University of Bergen

Department of Informatics

Algorithms

Parallel algorithms for matching under preference

Student:
Håkon Heggernes Lerring

Supervisor:
Professor Fredrik Manne

Master Thesis
June 2017

Acknowledgement

First of all: I would like to thank Fredrik Manne for supervising me. I could not have done this without your help. I also want to thank Anna Maria Eilertsen for good support and relentless positive attitude

Contents

Chapter 1

Introduction

Every year hundreds of thousands of students express their preferences for higher education in Norway. Each hope-to-be student lists a couple of schools and programmes they want to be accepted to. The student orders the schools in the order of most preferred to least preferred, and submits them to The Norwegian Universities and Colleges Admission Service (NUCAS). NUCAS is then tasked with the job of deciding who gets to study where. While the task might seem daunting, it is not without hope. The problem college admissions has been studied since 1962 when Gale and Shapley introduced a solution to this problem: "College Admissions and the Stability of Marriage"[2]. With their solution we end up with a set of accepted applications and an important promise of stability: if a student is accepted to a less-than-preferred college he will not get accepted to a more preferred school by talking to them directly. Likewise for the school, if they only get students with bad grades they will not be able to admit a better student by talking to him/her directly.

Solving this problem for large amounts of students and colleges take time, and as we all know: time is money. The question we want to answer is then: can we solve problems like the college admissions problem faster by utilising multiple processors of modern CPUs?

Previous attempts at this have been less successful [7], but recent developments in the closely related field of weighted maximum matching suggest a new way [6][9]. In this thesis we use these ideas to present new parallel algorithms for the stable marriage problem, the b-Marriage problem and the Popular Matching problem.

Parts of this thesis is published as the paper "On Stable Marriages And Greedy Matchings"[8] to the SIAM Workshop on Combinatorial Scientific Computing (CSC16) and is provided in appendix A.1. The paper received the Best Paper Award.

Chapter 2

Background

2.1 Parallelisation

To understand the work we have done it is useful to have some basic understanding of how multi threading works on modern operating systems and some of the tools we used to develop our parallel algorithms. We assume basic knowledge of programming and a basic intuition of how programs compile to a sequential list of instructions (a program).

In regular sequential applications there is usually a single execution path through the program. Running the program, the operating system spawns a process, allocates some time on a processor to the process and executes it. To ensure that other process can interfer with the execution of the program, the memory allocated is usually kept private. Modern computers often feature multiple processors, letting the operating system safely run multiple programs concurrently without two programs modifying each others states. Running our program on only one processor at a time limit us to the execution speed of a single processor.

One way to speed up the execution of our program is to run multiple processes, each with their own memory space and a part of the problem to solve, without any communication between the processes. As long as there are no dependencies between the parts we run in parallel, this is a valid way to speed up the execution. When the execution of the processes depend on each other, they need to communicate. When possible, inter-process communication can be expensive time-wise and also requires careful planning of how the processes communicate. One option is to let the programs share memory, which is the basis for multithreaded programming, letting each program (or thread) make decisions based on a shared view of the memory. However, this can lead to what is know as a *race condition*. A race condition is a situation where the result of some computation depends on the timing of the execution. We illustrate this in Figure 2.1 where the result of two threads executing an incrementation of a variable (`a++`), depends on how the operations are interleaved. In our example incrementing is implemented as a three-step operation: 1. read the memory location to a temporary variable 2. write back the temporary variable plus one 3. return the temporary variable.

Time ↓

Thread 1	Thread 2	a Value
tmp = a		0
	tmp' = a	0
a = tmp + 1		1
	a = tmp' + 1	1
return tmp	return tmp'	1

Figure 2.1: Interleaved incrementation of a

Each thread first fetches the value of a to a temporary variable, increments the value of the temporary variable, and then store it back to variable a. Even though we expected the final value of the variable a to be 2, the value each thread writes back is 1.

Allowing each thread to do both operations in one go without the other thread doing any simultaneous operations would give a correct result, but this requires some kind of *synchronisation* between the threads. Synchronisation is the process where multiple threads, through some kind of communcation, agree to pause for some condition to occur. Synchronisation can reduce the gain we get from running on multiple processors by doing extra work or waiting for other threads.

There are two "layers" in which we are provided tools that help us with parallelising our code. One of which is OpenMP, an API for developing parallel programs on shared-memory computers, and the other being primitives provided by the CPU for doing safe concurrent operations.

2.1.1 OpenMP

OpenMP provides an interface for splitting the execution of a program into multiple threads. This is done by prepending a block of code with the `#pragma omp parallel` preprocessing directive which, when precompiled by a compiler supporting OpenMP, makes the block of code run in parallel on as many threads as supported by the system. OpenMP also provides a set of easy-to-use functions for common tasks inside a parallel region, most notable (for our use) of which are the `#pragma omp for`, `omp_set_lock`/`omp_unset_lock`, and `omp_get_wtime`.

The `#pragma omp for` takes a well-formed (no multiple exits or unpredictable jumps) for-loop and splits it so that each thread gets a part of the iterations. It can be configured with extra directives such as how to split the data (`schedule` directive), which variables should be private to the thread (`private` directive) or how to combine the result of the iterations through some kind of reduction function (the `reduction` directive).

The `omp_set_lock` and `omp_unset_lock` functions are used for synchronisation between the threads. The functions takes a shared memory location as parameter which is used as a lock instance. Once a lock is set, it cannot be set again from any other thread without being unset first. OpenMP also has a more coarse-grained version of these locks, the `#pragma omp critical` directive which does not require the programmer to set up a lock instance, but ensures that only one thread at any given time will be in the region of code covered by the critical pragma. In our algorithms we can prevent unnecessary waiting by ensuring that only threads with overlapping memory reads/writes use the same lock instance. A region of code protected by a lock is also called a critical section.

Some of the same functionality as OpenMP provices is available through for example the pthreads library (with quite a bit of extra programming), but we chose OpenMP for its ease of use and preexisting knowledge of the API.

2.1.2 Atomic operations

An operation (or a set of operations) is atomic if it appears to execute instantaneously from other threads. Even though several threads are executing the operations, it will never interleave with other operations on the same memory. Recall our example earlier with the `a++` operation. As we saw, the operation is in reality a set of the operations (`b = a; a = b + 1; return b;`) which may be interleaved with another thread performing the same operation on a. Many modern CPUs provide an atomic instruction for this type of situation called atomic fetch and add. The ones we use in our programs are `atomic_compare_and_swap` and `atomic_fetch_and_add` (Figure 2.2.

2.1.3 Load balancing

Load balancing between a set of threads is the process of moving work from a busy thread to another less busy thread in an attempt to make the amount of

Operation	Description

atomic_fetch_and_add(*a, b)	tmp = *a;
	a = tmp + b;
	return tmp;

atomic_compare_and_swap(*a, b, c)	if(*a != b) {
	return false;
	}
	a = c;
	return true;

Figure 2.2: Relevant atomic operations

work left more even. Uneven load balancing is a situation where some thread of the program continutes to execute tasks with other threads idling. This can be because the problem instance is divided between the threads in a way that leaves some threads with more work than others. This is waste of precious CPU-time as some processors may go idle while there is still work to do. Knowing the partitioning of the problem instance beforehand can be be computationally hard and require too much time to process, and as a result load balancing is often done during runtime.

2.2 Matching

In this section we give an introduction to the basic concepts and terminology of matching. We assume the reader has some knowledge of graphs.

Given a graph $G = (V, E)$, a *matching* M is a subset of edges $M \subseteq E$ such that no edges in M share a common endpoint. A matching problem often has some condition to it. A *maximum weighted matching* M is a matching in a weighted graph (where each edge has some weight associated to it, typically an integer or real number), such that the sum of the edges in the matching is the greatest compared to all other matchings in the graph. A *maximum cardinality matching* is a matching M in G such that no other matching has a size (the number of edges) larger than M.

2.3 Stable Marriage

The stable marriage problem is defined as follows: Given a set L of n men, a set R of m women, how the men rank the women, and how the women rank the men. Can we find a matching between the men and the women such that the matching is *stable*? A matching is stable if it contains no *blocking pair*. A pair (l, r), where $l \in L$ and $r \in R$, is a blocking if it is not in the matching while both l and r prefer each other over their current match.

There are many variants to the problem, but we only look at two: the stable marriage problem (SM) and the stable marriage with incomplete lists (SMI). The difference between SM and SMI is that where in SM each man ranks each woman and vice versa, this might not be true for an instance of SMI. We call instances of these problems *complete* and *sparse* respectively.

The problem of greedy matching has been shown to reduce to an instance of *stable marriage*.[8]

2.4 Popular Matching

In the problem of maximum size popular matching we, instead of trying to find a stable matching, try to maximise the number of men and women who prefer the matching to any other possible matching. More formally we say that a matching M' is more popular than M if there is more men and women that prefer M' to M than the other way around. A matching M is popular if there are no matchings M' that is more popular than it [4]. Note that a maximum size popular matching might not be stable, but complete stable matchings (where every vertex of the graph is matched to some other vertex) are popular. This means we allow our matching to become unstable while increasing the matching size.

Chapter 3

Sequential algorithms

In this chapter we present background material necessary to understand our parallel algorithms presented in Chapter 4. While the sequential algorithms presented in this chapter is not our work, the implementations of them are.

3.1 Stable Marriage

In this section we present two existing $\mathcal{O}(N^2)$ algorithms for solving the problem: the original algorithm by Gale and Shapley, and a recursive variant by McVitie and Wilson.

3.1.1 Introduction

The problem of college admissions was first introduced by Gale and Shapley with their algorithm for solving the problem of admitting students to colleges. In [2], they describe the familiar situation where the students on one side wants to be accepted by the best (based on some personal ranking) college, while the colleges on the other side wants to admit their best (by grades or other measures) ranked students.

Gale and Shapley gave their solution (which we call the GALE-SHAPLEY algorithm) to another problem: the stable marriage problem. In the stable marriage problem each side has a quota of 1 to fill. Instead of students and colleges there are men an women ready to marry. McVitie and Wilson later expanded on the GALE-SHAPLEY algorithm [10], by giving a recursive version we call the MCVITIE-WILSON algorithm.

3.1.2 The GALE-SHAPLEY algorithm

The GALE-SHAPLEY algorithm is based on simulating the process of finding a matching, with one side proposing and the other side rejecting or accepting proposals based on their preferences. The algorithm operates in rounds. In each round it picks men from a list of unmatched men. Each of these men propose to his most preferred woman among those he has yet to propose to. Each woman then rejects all but their current best offer. The men with rejected proposals are put back on a list of unmatched men. We obtain our stable matching when, at the end of a round, all the men either have pending proposals or are out of women to propose to. In Algorithm 1 we give pseudo-code for the GALE-SHAPLEY algorithm.

In Algorithm 1 we define $ranking(r, l)$ to return r's ranking of l as a number from 1 to n with 1 being the best ranked and n being the worst. $ranking(r, NULL)$ returns $n+1$ to ensure that any partner is better than no partner. $nextCandidate(l)$ returns r's sequence of ranked partners one by one with each successive call, starting with the best ranked partner. The variable $suitor(r)$, which represents the current best proposer to r, is initially defined as $NULL$. Q is a queue that supports the operations $enqueue(l)$ to add an element from L to the queue, and $dequeue()$ to remove and return the first element from the queue. The algorithm differs slightly from the original one by putting rejected men onto a queue, reducing the needed to scan through L for each round.

Algorithm 1 The GALE-SHAPLEY algorithm

```
$Q = L$
while $Q \neq \emptyset$ do
    $u = Q.dequeue()$
    $partner = nextCandidate(u)$
    while $partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, suitor(partner))$ do
        $partner = nextCandidate(u)$
    end while
    if $partner \neq NULL$ then
        if $suitor(partner) \neq NULL$ then
            $Q.enqueue(suitor(partner))$
        end if
        $suitor(partner) = u$
    end if
end while
```

Algorithm 1 terminates when there are no more elements on the queue, at which point the set of edges $\{(suitor(r), r) \mid r \in R\}$ is our matching. In SM instances, where each man ranks each woman and vice versa, it is shown that Algorithm 1 always produce a complete stable solution.[2] This may not be true for the case of SMI instances, which will be stable but not all men and women might get a match. To support SMI instances we modify the $nextCandidate(l)$ routine to return $NULL$ after the last candidate.

3.1.3 The MCVITIE-WILSON algorithm

The algorithm proposed by McVitie and Wilson[10] is based on the GALE-SHAPLEY algorithm, together with the observation that the order in which the suitors propose does not change the set of matched vertices. [3] The algorithm starts out with an empty set of men, adding men to the solution while keeping the solution stable. The algorithm proceeds through the set of men, adding one at a time to the solution by letting him propose to his best ranked woman that he has yet to propose to. As soon as a woman has more than one proposal the solution is no longer stable. In this case the woman rejects the worst ranked man. The rejected suitor is recursively re-added to the solution again by letting him propose to his next best choice. The recursive procedure terminates when a man proposes to a woman with no previous proposals. At this point the algorithm continues with the next unmatched man. In Algorithm 2 we present a pseudo-code implementation of the MCVITIE-WILSON algorithm.

