $U \leftarrow R$
```

tion to already available color information). In this phase, two vertices that are distance-2 neighbors with each other and are handled by two different threads may be colored concurrently and receive the same color, causing a *conflict*. In the second phase, threads concurrently check the validity of colors assigned to their respective vertices in the current round and identify a set of vertices that needs to be re-colored in the next round to resolve any detected conflicts. The algorithm terminates when every vertex has been colored correctly.

Although the two phases (tentative coloring and conflict detection) in each round iterate over the same set U of vertices, the runtime of the second phase is likely to be significantly lower than the first. This is because of the **break** statements used in the conflict detection phase, where the search for conflict in the distance-2 neighborhood of the current vertex v is stopped immediately the moment a conflict impacting v is discovered—note that the *cont* boolean variable in Line 12 is used to break out of the for-loop in Line 13 due to a condition in the for-loop in Line 16. Due to the use of the early breaks, we observed that the second phase typically takes roughly around 25% of the overall runtime of the algorithm; without the breaks second phase would have taken the same time as the first.

As written, Algorithm 4 is a *template* and can be *specialized* in a number of ways depending on the strategies employed in *selecting a permissible color* in Line 9 and *selecting a vertex to re-color* in Lines 14 and 17 in the event of a conflict.

For the color choice in Line 9 we investigated four alternatives: (i) First Fit (FF), where each thread searches for a permissible color for the vertex v starting from *color 1*; (ii) Staggered First Fit (SFF), where each thread searches for a permissible color for the vertex v staring from a location *staggered* according to the thread ID (and rolls back to another interval as needed); (iii) Least Used (LU), where each thread choses the *least used* permissible color for v; and (iv) Random (R), where each thread *randomly* chooses a color for v among all permissible colors for v. Compared to FF, the strategies SFF, LU, and R reduce the likelihood of conflicts, at the expense of increasing the number of colors used by the algorithm. Experiments we conducted showed that FF offered a better trade-off and would be used in the results we report in Sect. 4.

In the event of a conflict (two vertices u and v received the same color while being distance-2 neighbors), it suffices to re-color one of the vertices to resolve the conflict. We investigated two alternative strategies (the function f in Lines 14 and 17) in choosing the vertex to recolor. In the first strategy, we let $f(u) = u$ (the id of the vertex u), in the second, we let $f(u) = rand(u)$ (a random number associated with the vertex u). Compared to a function that uses vertex IDs, a random function improves *load balance* among threads in re-coloring rounds, but comes at the expense of runtime overhead in generating the random numbers. Experiments we conducted (not reported here) indicated that vertex IDs provided better trade-off and would be used in the results we report in Sect. 4.

4 Experimental Results

We present in this section results on experiments performed on an Intel Nehalem machine equipped with Intel(R) Core(TM) i7 CPU 860 processors running at 2.8GHz. The system has 4 cores with 2 threads on each. The total memory size is 16 GB, with 4 × 32 KB Instruction and 4 × 32 KB Data Level-1 cache, 4×256 KB Level-2 cache, and 8 MB shared Level-3 cache. The operating system is GNU/Linux.

Our testbed consists of 20 graphs, 5 of which are real-world graphs drawn from various *scientific computing* (sc) applications and are downloaded from the University of Florida Sparse Matrix Collection, and the remaining 15 are synthetically generated using the R-MAT algorithm [3]. By combining the four input parameters of the R-MAT algorithm in various ways (the sum of the parameters needs to be equal to one), it is possible to generate graphs with varying properties. We generated three types of R-MAT graphs: (i) *Erdös-Renyi random* (er) graphs, using the set of parameters $(0.25, 0.25, 0.25, 0.25)$; (ii) *small-world type*

Table 2: Structural properties of the various graphs in the testbed: scientific computing (sc), rmat-random (er), rmat-good (g), and rmat-bad (b).