Algorithm 2 terminates when there are no more members in L to loop over and the call to $introduceSuitor$ returns, at which point the set of edges $\{(suitor(r), r) \mid r \in R\}$ is our stable matching. It is shown that the solution of the MCVITIE-WILSON algorithm produces the same result as the GALE-SHAPLEY algorithm.[10]

3.2 The b-Marriage Problem

In this section we introduce the b-marriage problem, and present an $\mathcal{O}(N^2)$ algorithm for solving it.

3.2.1 Introduction

The b-marriage problem is similar to the previously described stable marriage problem in Section 2.3. The difference being that each vertex can be specified to be matched with any number of neighbours. This is closer to the original college

Algorithm 2 The MCVITIE-WILSON algorithm

```
procedure MCVITIE-WILSON
    for each $l$ in $L$ do
        $introduceSuitor(l)$
    end for
end procedure
procedure INTRODUCESUITOR($u$)
    $partner = nextCandidate(u)$
    while $partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, suitor(partner))$ do
        $partner = nextCandidate(u)$
    end while
    if $partner \neq NULL$ then
        $rejected = suitor(partner)$
        $suitor(partner) = u$
        if $rejected \neq NULL$ then
            $introduceSuitor(rejected)$
        end if
    end if
end procedure
```

admissions problem where one college can be assigned multiple students. For the college admissions problem, b is the number of students the college can admit. For the students, on the other hand, b is fixed to 1 as one student should not get admitted to multiple colleges simultaneously. In this case where only one side of the problem has a b value greater than 1 this can be solved with an algorithm for the stable-marriage problem by duplicating the vertices for colleges b times (and any incident edges). If both sides has a $b > 1$ then the copying technique, applied to both sides can give a situation where one student is admitted to the same college multiple times. Instead of dealing with colleges and students, we continue with men and women. Imagining a situation where we have to find a set of polygamous stable marriages. Each man and woman can have individual "quotas" on how many of the other sex he/she wants to be matched with. In our experiments the quotas for the men and women will be uniform, but individual quotas does not change the problem. We denote the number of wives for man l as $b_m(l)$ and the number of husbands for woman r as $b_w(r)$. We only look at the case where both $b_m(l)$ and $b_w(r)$ is > 1 $\forall(l, r) \in LXR$.

This algorithm is based on MCVITIE-WILSON but draws inspiration from the B-SUITOR [9] algorithm, which is an approximation algorithm for weighted b-matching.

It has been shown that the problem of greedy weighted b-matching, a more general matching problem on weighted graphs, can be reduced to the b-marriage problem [A.1].

3.2.2 Algorithm

In both the GALE-SHAPLEY (Section 3.1.2) and the MCVITIE-WILSON algorithms (Section 3.1.3) a woman with more than one proposal rejects the worst ranked. The algorithm for solving b-marriage is similar to these algorithms, but allows a woman w to have up to $b_w(w)$ proposals before she has to reject one. This is done by introducing a variable $pq(w)$ which returns a max-priority-queue (based on the ranks of w) capable of storing the $b_w(r) + 1$ suitors for woman w. Initially all priority queues are empty. When a man proposes to a woman w the proposal is added to the queue $p(w)$. If the size of w's queue $|pq(w)| \leq b_w(w)$ the worst ranked suitor is rejected (by a dequeue operation) and is recursively added to the solution, trying his next candidate. Observe that matching a man to b_m women is the same as matching b_m copies of the man with each their woman, as long as two copies of the same man is not matched to the same woman. We use this observation in the algorithm and introduce each man b_m times to the solution. Because the copies of a man share the $nextCandidate(l)$ pointer, which is increasing with each introduction, we avoid matching two copies of a man to the same woman.

The algorithm terminates when all the copies of the men has been introduced to the solution and the recursive calls have returned. At this point the contents of the queues holds the edges of the stable matching. Each woman w is matched is matched to each suitor in her queue $pq(w)$.

In Algorithm 3 we give a pseudo-code implementation of this algorithm.

3.3 Maximum Size Popular Matching

In this section we present an algorithm by Huang and Kavitha[4] with an $\mathcal{O}(|E| * |N_0|)$ runtime, where $N_0 = min(L, R)$ and $|E|$ is the total number of rankings.

3.3.1 The HUANG-KAVITHA algorithm

Recall the introduction to the problem in Section 2.4

Algorithm 3 The b-marriage algorithm

```
procedure B-MARRIAGE
    for each $l$ in $L$ do
        for each $c$ in $c \mid c > 0, c < b_m(l)$ do
            $introduceSuitor(l)$
        end for
    end for
end procedure
procedure INTRODUCESUITOR($u$)
    $partner = nextCandidate(u)$
    while ($partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, \lfloor pq(partner).peek() \rfloor)$) do
        $partner = nextCandidate(u)$
    end while
    if $partner \neq NULL$ then
        $pq(partner).enqueue(u)$
        if $|pq(partner)| > b_w(partner)$ then
            $introduceSuitor(pq(partner).dequeue())$
        end if
    end if
end procedure
```

The HUANG-KAVITHA algorithm works by first computing a stable matching (we can find this with an algorithm solving the stable marriage problem, e.g. the MCVITIE-WILSON algorithm). Any men rejected by all his ranked women is immediately "lifted" to a higher level and re-introduced to the solution. Men at the higher level is always preferred by women to men at the lower level. If a man at the higher level is rejected by all his ranked women he will remain unmatched. The algorithm terminates when all men have either been matched or been by his ranked women at the higher level. The matching returned by the MCVITIE-WILSON-algorithm is then the maximum size popular matching.

We base our algorithm on the MCVITIE-WILSON algorithm presented in Section 3.1.3 and present the pseudocode in Algorithm 4.

In Algorithm 4 we introduce the variable $level(l)$ which stores the current level of a man l. $level(l)$ is initialised to 0 for every man. We also introduce the operation $resetNextCandidate(l)$ which resets the sequence of candidates returned by $nextCandidate(l)$ back to the beginning of the sequence. The $peek()$ operation on a queue returns the first element from the queue without removing it.

Algorithm 4 The huang-kavitha algorithm

```
procedure HUANG-KAVITHA
    for each $l$ in $L$ do
        $introduceSuitor(l)$
    end for
end procedure
procedure INTRODUCESUITOR($u$)
    repeat
        $partner = nextCandidate(u)$
        $level_u = level(u)$
        if $partner \neq NULL$ then
            $level_p = level(suitor(partner))$
            $rank_u = ranking(partner, u)$
            $rank_p = ranking(partner, suitor(partner))$
        end if
    until $partner == NULL \vee$
    $level_u > level_p \ \vee \ (level_u == level_p \ \wedge \ rank_u < rank_p)$
    if $partner \neq NULL$ then
        $rejected = suitor(partner)$
        $suitor(partner) = u$
        if $rejected \neq NULL$ then
            $introduceSuitor(rejected)$
        end if
    else
        $level(u) = level(u) + 1$
        $resetNextCandidate(l)$
        $introduceSuitor(u)$
    end if
end procedure
```

Chapter 4

Parallel algorithms

In this chapter we introduce our main contributions: a set of parallel algorithms for solving the stable marriage problem, the b-marriage problem, and the popular matching problem.

4.1 Parallel Stable Marriage

In this section we present two parallel algorithms for solving the stable marriage problem: a parallel version of the GALE-SHAPLEY algorithm presented in Algorithm 1, and a parallel version of the MCVITIE-WILSON algorithm presented in Algorithm 2.

There are previous parallel algorithms solving the stable marriage problem. One algorithm by Hull [5] models the men and woman as agents, with the men-agents sending proposal-messages to the women and the woman-agents accepting and rejecting based on their preferences. Tseng and Lee proposed [12] an algorithm based on the divide-and-conquer approach where partial solutions are solved for subsets of the men and merged together by solving the cases of conflict (two men in different subsets can be matched with the same woman). In this approach each subset can be solved in parallel without any synchronisation, however the merging would require synchronisation to be done in parallel. Larsen [7] presented two parallel algorithms for a distributed memory computer. One of the GALE-SHAPLEY algorithm and one of the algorithm by Tseng and Lee.

More recent advances in parallel algorithms for the weighted maximum matching problem like the SUITOR[9] algorithm by Manne and Bisseling, bears strong resemblance to the MCVITIE-WILSON and GALE-SHAPLEY algorithms when applied to general edge weight graphs [A.1]. This leads us to believe we can use some of the same techniques for parallelising the MCVITIE-WILSON and GALE-SHAPLEY algorithms as was used for the parallel SUITOR algorithm.

4.1.1 A parallel GALE-SHAPLEY algorithm

We now present our parallel GALE-SHAPLEY algorithm and explain some design decisions. We base the parallel algorithm on the sequential GALE-SHAPLEY algorithm as presented in Section 3.1.

In the sequential algorithm the list of men is processed in sequence. As noted in Section 3.1.3, the order the men are processed does not affect the outcome. This lets us split L into $\frac{|L|}{p}$ sized partitions, where p is the number of threads. We then execute a modified GALE-SHAPLEY algorithm on these partitions in parallel, sharing the $suitor(r)$ variable between the threads. The new modified GALE-SHAPLEY algorithm has to take into account that proposals can be checked in parallel. Observe that the checking can be done in parallel, but have to be executed atomically: We can check if a man is a better suitor, but exchanging the value of $suitor(r)$ has to be done without anyone changing the value in the mean time. This is because we might end up with a situation where a proposal is rejected twice and one proposal is lost as exemplified in Figure 4.1. In Figure 4.1 both thread 1 and thread 2 are proposing simultaneously to woman a. On Thread 1 man x wants to propose to a and on Thread 2 man y wants to propose to a. Both threads checks simultaneously if their man is better ranked than z, a's current suitor. Both threads see their man as better and update the value of $suitor(a)$. Observe that this also enqueues two copies of z to each threads queue.

Time $\downarrow$

Thread 1	Thread 2	$suitor(a)$
$rejected = suitor(a)$		z
	$rejected = suitor(a)$	z
if (ranking(a,x) < ranking(a,rejected))		z
	if (ranking(a,y) < ranking(a,rejected))	z
$Q.enqueue(rejected)$		z
	$Q.enqueue(rejected)$	z
$suitor(a) = x$		x
	$suitor(a) = y$	y

Figure 4.1: Example of interleaved proposal

One way to mitigate this issue is to lock the threads before updating, one per woman to ensure that each thread gets exclusive access to this critical section. As noted in Section 2.1 once a thread has obtained a lock, no other thread can obtain the same lock before it is unlocked. An alternative method is to use atomic compare and swap on the value of *suitor*. An atomic compare and swap fails if another thread has already changed it from the value of *rejected*, letting the thread fetch the new value for $suitor(a)$ and retry the operation.

We opted for a setup where each thread has their own queue of suitors. The alternative is one shared queue with lock-protected access, but this creates contention as the thread count grows. Our approach reduces the sequential portion of the program by not having to lock on a shared queue. The downside being that the lengths of the queues might become unbalanced without any ability to share work between threads. Note that as the algorithm progresses the men will move between queues.

In Algorithm 5 we present our pseudo-code for the parallel GALE-SHAPLEY algorithm. In Algorithm 5 the **parfor** operation splits the men in L into $\frac{|L|}{p}$ partitions, where p is the number of threads, iterating the different partitions in different threads. The men are added to the queue Q, which contains the rejected or not yet matched men. One important invariant kept is that a man can not exist in more than one queue at at any time. This invariant is useful as we can progress the $nextCandidate(u)$ sequence from one thread without worrying about other threads interfering. Once a potential partner for man u has been found the current suitor of the partner is swapped with u using the $CAS(*a, b, c)$ operation. The $CAS(*a, b, c)$ operation is the compare-and-swap operation as presented in Section 2.1.2, which will only update (atomically) the $suitor(partner)$ value if it still is the same as *opponent*. If the value of $suitor(partner)$ was updated since we read the value the operation will fail as the CAS operation returns another opponent than the one we previously compared our current suitor against. The current man u is then retried against the new opponent. If he cannot beat the new opponent we put the man back on the queue to be retried against a new partner.

The algorithm terminates when all the queues are empty.

4.1.2 A parallel MCVITIE-WILSON algorithm

We will now present our parallel MCVITIE-WILSON algorithm. The parallel algorithm is based on the sequential MCVITIE-WILSON algorithm as presented in Section 3.1.3, but borrows a lot of inspiration from the parallel GALE-SHAPLEY algorithm and the parallel SUITOR algorithm[9].

As with the parallel GALE-SHAPLEY algorithm, the set L containing the men is partitioned and each partition is processed in parallel by a modified *introduceSuitor* procedure. The same observations hold for the parallel MCVITIE-WILSON algorithm as for the parallel GALE-SHAPLEY algorithm: we can check if u is a prospective partner for a woman in parallel, but the rejection has to be done atomically. The modified *introduceSuitor* procedure uses compare and swap to atomically replace the value of $suitor(partner)$ if no other thread modified it since the decision to replace was made. If a change was made to the $suitor(partner)$ variable the algorithm immediately retries u against the womans new suitor. If the new opponent is better ranked than u the man is considered as rejected and is introduced into the solution again starting at his next candidate.

A notable difference to the GALE-SHAPLEY algorithm is, as noted in Section 3.1.3, that when the last thread is done processing its part of the problem the algorithm is done. This property allows the parallel MCVITIE-WILSON algorithm to allocate parts of L to threads on the go as they are finished with their work, preventing load imbalance.

In Algorithm 6 we present pseudo-code for the algorithm.

4.2 Parallel b-Marriage

In this section we introduce a parallel algorithm for the b-Marriage problem, and give pseudo-code for it. The parallel algorithm is based based on the sequential B-MARRIAGE algorithm presented in Section 3.2, but draws inspiration from the parallel b-SUITOR algorithm by Khan et.al.[6]

In the parallel version of the algorithm, similarly to the parallel MCVITIE-WILSON and GALE-SHAPLEY algorithms, the set of men is split among the threads. The men are then processed by a modified B-MARRIAGE algorithm.

Recall the sequential B-MARRIAGE algorithm in Algorithm 3 and observe that the *nextCandidate* sequence is shared between the b copies of the men. In the sequential version the implementation of *nextCandidate* is not crucial, but in a multithreaded setting there might be multiple processors trying to advance this pointer at the same time. This is because different copies of the same man can be the current man of multiple threads concurrently. This can lead to inconsistent results like a man being matched multiple times with the same woman, if the counter is incremented as shown in Figure 2.1. Using atomic operations, like atomic fetch and add, we can safely increment the counter. Atomic fetch and add gives each thread a unique candidate at a little overhead relative to the simple write[11].

The second part that requires synchronisation is when we check wether or not a man will be considered for a woman. Here we implemented the check-lock-recheck method where we first check if a man can win against the current opponent (the worst ranked man that has proposed), lock and check if that man still is a better suitor for the partner, and add the man to the priority queue. If the man no longer is a better suitor he is considered rejected and is immediately recursively retried with his next partner. The check before locking lets the algorithm avoid locking for cases where there is no chance of the application being accepted by the college. Using thread-safe priority-queue implementations would avoid locking here, but we want our algorithm to not depend on the safety of the queue implementation.

In Algorithm 7 we give our implementation of the parallel b-marriage algorithm. In this implementation we show a new implementation of $nextCandidate(l)$ which is thread-safe. To implement the nextCandidate we introduce a new variable $candidate(l)$ which we can atomically fetch and add (FAA) (see Section 2.1.2).

4.3 Parallel Popular Matching

In this section we present our parallel algorithm for popular matching based on the algorithm presented in Section 3.3.1, and present an implementation of the algorithm based on Algorithm 4.

The parallel HUANG-KAVITHA algorithm works much in the same way as the parallel MCVITIE-WILSON algorithm. Recall that in the sequential algorithm (Algorithm 4) we compare both the level and the ranking our current man with that of the potential partners current suitor. In the parallel algorithm, between the time we check if u is a better suitor and the time we compare and swap the value of $suitor(partner)$, the suitor may have leveled up. In this case the current suitor should win against our prospective suitor. To solve this we change the $suitor(l)$ variable to be a tuple of the suitor id and his level, and do compare and swap on the whole tuple. This ensures that if the level of the suitor id has changed, the compare and swap fails. As with the parallel MCVITIE-WILSON algorithm, in the case the CAS operation returns another opponent than the one we previously compared our current suitor against, we have to retry against the new opponent. If he cannot beat the new opponent we immediately reintroduce the man to the solution.

Algorithm 5 The parallel GALE-SHAPLEY algorithm

```
#omp parallel do
    $Q = \emptyset$
    parfor each $l$ in $L$ do
        $Q.enqueue(l)$
    end parfor
    while $Q \neq \emptyset$ do
        $u = Q.dequeue()$
        $partner = nextCandidate(u)$
        $opponent = suitor(partner)$
        while $partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, opponent)$ do
            $partner = nextCandidate(u)$
            $opponent = suitor(partner)$
        end while
        while $partner \neq NULL$ do
            if $ranking(partner, u) < ranking(partner, opponent)$ then
                if $CAS(suitor(partner), opponent, u) == opponent$ then
                    $Q.enqueue(opponent)$
                    $partner = NULL$
                else
                    $opponent = suitor(partner)$
                end if
            else
                $Q.enqueue(u)$
                $partner = NULL$
            end if
        end while
    end while
end parallel
```

Algorithm 6 The parallel MCVITIE-WILSON algorithm

```
procedure PARALLEL-MCVITIE-WILSON
    #omp parallel do
        parfor each $l$ in $L$ do
            $introduceSuitor(l)$
        end parfor
    end parallel
end procedure
procedure INTRODUCESUITOR($u$)
    $partner = nextCandidate(u)$
    $opponent = suitor(partner)$
    while $partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, opponent)$ do
        $partner = nextCandidate(u)$
        $opponent = suitor(partner)$
    end while
    while $partner \neq NULL$ do
        if $ranking(partner, u) < ranking(partner, opponent)$ then
            if $CAS(suitor(partner), opponent, u) == opponent$ then
                $introduceSuitor(opponent)$
                $partner = NULL$
            else
                $opponent = suitor(partner)$
            end if
        else
            $introduceSuitor(u)$
            $partner = NULL$
        end if
    end while
end procedure
```

Algorithm 7 A parallel b-marriage algorithm

```
procedure PARALLEL-B-MARRIAGE
    #omp parallel do
        parfor each $l$ in $L$ do
            for each $c$ in $c \mid c > 0, c < b_m(l)$ do
                $introduceSuitor(l)$
            end for
        end parfor
    end parallel
end procedure
procedure NEXTCANDIDATE(u) return $FAA(candidate(l), 1)$
end procedure
procedure INTRODUCESUITOR($u$)
    $partner = nextCandidate(u)$
    while $(partner \neq NULL \wedge$
    $ranking(partner, u) > ranking(partner, pq(partner).peek()))$ do
        $partner = nextCandidate(u)$
    end while
    if $partner \neq NULL$ then
        $lock(partner)$
        if $ranking(partner, u) <$
     $ranking(partner, pq(partner).peek()))$ then
            $pq(partner).enqueue(u)$
            $unlock(partner)$
            if $|pq(partner)| > b_w(partner)$ then
                $introduceSuitor(pq(partner).dequeue())$
            end if
        else
            $unlock(partner)$
            $introduceSuitor(u)$
        end if
    end if
end procedure
```

Algorithm 8 The parallel HUANG-KAVITHA algorithm

```
procedure PARALLEL-HUANG-KAVITHA
    #omp parallel do
        parfor each $l$ in $L$ do
            $introduceSuitor(l)$
        end parfor
    end parallel
end procedure
procedure INTRODUCESUITOR($u$)
    repeat
        $partner = nextCandidate(u)$
        $level_u = level(u)$
        if $partner \neq NULL$ then
            $rank_u = ranking(partner, u)$
            $(opponent, level_o) = suitor(partner);$
            $rank_o = ranking(partner, opponent)$
        end if
    until $partner == NULL \vee$
    $level_u > level_o \;\vee\; (level_u == level_o \;\wedge\; rank_u < rank_o)$
    if $partner == NULL$ then
        $level(u) = level(u) + 1$
        $resetNextCandidate(l)$
        $introduceSuitor(u)$
    else
        while $partner \neq NULL$ do
            if $ranking(partner, u) < ranking(partner, opponent)$ then
                if $CAS(suitor(partner), (opponent, level_o), (u, level_u))$
                $== (opponent, level_o)$ then
                    $introduceSuitor(opponent)$
                    $partner = NULL$
                else
                    $(opponent, level_o) = suitor(partner);$
                    $rank_o = ranking(partner, opponent)$
                end if
            else
                $introduceSuitor(u)$
                $partner = NULL$
            end if
        end while
    end if
end procedure
```

Chapter 5

Experimental results

In this chapter we present the runtime results of the parallel algorithms presented in Chapter 4

5.1 Test setup

Our test setup is the computer Lyng with two Xeon E5-2699 v3 CPUs running at 2.30GHz with 252GB ram. Each of the CPUs has 18 cores. Connected to the machine is a Xeon Phi 7120 “Knights Corner” co-processor running at 1.23GHz with 16GB GDDR5 RAM and 61 active cores, one being reserved by the management OS running on it. The Xeon Phi cores has 4 hardware threads each, totaling to 244 hardware threads (240 available to OpenMP, 4 being reserved by the coprosessor operating system), while the Xeon E5 has 2 threads per core, totaling to 72 threads. For some experiments we used a second system “brake” with 240GB RAM and 40 cores. The specs are also listed in Figure 5.1. The machines are shared memory computers. This means all the cpu cores have direct access to all the memory.

All programs were written in C++ and compiled with the Intel compiler icpc (ICC) 17.0.3 20170404 with -O3 optimisation enabled. We had gcc available but it was not used because of the lacking support for offloading to the xeon phi.

Name	CPU	L1 cache	L2 cache	L3 cache
Brake	4x Xeon E7-4850 @ 2.00GHz	10x32KB	10x256KB	24MB
Lyng	2x Xeon E5-2699 v3 @ 2.30GHz	18x32KB	18x256KB	45MB
Xeon Phi	Xeon Phi 7120 @ 1.24 Ghz	61x32KB	61x512KB	None

Figure 5.1: Machine CPU specifications

For each of the algorithms we first run the sequential version where all OpenMP/parallelisation-related code has been removed. The sequential version is optimised for single-thread speed and does not have to care about race conditions with other threads. The run time measured on the best sequential algorithm is used as a baseline when calculating the speedup gained from parallelisation.

The run times in the parallel versions are measured using the OpenMP built-in omp_get_wtime function. The timing includes the allocation and initialisation of memory for use in the algorithm, but not the loading of input data.

5.1.1 Xeon Phi

The Xeon Phi coprocessor is a stripped-down computer with its own processors and memory. It is connected to a host computer via the PCIe bus, allowing for high-bandwidth asynchronous transfer of data between the host and the Phi. The bandwidth of the PCIe-bus (roughly 16GB/s) is significantly lower ($>$ 22x slower) than the on-card memory bandwidth (at 352 GB/s), so using the host memory directly is out of the question as the PCIe-bus would quickly become a bottleneck. The mic-architecture of the Xeon Phi builds upon a high number of stripped down, clocked down 64-bit x86 cores with a high-speed interconnect. The cpu cores are optimised for parallel numerical applications with its 4 hardware threads and extra wide vector instructions.

There are two modes of using the Xeon Phi:

- Native: compiling the program for the mic architecture, logging in to the linux-os running on the card and then execute the program.
- Offload: using OpenMP pragmas like offload, which transfers the given function and any dependent data via the PCIe-bus onto the card, resuming execution on the card.

We primarily used the offload mode because the native mode could make the card unstable. This would happen if the program was to run out of memory, in which case the card would hang rebooted.

5.2 Instances

To evaluate the effectiveness of our parallelisation we run our algorithms on a combination of random graphs and real graphs. We constructed 2 different types of random graphs: an “easy” random graph with a realtively low number of randomly

Table 5.1: Easy instances

Problem	**n**	**Edges**	**Min. degrees**	**Max. degree**
10M	10000000	340007576	$log(n)$	$2log(n)$
20M	20000000	709985096		
40M	40000000	1480074907		
50M	50000000	1849916651		
60M	60000000	2220111233		
75M	75000000	2887538634		
100M	100000000	3850014406		
125M	125000000	4812489007		

Table 5.2: Hard instances

Problem	**n**	**Edges**	**Min. degree**	**Max. degree**
100K	100K	10B	100K − 1	100K − 1
200K	200K	40B	200K − 1	200K − 1
300K	300K	90B	300K − 1	300K − 1
400K	400K	160B	400K − 1	400K − 1
500K	500K	250B	500K − 1	500K − 1

distributed edges (between $log(n)$ and $2 \cdot log(n)$) per vertex and a “hard” instance where each side had a common ranking and ($|M| \cdot |W|$) edges.

The easy instances were created by assigning a random number of edges between $log(n)$ and $2log(n)$ to each man. For each man the ranks were then shuffled. These random graphs are considered easy because the average number of proposals to reach a stable matching is < 5. Table 5.1 shows the easy graphs we generated. The 125M instance was the largest one we could fit in RAM on lyng without resorting to swapping data out and in from disk. Because of the restrictive memory limit on the Xeon Phi, the largest graph we could fit was at 20M vertices.

What we call a hard instance is the case where each man share the same preference for the women, meaning they rank them equally, and the woman do the same for the men. We consider this hard because in any stable matching on this instance type only one suitor would get his best ranked, only one would get his second best ranked and so on. This means our algorithms considers $1 + 2 + 3 + .. + n$ edges, summing up to $\frac{(n-1) \cdot n}{2}$ edges. Table 5.2 shows the hard instance we generated.