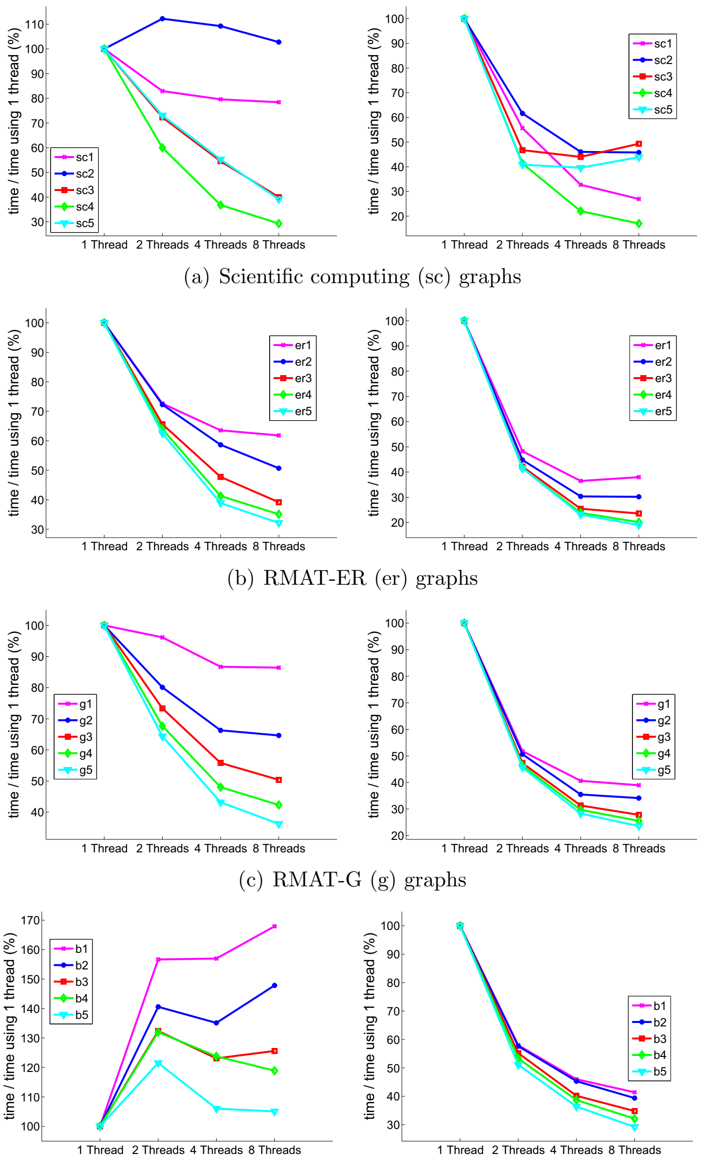
Name	$\lvert V\rvert$	$\lvert E\rvert$	Δ	Name	$\lvert V\rvert$	$\lvert E\rvert$	Δ
sc1 (bone010)	986,703	35,339,811	80	g1	262,144	2,093,552	558
sc2 (af_shell10)	1,508,065	25,582,130	34	g2	524,288	4,190,376	618
sc3 (nlpkkt120)	3,542,400	46,651,696	27	g3	1,048,576	8,382,821	802
sc4 (er1)	16,777,216	134,217,651	138	g4	2,097,152	16,767,728	1,069
sc5 (nlpkkt160)	8,345,600	110,586,256	27	g5	4,194,304	33,541,979	1,251
er1	262,144	2,097,104	98	b1	262,144	2,067,860	4,493
er2	524,288	4,194,254	94	b2	524,288	4,153,043	6,342
er3	1,048,576	8,388,540	97	b3	1,048,576	8,318,004	9,453
er4	2,097,152	16,777,139	102	b4	2,097,152	16,645,183	14,066
er5	4,194,304	33,554,349	109	b5	4,194,304	33,340,584	20,607

Table 3: Runtime in seconds while using one thread. OT shows ordering time, and CT shows distance-2 coloring time (Algorithm 4).