The real world graphs we use are listed in Table 5.3. The graphs were retrieved from the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection).[1] For these problems we ran a reduction as

Table 5.3: Real world problems

Problem	**n**	**edges**	**avg.degree**	**max.degree**
Fault_639	638802	58506732	44	1268
kron_g500-logn21	2097152	364168040	88	855616
Reuters911	13332	592152	48	9060

described in the paper by Manne, Naim, Lerring, and Halappanavar (AppendixA.1) where it is shown that the problem of Greedy Matching can be reduced to an instance of stable marriage. The reduction requires each side of the bipartite graph to contain all the vertices. This doubles the number of vertices, but for the stable marriage problem we let n be the size of either side. Because the original graph is undirected and the stable marriage instance is directed the number of edges is multiplied by 4. It is worth noting that the resulting stable matching is also a greedy matching in the original graph.

We ran both our sequential versions of the algorithms against the datasets, measuring which one was fastest and compared the runtime of the fastest against each of the parallel algorithms. This gives us the speedup gained by utilising multiple processors. We ran each algorithm/instance size pair 3 times and took the median runtime. This is to avoid any external factors like other jobs running on the machine that might disturb the timing. The parallel runs were done on the Lyng machine at UiB, as described in Section 5.1, with up to 72 threads on the host machine and up to 240 threads on the Xeon Phi.

5.3 Stable marriage results

In this section we present our experimental results. For each algorithm we first present the sequential benchmarks, then the parallel runtimes, speedup and efficiency. For brewity sake we present only smaller subsets of the instances, with the note that the rest of the results tell a similar story.

5.3.1 Solution quality

As noted in Section 2.3, a stable marriage in an SMI instance may not be a complete match. The total solution quality for the easy instances are quite good, as can be seen in Figure 5.2 where the number of men and women with a match is 98% $\pm$ 0.2. Note that any algorithm solving the stable marriage problem with incomplete preference lists obtains the same solution size on these instances [3].

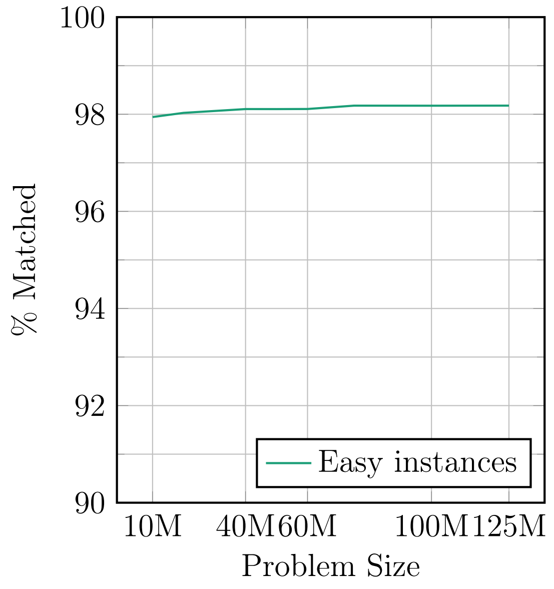

Figure 5.2: Easy instances solution quality

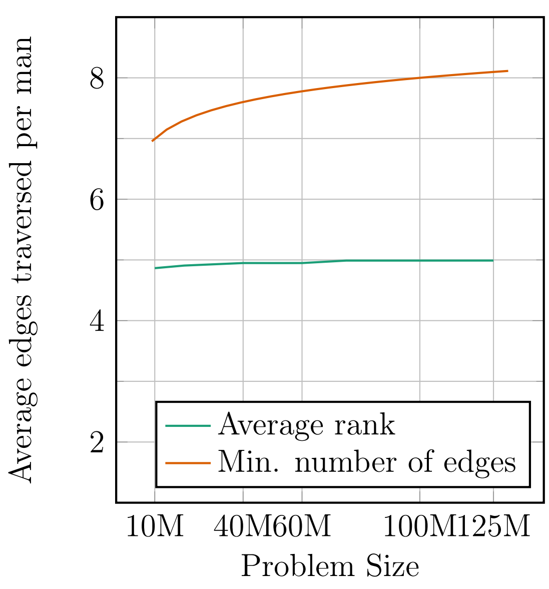

Figure 5.3: Easy instances average mens rank

The solution quality is expected as the average final rank in the man-optimal solution is < 5, as showin in Figure 5.3.

5.3.2 Easy instances

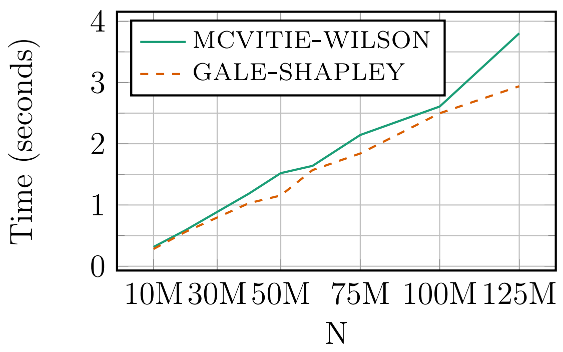

Figure 5.5: Easy instances runtime on Lyng (less is better)

In Figure 5.4 we see the sequential runtime of the two variants of the algorithm. The GALE-SHAPLEY algorithm being a few percent faster in every instance. We believe this is because the GALE-SHAPLEY algorithm starts out with the first queue in the order the suitors are stored in memory, leading to good cache hit rates on the arrays storing information about the men like the nextCandidate. In our c++ implementation each man takes 12 bytes. 8 bytes for a lookup table and 4 bytes for

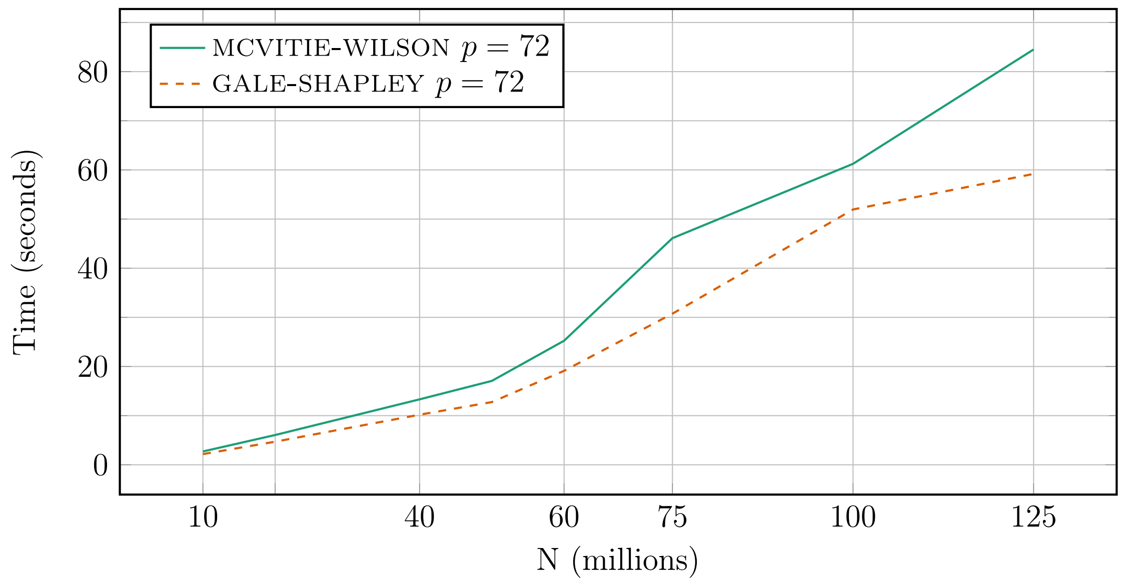

Figure 5.4: Easy instances sequential runtimes on Lyng (less is better)

the next candidate rank. This allows
for a prefetch of 8 values of the lookup table and 16 values of the next candidate rank per cache line.

After the first queue is consumed the algorithm starts processing the "rejects" of the previous round. These next men are now in random order. The MCVITIE-WILSON algorithm does not obtain as good caching of these two arrays, as it jumps around in the memory from the start, stabilising the solution before letting a new suitor propose. Our timings show that the GALE-SHAPLEY algorithm spends around 33% of its time in this first memory-sequential phase.

Figure 5.5 show the runtimes on the parallel GALE-SHAPLEY algorithm. Here we see that the trend with GALE-SHAPLEY algorithm getting better runtimes also extends to the parallel algorithm.

Figure 5.6 shows the speedup gained with our parallel algorithms. We get a speedup of between 16x and 20x with the parallel GALE-SHAPLEY algorithm compared to the sequential GALE-SHAPLEY algorithm. The parallel MCVITIE-WILSON algorithm does a little bit worse with between 14x and 19.9x speedup compared to the sequential GALE-SHAPLEY algorithm. The dip in speedup from 27 to 36 threads could be due to another task on the system using one thread. This is further substantiated with our results on the older system "brake" which did not

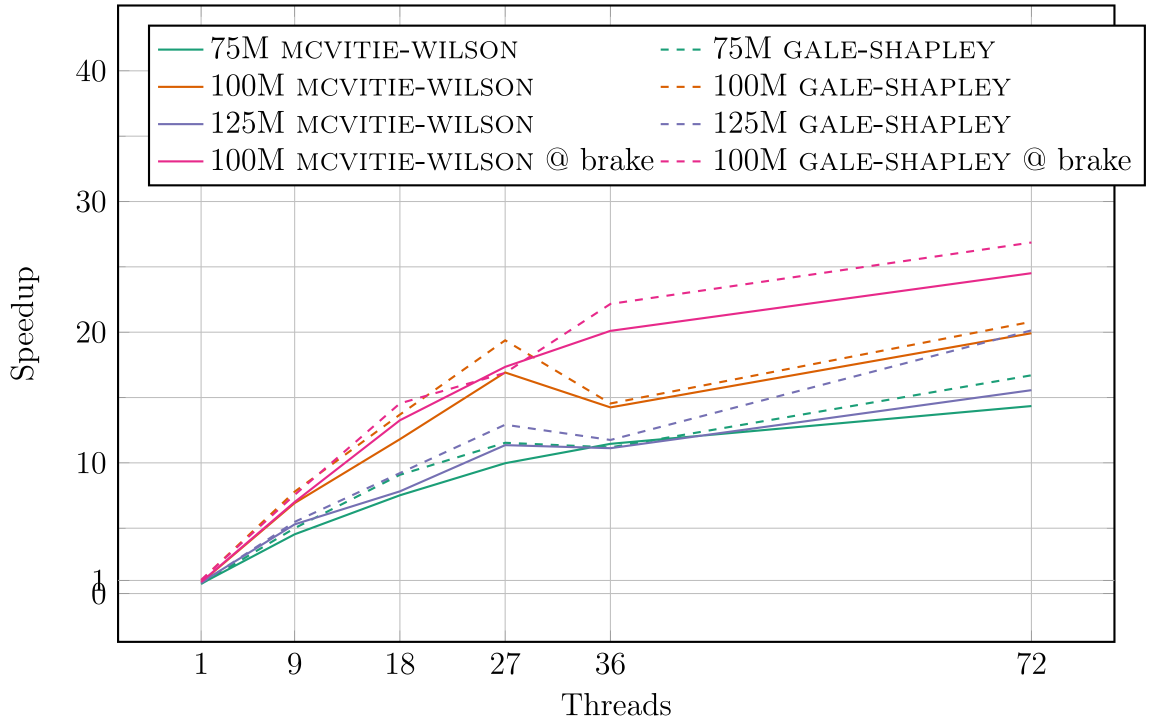

Figure 5.6: Easy instances speedup ($T(1)/T(p)$,more is better)

display this dip in performance. On 72 threads there are multiple threads per processor which lessens the impact of one slower thread. We believe this difference in speedup, with the GALE-SHAPLEY algorithm getting a better speedup, is due to the additional cache available with each extra cpu core.

5.3.3 Hard instances

On the hard instances, even though the instances are greatly reduced in memory footprint, the time required to solve them is greatly increased, as shown in the difference between the figures 5.4 and 5.7. Here we see the largest instances taking over 50 minutes to complete by the sequential algorithm. This is because the average number of edges we have to process has grown from < 5 to $\frac{n-1}{2}$. The memory required to store these instances is now reduced to 1.6 - 4.6MB, an amount that is cacheable even with the random access patterns. We can see the effect of this because now the McVitie-Wilson algorithm is the fastest and Gale-Shapley the slower one. We believe this is due to the extra queue operations, which now outweighs the cache gains from the sequential memory access in the first part, now being less than 0.3% of the total runtime on the sequential version.