	SL-Relaxed		SL-Regular			SL-Relaxed		SL-Regular	
Name	OT	CT	OT	CT	Name	OT	CT	OT	CT
sc1	1.18	30.45	1.73	31.18	g1	0.18	3.82	0.32	3.84
sc2	0.87	10.13	0.91	10.25	g2	0.42	8.86	0.74	9.03
sc3	1.64	16.45	6.39	28.89	g3	1.07	23.25	1.74	23.69
sc4	31.19	452.76	51.68	479.81	g4	2.54	61.98	4.18	65.64
sc5	4.29	39.86	17.68	73.91	g5	6.01	168.64	9.59	171.84
er1	0.18	2.13	0.32	2.21	b1	0.16	12.68	0.44	12.63
er2	0.45	5.02	0.71	5.23	b2	0.37	32.11	0.95	32.36
er3	1.20	12.75	1.84	13.54	b3	0.87	94.30	2.09	95.60
er4	2.86	33.62	4.54	36.07	b4	2.00	280.00	4.48	281.39
er5	6.43	83.74	10.51	88.77	b5	4.85	785.80	9.87	797.86

1 (g) graphs, using the set of parameters $(0.45, 0.15, 0.15, 0.25)$; and (iii) *small-world type 2* (b) graphs, using the set of parameters $(0.55, 0.15, 0.15, 0.15)$. These three R-MAT generated graph types vary widely in terms of *degree distribution* of vertices and *density of local subgraphs* and represent a wide spectrum of input types for the ordering and coloring algorithms. The er graphs have *normal* degree distribution, whereas the g (for "good") and b (for "bad") graphs contain many dense local subgraphs. The latter differ primarily in the magnitude of maximum vertex degree they contain, the bad graphs have much larger maximum degree. Table 2 provides structural information on all twenty test graphs.

Figure 1 shows scalability results on the two parallel Smallest Last ordering algorithms, SL-Regular (Algorithm 2) and SL-Relaxed (Algorithm 3). The plots show runtimes for various numbers of threads normalized by the runtime when 1 thread is used; the raw numbers for the 1 thread case are listed in Table 3. Clearly, the algorithm SL-Regular scaled poorly especially for the sc and rmat-b graphs,



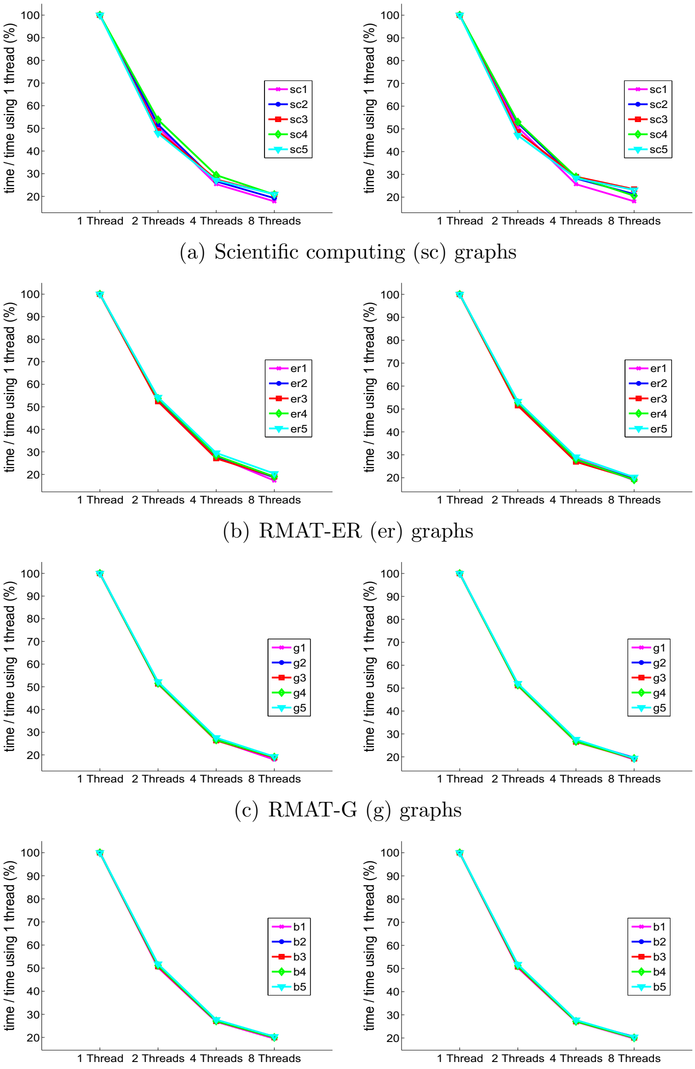
(a) Scientific computing (sc) graphs

(b) RMAT-ER (er) graphs

(c) RMAT-G (g) graphs

(d) RMAT-B (b) graphs

Figure 1: Scalability results on the two parallel SL ordering algorithms. Left column: Algorithm 2 (SL-Regular). Right column: Algorithm 3 (SL-Relaxed). The plots show runtimes normalized by the runtime when 1 thread is used; the raw numbers for the case of 1 thread are listed in Table 3.