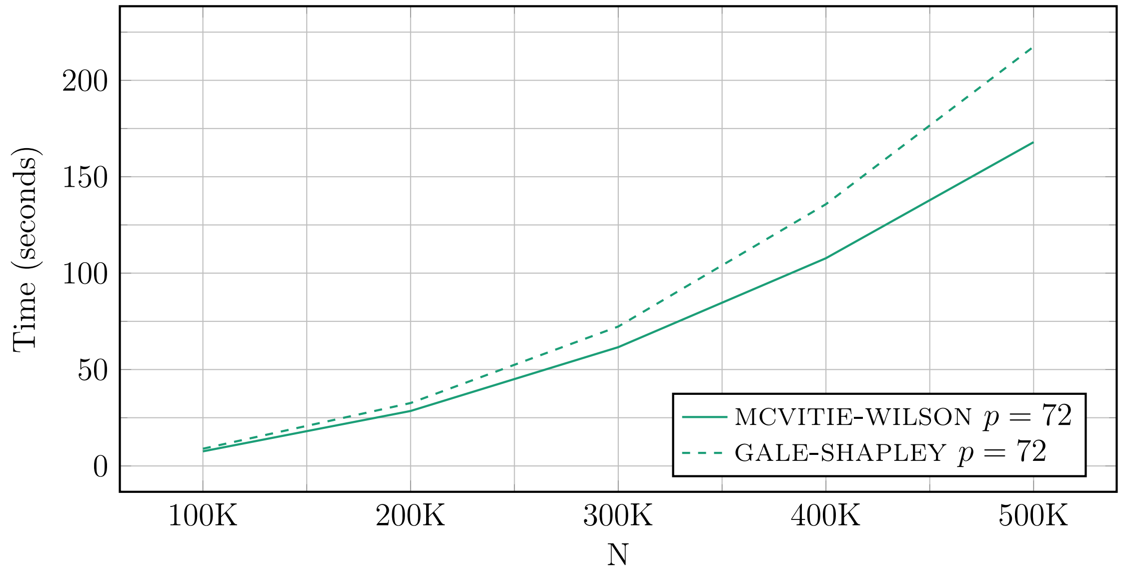

Figure 5.8: Parallel runtime on hard instances (less is better)

Our parallel algorithms performs just as well on the hard instances. The parallel MCVITIE-WILSON algorithm traverses 744M edges per second at 72 threads, reducing the runtime from 47 minutes to 2.75 minutes. The GALE-SHAPLEY variant gets gradually worse relative to the parallel MCVITIE-WILSON, likely because of the $\mathcal{O}(n^2)$ number of queue reads and writes it does to keep track of the unmatched men, traversing 574M edges per second at 72 threads, going from 53 minutes to 3.5 minutes.

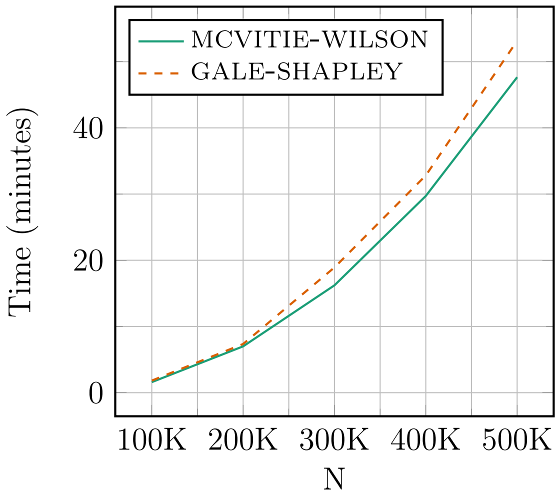

Figure 5.7: Hard instances sequential runtimes on Lyng (less is better)

As we can see from Figure 5.10, the efficiency of the parallel GALE-SHAPLEY and parallel MCVITIE-WILSON

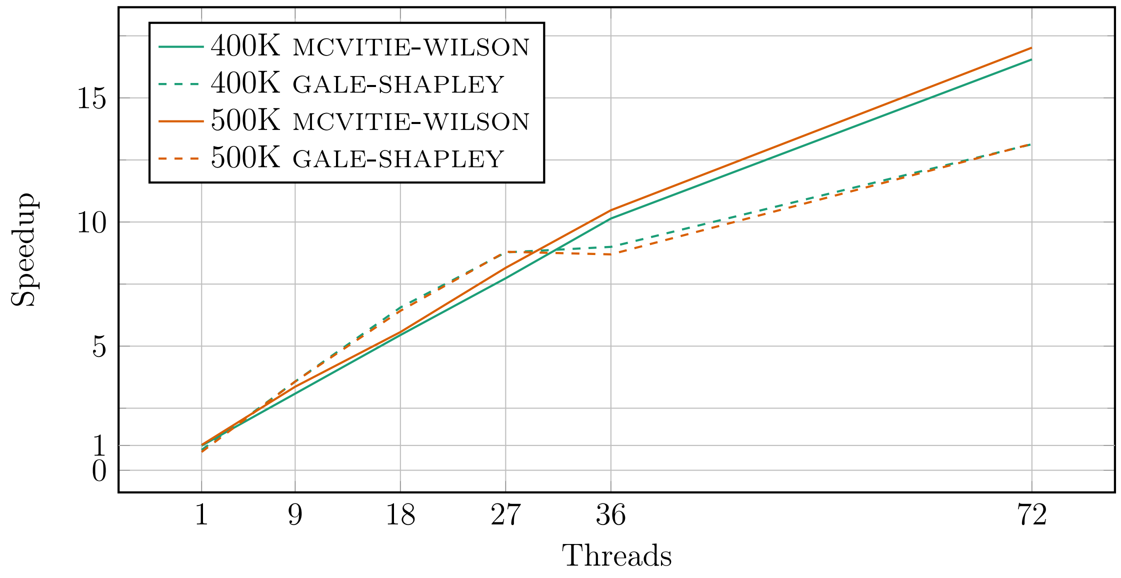

Figure 5.9: Speedup on hard instances (more is better)

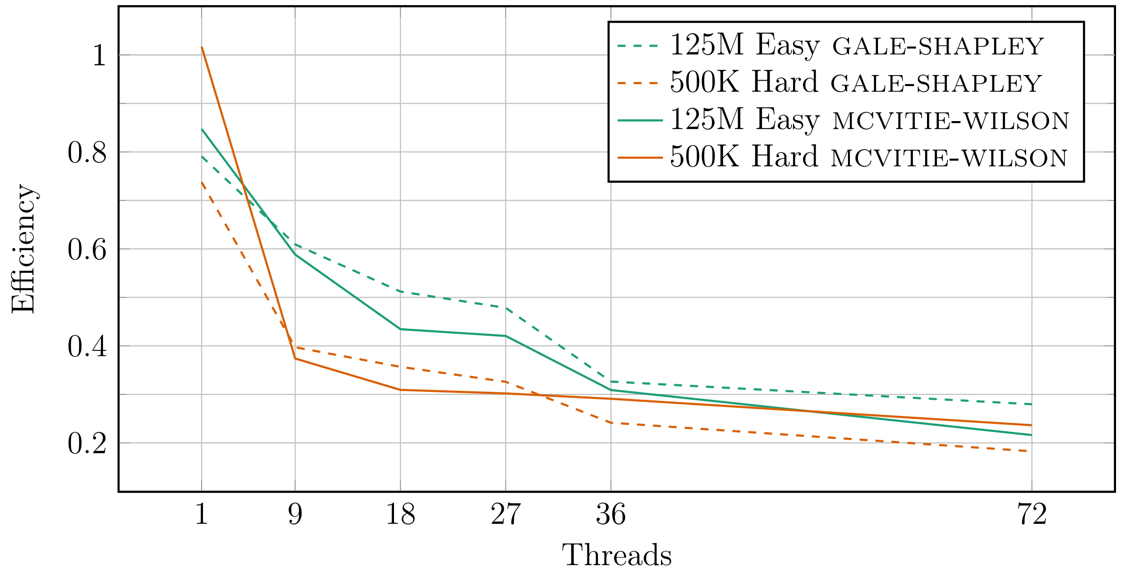

Figure 5.10: Efficiency on easy and hard instances (more is better)

is quite similar. The efficiency of GALE-
SHAPLEY is a little more divergent on the higher thread counts, which is explained by the slight load imbalance that may ocurr.

5.3.4 Real graphs

We will now present our results on the graphs obtained from the SuiteSparse Matrix Collection.

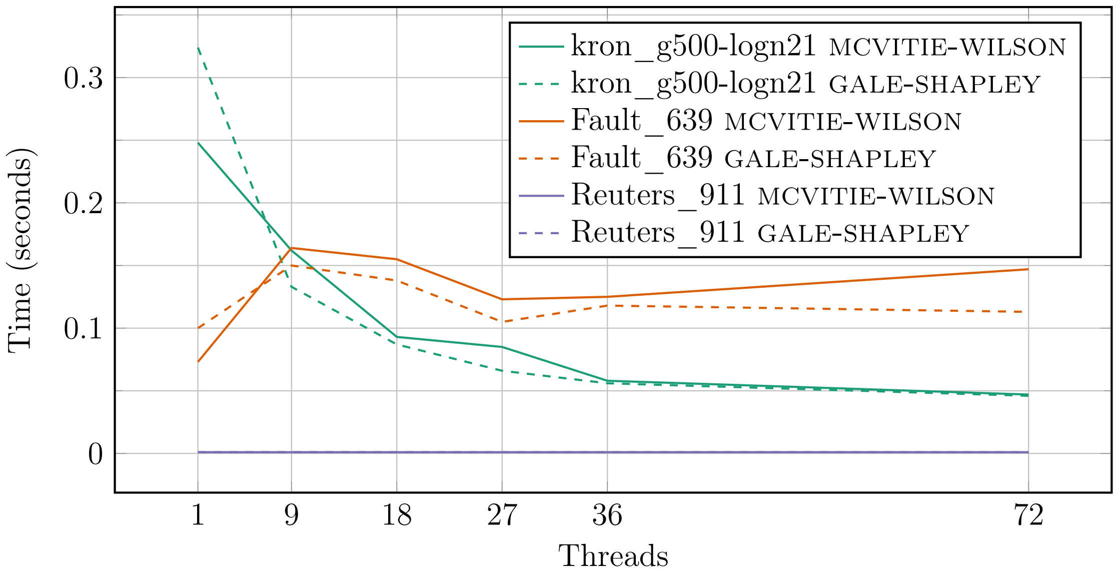

Figure 5.11: Real instances timing (less is better)

As previously noted these graphs are preprocessed into a stable marriage instance, which includes sorting and expansion of the graph, but the time for this is not included in our timing. Because of this the time is not really comparable to other Greedy Matching results. As Figure 5.11 shows, the time it takes to obtain a stable matching in these graphs is quite small (all instances were solved in under a second sequentially), and the gain from parallelising is diminishing. Regardless of that we get a speedup of 4.5 (Figure 5.12) on the kron_g500-logn21 graph. The time it took to solve the Reuters911 graph was below the resolution of our measurement.

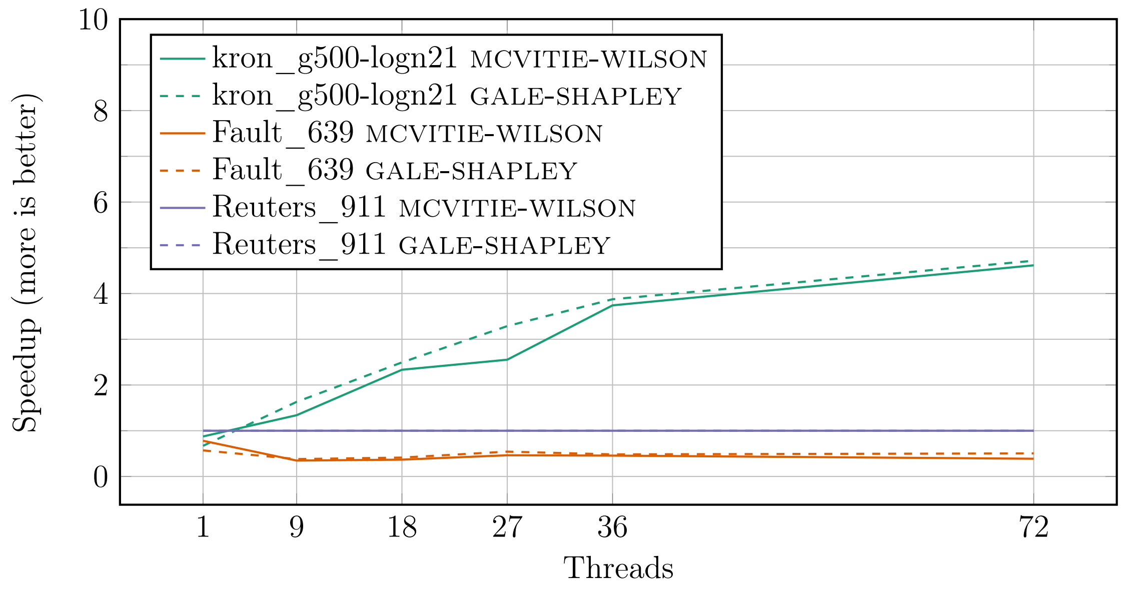

Figure 5.12: Real instances speedup (more is better)

5.3.5 Xeon Phi performance

In this section we present our results for running the GALE-SHAPLEY and MCVITIE-WILSON algorithms on the Xeon Phi.

In Figure 5.13 and Figure 5.14 we compare the best runtime on the Xeon Phi to the best runtime on Lyng. With only one exception (MCVITIE-WILSON on the easy instances with 10M men/women), the Phi performed significantly worse than on Lyng. The per-thread reduction in performance is expected since the clock speed of the Xeon Phi is roughly 50% that of Lyng (the raw speed of 144 threads compare to that of 72 threads on the host). This, combined with the increased contention and subsequent retries explains the poorer performance on higher thread counts. In Figure 5.15 we present the speedup from running MCVITIE-WILSON and GALE-SHAPLEY on the Xeon Phi. Note that the speedup is measured relative to the sequential GALE-SHAPLEY on Lyng. The jitter observed in this plot is likely because of the random nature of the retry when compare and swap fails.