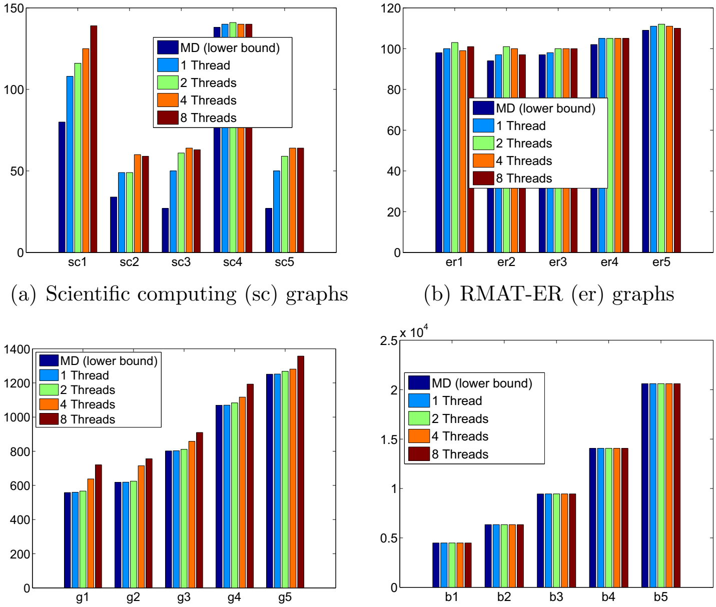
(a) Scientific computing (sc) graphs

(b) RMAT-ER (er) graphs

(c) RMAT-G (g) graphs

(d) RMAT-B (b) graphs

Figure 2: Scalability results on the parallel distance-2 coloring algorithm (Algorithm 4) while employing the parallel ordering algorithm SL-Relaxed (Algorithm 3). Left column: only distance-2 coloring time. Right column: ordering plus distance-2 coloring time. The plots show runtimes normalized by the runtime when 1 thread is used; the raw numbers for the case of 1 thread are listed in Table 3.



(a) Scientific computing (sc) graphs

(b) RMAT-ER (er) graphs

(c) RMAT-G (g) graphs

(d) RMAT-B (b) graphs

Figure 3: Number of colors used by the parallel distance-2 coloring algorithm (Algorithm 4) while employing the SL-Relaxed ordering algorithm (Algorithm 3). MD, the maximum degree in a graph, is a lower bound on the optimal number of colors needed.

whereas SL-Relaxed scaled well across all the graph types tested. We therefore present further results using the better performing algorithm SL-Relaxed.

Figure 2 shows scalability results for the parallel distance-2 coloring algorithm (Algorithm 4) while using the SL-Relaxed algorithm for parallel ordering. The left column shows runtime results considering *only* the coloring stage, whereas the right column shows results on *total* (ordering plus coloring) time. Since distance-2 coloring takes substantially more time than the ordering, the scalability behavior of just the coloring stage is nearly identical to that of the overall execution. It can be seen that the coloring algorithm (including the ordering stage) scaled well across all the graphs in the testbed.