5.4 b-Marriage results

In this section we present our experimental results with the parallel B-MARRIAGE algorithm. We compare the parallel B-MARRIAGE algorithm (Algorithm 7) to the

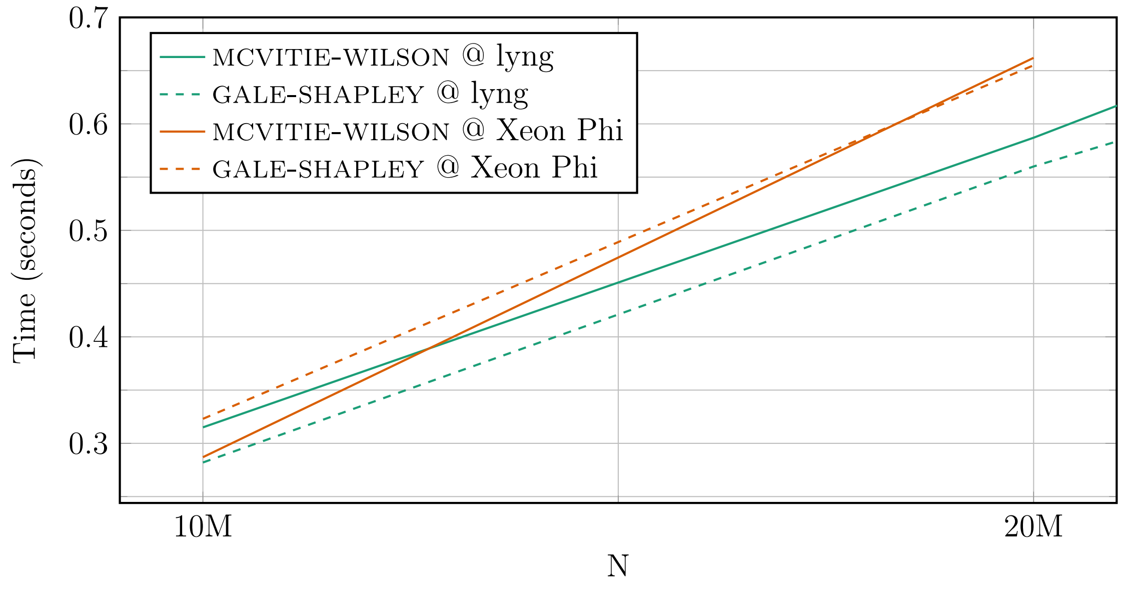

Figure 5.13: Best Xeon Phi (240 threads) vs Lyng (72 threads) times on easy instances (less is better)

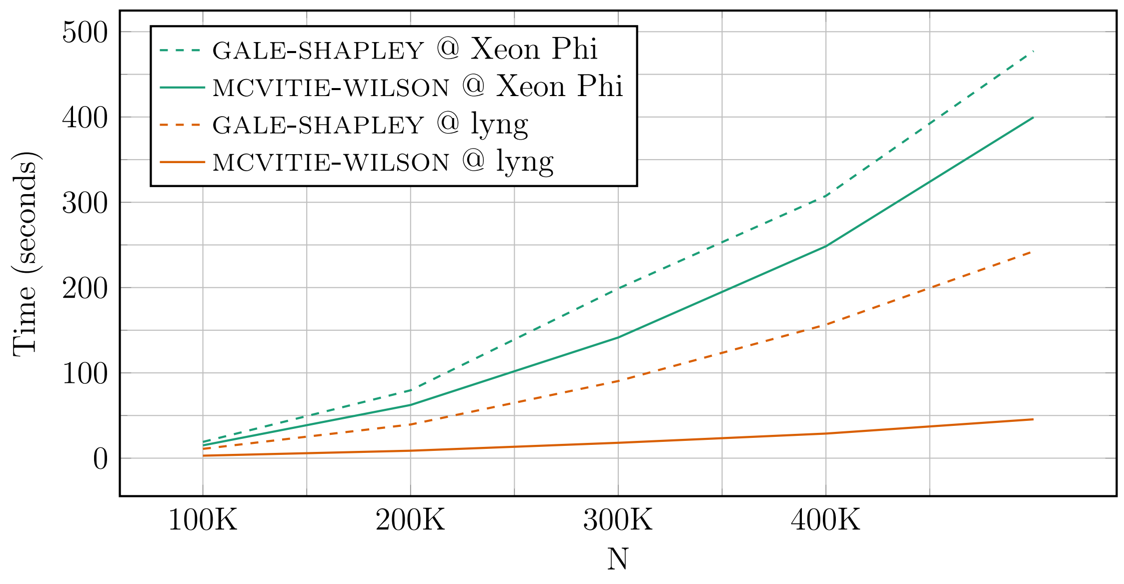

Figure 5.14: Best Xeon Phi (240 threads) vs Lyng (72 threads) times on hard instances instances (less is better)

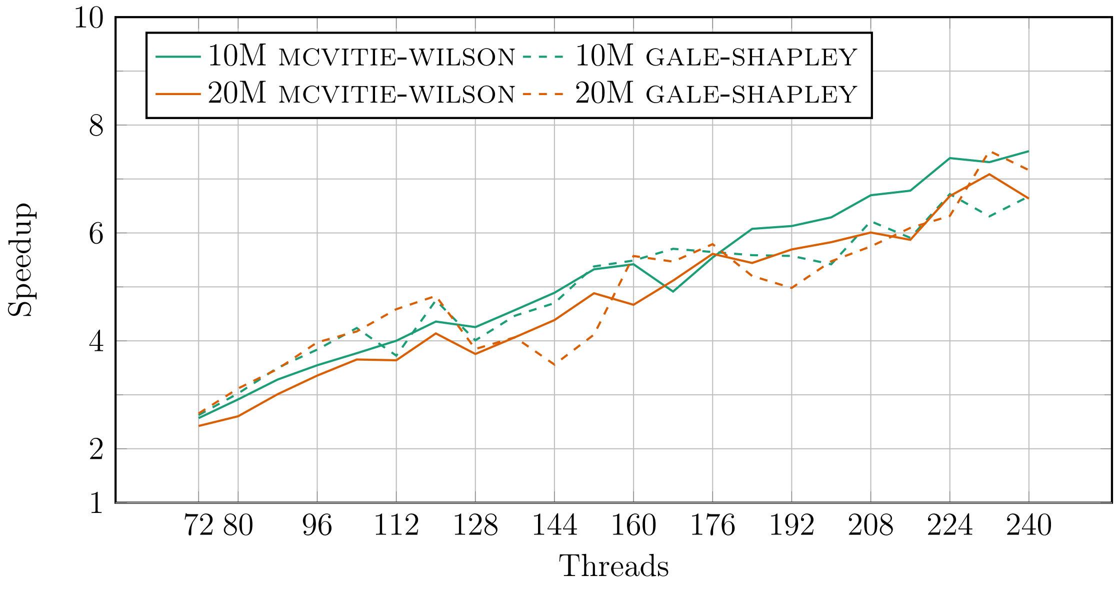

Figure 5.15: Easy instances speedup on Xeon Phi over GALE-SHAPLEY on Lyng (more is better)

parallel MCVITIE-WILSON algorithm (Algorithm 6), to show the overhead imposed by keeping a queue of suitors. We present the speedup on the different datasets with $b = 1$, 3 and 5. We tested 2 different implementations for the priority queue used to store the suitors in. One using linear search (LS) through an array and one using a heap. We show that different priority queue implementations have a significant impact on the efficiency when we vary b.

First it is interesting to know how the B-MARRIAGE algorithm performs in relation to the MCVITIE-WILSON algorithm. Figure 5.16 shows the runtime of the parallel MCVITIE-WILSON algorithm and the parallel B-MARRIAGE algorithm with $b = 1$ on easy instances. It shows the B-MARRIAGE to use between $40\% - 50\%$ more time than the MCVITIE-WILSON algorithm. This is likely due to three factors: the locking necessary for updating the priority queue, the atomic fetch and add in the *nextCandidate* function, and extra operations to update the priority queue itself. We did not test the algorithm on $b = 1$ above 60M as we believe as we believe a point was made: a specialised version of the algorithm for problems where $b = 1$ yields significant runtime gains over using the more general B-MARRIAGE algorithm.

In Figure 5.17 we present the runtime results for the B-MARRIAGE algorithm. We see that the runtime on $b = 5$ is significantly slower than the same datasets

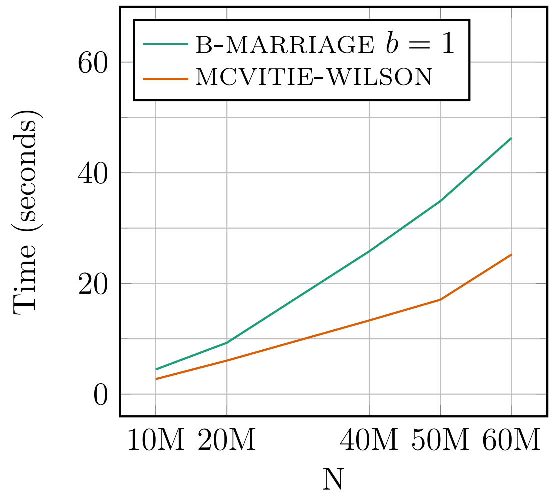

Figure 5.16: B-MARRIAGE (LS) $b = 1$ vs MCVITIE-WILSON runtime (less is better)

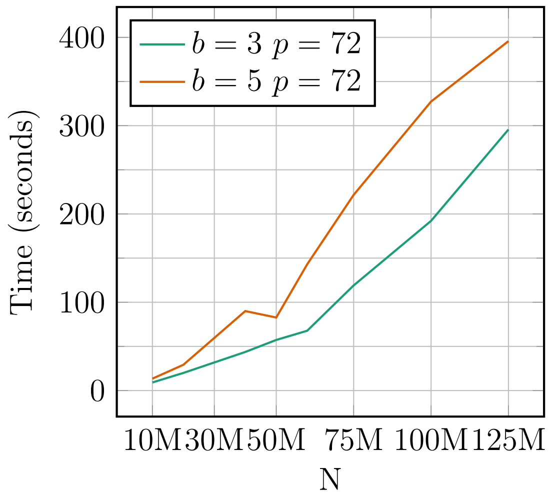

Figure 5.17: B-MARRIAGE (heap) best runtime (less is better)

when $b = 3$. This is likely due to there being $\frac{5}{3}$ times more matches to be made.

Figure 5.18 shows speedup on easy instances (with heap as queue implementation) with $b = 3$ and $b = 5$. Here we see that that the parallel algorithm gives speedups of up to $41x$ on $b = 5$ and up to $27.8x$ on $b = 3$. We think the increase in speedup with the increased value of b is caused by the parallel algorithm spending proportionally less time in a critical section.

In our tests the atomic operation did better than locking and unlocking with OpenMP lock structures.

Recall from Algorithm 7 that if the queue is not full it does not dequeue the worst candidate. When we increase the value of b, we increase the amount of matches being made without rejecting an existing proposal, and therefore spend proportionally less time with a lock. Depending on the implementation of the queue this operation might be expensive. The results presented Figure 5.18 were done with a heap, which requires $O(log(b))$ operations for each enqueue/dequeue operation.

In Figure 5.19 we run the same algorithm where the heap is replaced with a linear search through an array. The linear search requires $O(b)$ operations for both enqueue and the enqueue/dequeue operation, but has a much smaller overhead on small values of b. The difference in time spent locked in the two execution paths is visible in the plot, making the difference in speedup between the two values of b smaller. This difference in efficiency is also evident in the effiency plot presented

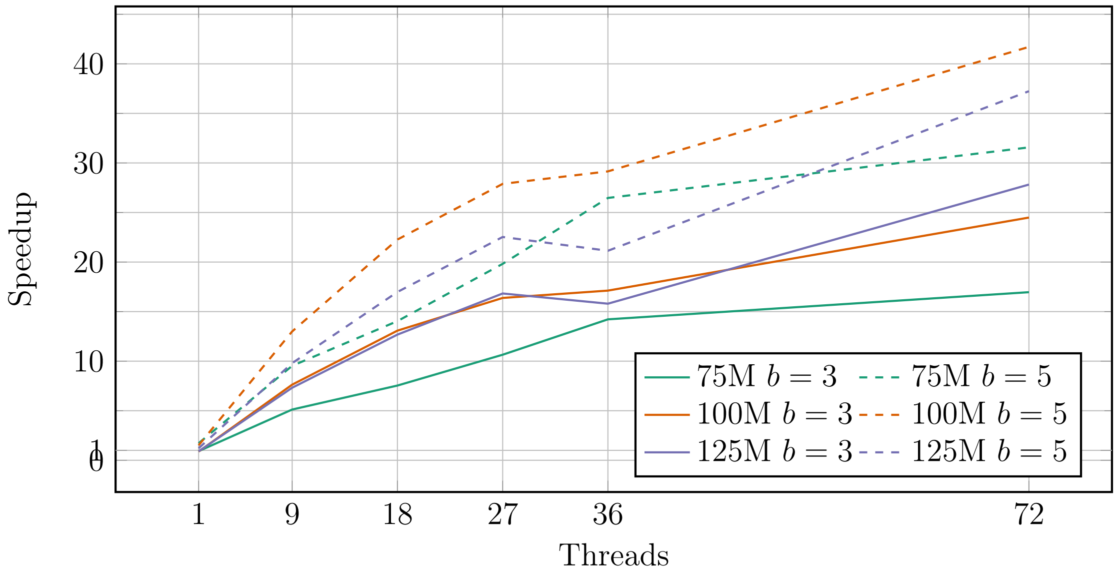

Figure 5.18: Easy instances speedup with heap variant (more is better)

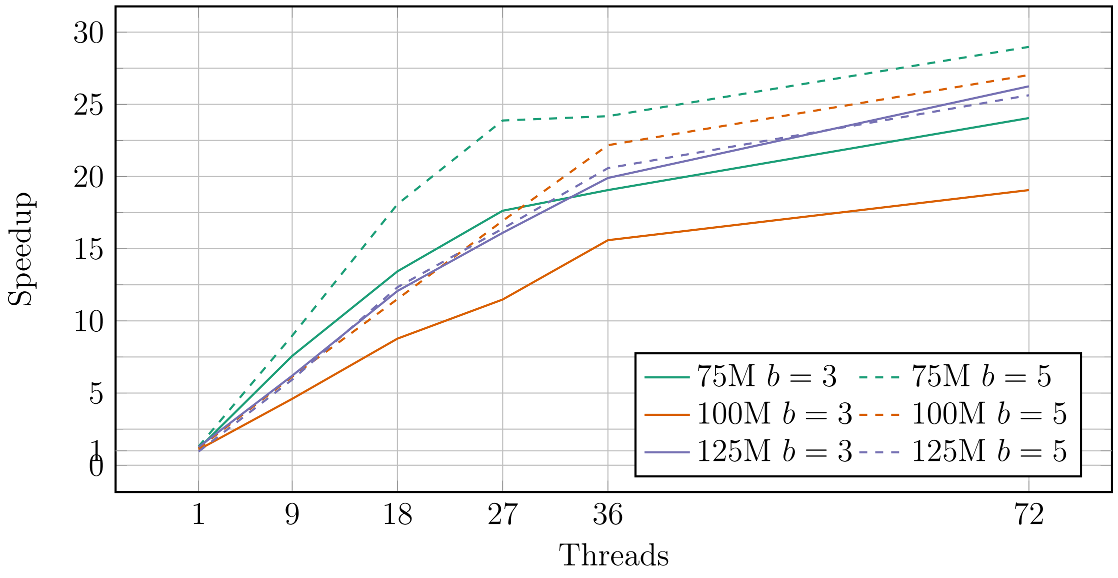

Figure 5.19: Easy instances speedup with linear search variant (more is better)

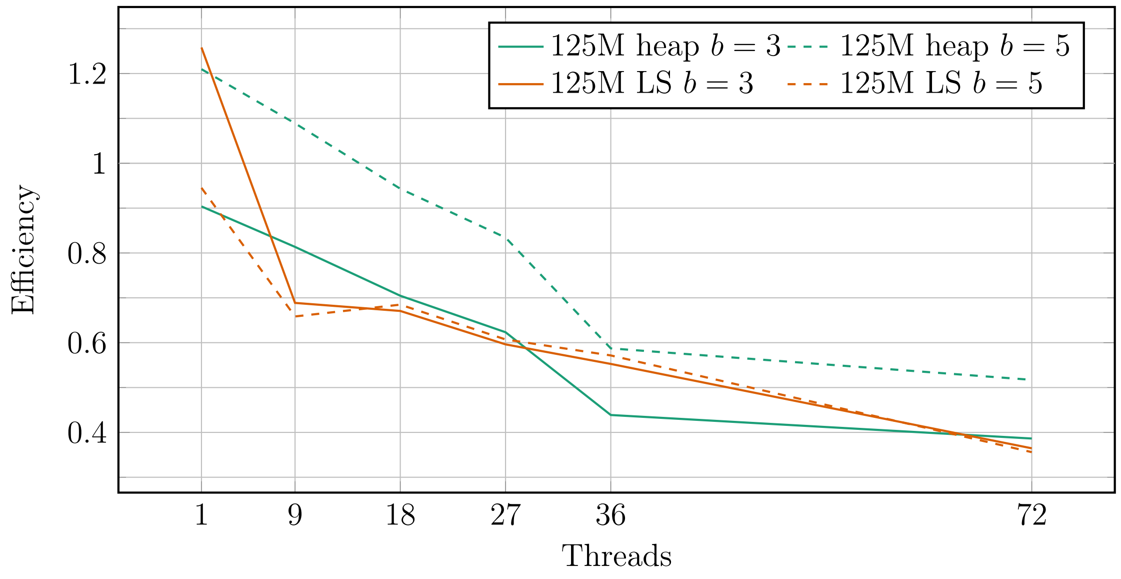

Figure 5.20: Easy instances efficiency comparison ($T(1)/T(p)/p$, more is better)

in Figure 5.20, where the efficiency of the LS variant is less dependent on b than the heap variant.

For the hard instances we experienced more variance in the speedup, as shown in Figure 5.21. For these results we did not have exclusive access to the machine, but they serve as an indication of what we can expect without competition.

5.5 Popular matching results

In this section we present the experimental results of running the sequential and parallel versions of the HUANG-KAVITHA algorithm. As explained in Section 3.3, the popular matching problem is primarily a problem on SMI instances (on SM instances this is simply an instance of the stable marriage problem). Because of this we only test the algorithms on SMI instances.

In Figure 5.22 we compare the runtime of the parallel MCVITIE-WILSON algorithm and the parallel HUANG-KAVITHA algorithm. Recall from Section 3.3.1 that only the unmatched men of the first stable marriage solution is lifted to a higher level and retried. Because the stable marriage match percentage on these instances is quite high (see Section 5.3.1), the number of men brought up to the second level is quite low (less than 3%). This makes the HUANG-KAVITHA algorithm perform quite close to the MCVITIE-WILSON algorithm, with a slight deviation at the 75M,

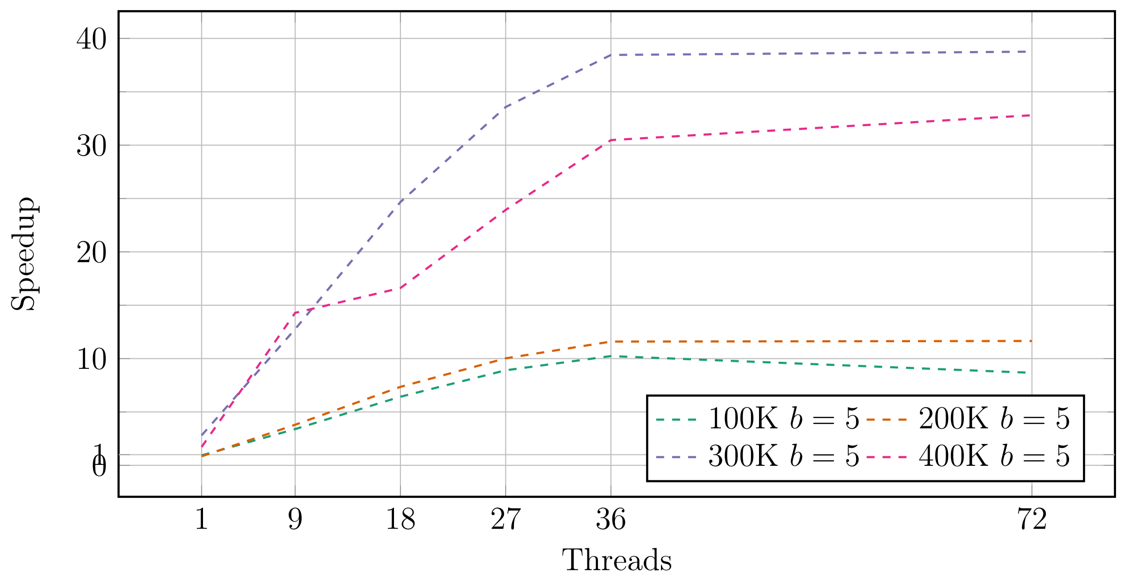

Figure 5.21: Hard instances speedup (more is better)

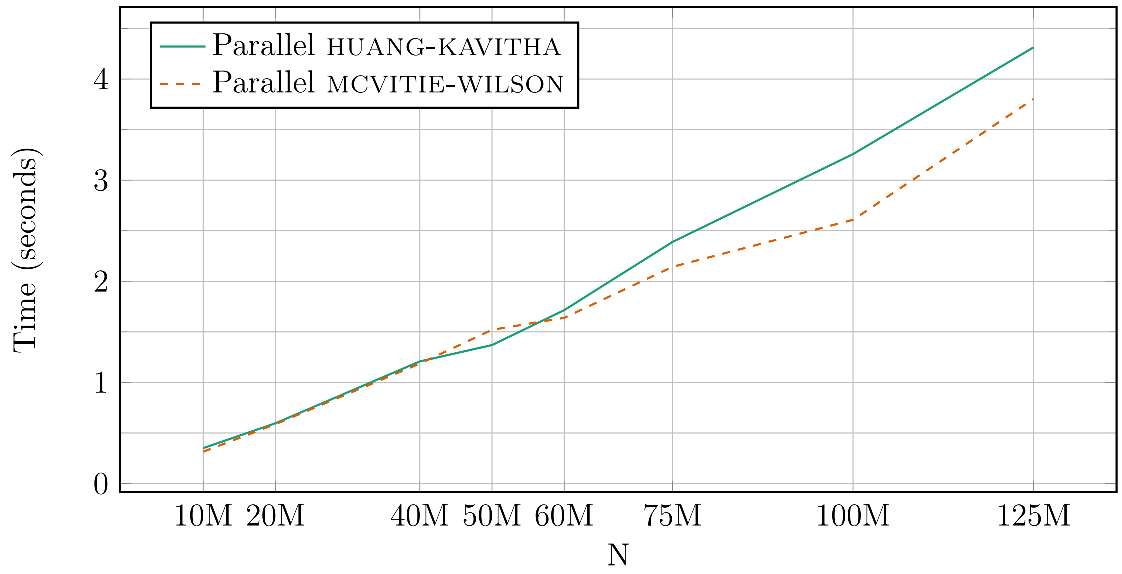

Figure 5.22: HUANG-KAVITHA time vs MCVITIE-WILSON time on easy instances (less is better)

100M and 125M instances.

On our hardware the single-width and double-width compare and swap operations perform with the same latency, so the increased data to compare and swap should not impact the speedup. This is also our observation in Figure 5.23 where the speedup is comparable to that of the stable marriage algorithms presented in Section 5.3.

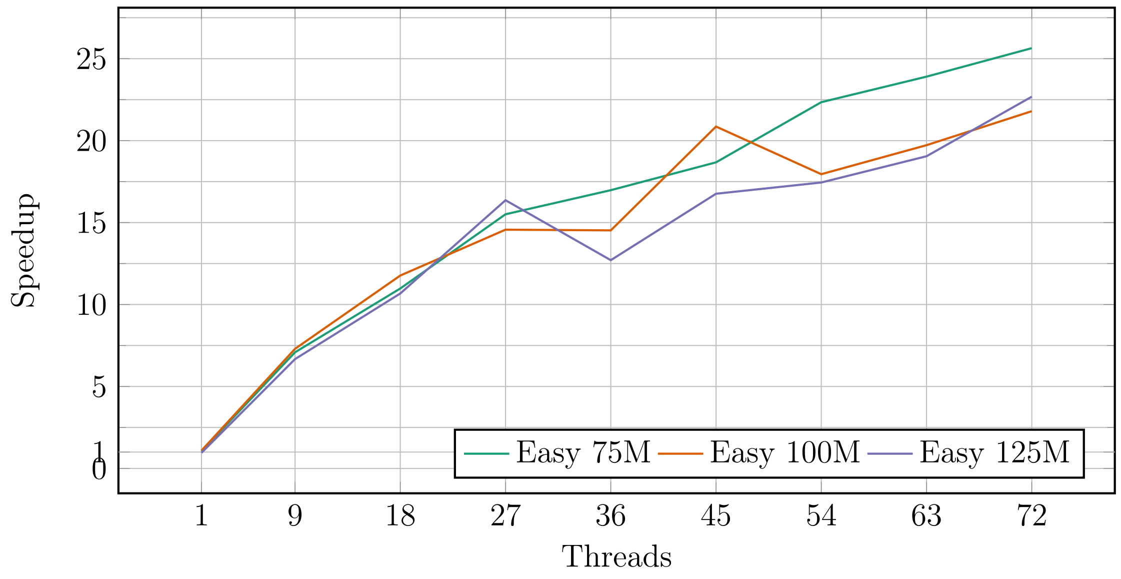

Figure 5.23: HUANG-KAVITHA speedup on easy instances (more is better)

Chapter 6

Conclusion

In this chapter we will first give a summary of the thesis, a short conclusion based on our experimental results, and then some thoughts about further improvements.