Figure 3 shows the number of colors the parallel distance-2 coloring algorithm (Algorithm 4) used while employing the SL-Relaxed ordering algorithm. In each subfigure, a bar corresponding to the maximum degree (MD) in a graph, which is a lower bound on the optimal number of colors needed to distance-2 color a graph, is included. It can be seen that the number of colors the parallel algorithm used remained reasonably constant as the number of threads is increased. Further, it can be seen that the number in each case is either optimal or very close to optimal.

Because of its “closeness” to the serial SL ordering algorithm, the parallel algorithm SL-Regular is expected to use fewer colors than SL-Relaxed. We observed this to be the case in the experiments we conducted.

5 Conclusion

We presented new parallel ordering and coloring algorithms and demonstrated scalable performance on a multicore machine supporting a modest number of threads. The techniques used for computing the ordering and coloring in parallel are applicable to other problems where there is an inherent serial ordering to the computations that needs to be relaxed for increasing concurrency. In future work, we plan to conduct studies on larger machines and present more empirical results.

Acknowledgements This research was supported by the U.S. Department of Energy through the CSCPAES Institute grant DE-FC02-08ER25864 and by the National Science Foundation through grant CCF-0830645.

References

[1] D. Bozdağ, A. H. Gebremedhin, F. Manne, E. Boman, and Ümit V. Çatalyürek, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.

[2] D. Bozdağ, Ümit V. Çatalyürek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, *Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation*, SIAM Journal on Scientific Computing, 32 (2010), pp. 2418–2446.

[3] D. Chakrabarti and C. Faloutsos, *Graph mining: Laws, generators, and algorithms*, ACM Computing Surveys, 38 (2006), p. 2.

[4] A. H. Gebremedhin, F. Manne, and A. Pothen, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Review, 47 (2005), pp. 629–705.

[5] A. H. Gebremedhin, D. Nguyen, A. Pothen, and M. M. A. Patwary, *ColPack: Graph coloring software for derivative computation and beyond*, Submitted to ACM Transactions on Mathematical Software, (October, 2010).

[6] A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen, *New acyclic and star coloring algorithms with applications to computing Hessians*, SIAM Journal on Scientific Computing, 29 (2007), pp. 1042–1072.

[7] M. Jones and P. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Computing, 20 (1994), pp. 753–773.

[8] Y. Saad, *ILUM: A multi-elimination ILU preconditioner for general sparse matrices*, SIAM Journal on Scientific Computing, 17 (1996), pp. 830–847.

[9] Ümit V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, *Multithreaded algorithms for graph coloring.* Submitted for journal publication, 2011.

Errata

Errata

- Introduction

  1. Page 2: “Such packages. . . runs” should be “Such packages. . . run”.

- Paper II

  1. Page 1: “in image processing” should be “image processing” and “. . . graphs, and is. . . ” should be “. . . graphs. It is. . . ”.
  2. Page 4, Algorithm 1: Title should be “The sequential Union-Find algorithm for computing connected components of a graph”.
  3. Page 4: “therefor” should be “therefore”.
  4. Page 10: “. . . algorithms outperforms. . . ” should be “. . . algorithms outperform. . . ” and “algorithms and” should be “algorithms, and”.
  5. Page 11, Bibitem [4]: “D. J. Bader” should “D. A. Bader”.

- Paper III

  1. Page 6, Table 1: The extra parenthesis in the “Call” column for “criticality” should be removed.
  2. Page 19, Bibitem [9]: Volume number 0 should be removed.

- Paper IV

  1. Page 7: “algorithms,and” should be “algorithms, and”.
  2. Page 17, Lemma 5.3: “. . . smaller or . . . ” should be “. . . smaller than or . . . ”.

- Paper V

  1. Page 3: “$(O|E|)$” should be “$O(|E|)$”.