In the first three chapters we introduced the notion of matching, parallelising and preference. We then gave implementations to three different problems for matching under preference: the problem of stable marriage, the closely related b-marriage and popular matching problem. In chapter 4 we first presented some previous work on parallel algorithms for the stable marriage problem. We introduced our parallel algorithm for the previously presented problems based on ideas from the parallel SUITOR and b-SUITOR algorithms. In chapter 5 we presented runtime results for both the sequential and parallel algorithms showing scaling on all but the smallest instances.

The experimental results show that the ideas presented in the SUITOR and b-SUITOR transfer well to the parallel algorithms for matching under preferences presented in this paper.

Bibliography

[1] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1 – 25, 2011.

[2] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[3] David Gale and Marilda Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11(3):223 – 232, 1985.

[4] Chien-Chung Huang and Telikepalli Kavitha. Popular matchings in the stable marriage problem. *Information and Computation*, 222:180 – 194, 2013. 38th International Colloquium on Automata, Languages and Programming (ICALP 2011).

[5] M.Elizabeth C. Hull. A parallel view of stable marriages. *Information Processing Letters*, 18(2):63 – 66, 1984.

[6] Arif Khan, Alex Pothen, Md. Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM Journal on Scientific Computing*, 38(5):S593–S619, 2016.

[7] Jesper Larsen, Claus C. Carøe, and David Pisinger. A parallel approach to the stable marriage problem. pages 277–287, 1997.

[8] F. Manne, M. Naim, H. Lerring, and M. Halappanavar. *On Stable Marriages and Greedy Matchings*, pages 92–101. Society for Industrial and Applied Mathematics, 2017/05/31 2016.

[9] Fredrik Manne and Rob H. Bisseling. *A Parallel Approximation Algorithm for the Weighted Maximum Matching Problem*, pages 708–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] D. G. McVitie and L. B. Wilson. The stable marriage problem. *Communications of the ACM*, 14(7):486–490, July 1971.

[11] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, Oct 2015.

[12] S. S. Tseng and R. C. T. Lee. A parallel algorithm to solve the stable marriage problem. *BIT Numerical Mathematics*, 24(3):308–316, 1984.

Appendix A

Appendix

A.1 On Stable Marriages and Greedy Matchings

On Stable Marriages and Greedy Matchings

Fredrik Manne*, Md. Naim*, Håkon Lerring*, and Mahantesh Halappanavar†

Research on stable marriage problems has a long and mathematically rigorous history, while that of exploiting greedy matchings in combinatorial scientific computing is a younger and less developed research field. In this paper we consider the relationships between these two areas. In particular we show that several problems related to computing greedy matchings can be formulated as stable marriage problems and as a consequence several recently proposed algorithms for computing greedy matchings are in fact special cases of well known algorithms for the stable marriage problem.

However, in terms of implementations and practical scalable solutions on modern hardware, work on computing greedy matchings has made considerable progress. We show that due to the strong relationship between these two fields many of these results are also applicable for solving stable marriage problems. This is further demonstrated by designing and testing efficient multicore as well as GPU algorithms for the stable marriage problem.

1 Introduction

In 1962 Gale and Shapley formally defined the stable marriage problem and gave their classical algorithm for its solution [6]. Since then this field has grown tremendously with numerous applications both in mathematics and in economics. For a recent overview see the book by Manlowe [17]. Graph matching is a related area where the object is also to find pairs of entities satisfying various optimality criteria. These problems find a large number of applications. For an overview of some of these motivated from combinatorial scientific computing see [24].

While research on stable marriage problems has mainly focused on theory and mathematical rigor, work on graph matching in combinatorial scientific applications has had a larger practical component concerned with implementing and testing code on various computer architectures with the intent of developing fast scalable algorithms.

In this paper we investigate the connection between one type of matching problems, namely those of computing greedy weighted matchings, and algorithms for solving stable marriage problems. Although there exist exact algorithms for solving various weighted matching problems these tend to have running times that typically involve the product of the number of vertices and the number of edges. As large graph instances can contain tens of millions of vertices and billions of edges it is clear that such algorithms can easily become infeasible. For this reason there has been a strong interest in developing fast approximation algorithms and also in parallelizing these, see [19] and the references therein. Although such algorithms typically only guarantee an approximation factor of 0.5 compared to the optimal one, practical experiments have shown that they are very often only within a few percent from optimal. One such algorithm is the classical greedy algorithm applied to an edge weighted graph. Here edges are considered by decreasing weight and an edge is included in the matching if it is not adjacent to an already included edge.

The main contributions of this paper are as follows. Initially we consider implementation issues when designing efficient algorithms for the stable marriage problem. Next, we show that several recently published algorithms for computing greedy matchings are in fact special cases of classical algorithms for stable marriage problems. This also includes a generalization of the matching problem known as b-matching. Here each vertex can be matched with several other vertices in the final solution. Due to the strong similarities between the stable marriage problem and greedy matching, we show that one can apply recent results on designing scalable greedy matching algorithms to the computation of stable marriage solutions. This is verified by presenting efficient parallel implementations of various types of Gale-Shapley type algorithms for both multithreaded computers as well as for GPUs.

The remainder of the paper is organized as follows. In Section 2 we review the Gale-Shapley algorithm and consider implementation issues related to this. Next, in Section 3 we show that the computation of a greedy matching can be reformulated as a stable marriage problem. This is also shown to be true for greedy b-matching. In Section 4 we give parallel implementations of the Gale-Shapley and McVitie-Wilson algorithms and show their scalability, before concluding in Section 5.

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {fredrikm,naim}@ii.uib.no, hakon@lerring.no

†Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA. Email: mahantesh.halappanavar@pnnl.gov

2 The Stable Marriage Problem

In the following we describe the stable marriage (SM) problem and how it can be solved using the Gale-Shapley algorithm. We also consider some implementation issues related to the algorithm. Finally, we review some generalizations of the SM problem.

The SM problem is defined as follows. Let L and R be two equal sized sets $L = \{l_1, l_2, \ldots, l_n\}$ and $R = \{r_1, r_2, \ldots, r_n\}$. The entries in L are typically referred to as "men", while the entries in R are referred to as "women". Every man and woman has a total ranking of all the members of the opposite sex. These rankings give the "desirability" for each participant to match with a participant in the other set. The object is now to find a complete matching M (i.e. a paring) between the entries in L and R such that no two $l_i \in L$ and $r_j \in R$ both would obtain a higher ranked partner if they were to abandon their current partner in M and rematch with each other. Any solution satisfying this condition is *stable*.

Gale and Shapley [6] defined the stable marriage problem and also proposed the first algorithm for solving it. The algorithm operates in rounds as follows. In the first round each man in L proposes to his most preferred woman in R. Each woman will then reject all proposals except the one she has ranked highest. In subsequent rounds each man that was rejected in the previous round will again propose to the woman which he has ranked highest, but now disregarding any woman that he has already proposed to in previous rounds. Gale and Shapley showed that this process will terminate with each man in L being matched to a woman in R and that this solution is stable. Although an SM instance can have many stable solutions, the Gale-Shapley algorithm will always produce the same one.

An important variant of this problem is when each participant has only ranked a subset of the opposing participants. This problem is known as the stable marriage problem with incomplete lists (SMI). Any solution M to an SMI instance must then in addition to being stable, also consists of mutually ranked pairs (l_i, r_j). The SMI problem is solved by the Gale-Shapley algorithm but the solution might not be complete leaving some participants unmarried [6].

There exists a number of variants of the SM problem, for two comprehensive surveys see the books [10, 17]. In the following we will only consider the classical SM and SMI problems.

The original Gale-Shapley algorithm is described as operating in rounds, where only the men who were rejected in round t will propose in round $t + 1$. It is not stated in which order the proposals in a round should be made or what kind of data structures to use. If one traverses the men in L in their original order in each round and lets each rejected man propose once it is discovered, then this will ensure that the men always propose in the same relative order in each round. The running time of such a scheme is $\Theta(n^2)$ even for an instance of SMI.

If one is willing to forgo the requirement that the proposals in each round must be made in the same relative order then it is not hard to design an implementation of the Gale-Shapley algorithm with running time proportional to the number of actual proposals made. To do this one can maintain a queue Q of men waiting to make their proposals. Initially $Q = L$ and in each step of the algorithm the man at the front of the queue gets to propose to his current best candidate $r_j \in R$, and any rejected l_i is inserted at the end of the queue. This will ensure that all men rejected in round t gets to propose before any man rejected in round $t + 1$, but the relative order among the men might not always be the same. The algorithm terminates when the queue is empty.

One simple enhancement one can make to the Gale-Shapley algorithm is that no $l_i \in L$ should propose to an $r_j \in R$ who already has a proposal from someone whom r_j ranks higher than l_i, as such a proposal will be rejected. Thus each l_i should propose to his most preferred r_j where l_i has not already been rejected and where r_j ranks l_i higher than her current best proposal (if any). In terms of implementation this means that it is sufficient to only maintain the current best proposal for each r_j. When the algorithm terminates these proposals will make up the solution. We give our complete implementation of the Gale-Shapley algorithm in Algorithm 1.

In Algorithm 1 each r_j has a variable $suitor(r_j)$ initialized to $NULL$ that holds her current best proposal. Similarly, $ranking(r_j, l_i)$ returns r_j's ranking of l_i (as a number in the range 1 through n). We define $ranking(r_j, NULL) = n + 1$ to ensure that any proposal is better than no proposal. The function $nextCandidate(l_i)$ will initially return l_i's highest ranked woman and then for successive calls return the next highest ranked one following the last one retrieved.

For an SM instance it is straight forward to precompute the values of $ranking()$ in $O(n^2)$ time. However, for an SMI instance maintaining a complete $ranking()$ table would require $O(n^2)$ space and also proportional time to initialize it. In this case it is more efficient to store the value of $ranking(r_i, l_j)$ together with r_i in l_j's ranking list so that it can be fetched in $O(1)$ time when needed. These values can be precomputed in time proportional to the sum of the lengths of the ranking lists. To do this one first traverses the women's lists building up lists for each man l_j with the women that have ranked him along with in what position. Then using an array $position()$ of length n initially set to 0, the list of each man l_j is processed as follows. For each woman r_i that has ranked l_j we store the value l_j along with in what position r_i has ranked l_j in $position(r_i)$. We next traverse l_j's priority list and for each r_i in the list we look up $position(r_i)$ and see if it contains l_j. If so, we fetch r_i's ranking of l_j and store it together with l_j's ranking of r_i. At the same time any r_i that has not ranked l_j but which l_j has ranked can be purged from the priority list of l_j.

Algorithm 1 The Gale-Shapley algorithm using a queue

```
1: $Q = L$
2: while $Q \neq \emptyset$ do
3:     $u = Q.dequeue()$
4:     $partner = nextCandidate(u)$
5:     while $ranking(partner, u) >$
       $ranking(partner, suitor(partner))$ do
6:         $partner = nextCandidate(u)$
7:     if $suitor(partner) \neq NULL$ then
8:         $Q.enqueue(suitor(partner))$
9:     $suitor(partner) = u$
```

McVitie and Wilson [5] gave a recursive implementation of the Gale-Shapley algorithm. This algorithm also iterates over the men, allowing each one to make a proposal to his most preferred woman. But if this proposal is rejected or if it results in an existing suitor being rejected then the algorithm recursively lets the just rejected man make a new proposal to his best remaining candidate. The recursion continues until a proposal is made such that no man is rejected (because the last proposed to woman did not already have a suitor). At this point the algorithm will continue with the outer loop and process the next man. When all men have been processed the algorithm is finished. It is shown in [5] that the McVitie-Wilson algorithm produces the same solution as the Gale-Shapley algorithm. We note that similarly to the Gale-Shapley algorithm it is possible to speed up the algorithm by avoiding proposals that are destined to be rejected because the proposed to woman already has a better offer.

Comparing the two algorithms each man will consider exactly the same women before ending up with his final partner. The only difference is the order in which this is done. While the Gale-Shapley algorithm will maintain a list of men that needs to be matched, the McVitie-Wilson algorithm will always maintain a solution where each man considered so far is matched before including a new man in the solution.

We note that one can implement a non-recursive version of the McVitie-Wilson algorithm simply by replacing the queue Q in Algorithm 1 by a stack and replacing the $dequeue()$ and $enqueue()$ operations with $pop()$ and $push()$ operations respectively. To see that this will result in the McVitie-Wilson algorithm it is sufficient to first note that the initial placement of L in Q is equivalent to an outer loop that processes each man once. Any rejected man will then be placed at the top of the stack and therefore be processed immediately, similarly to a recursive call in the original algorithm.

Wilson [29] showed that for any profile of womens preferences, if the men's preferences are random, then the expected sum of men's rankings of their mates as assigned by the Gale-Shapley algorithm is bounded above by $n(1 + 1/2 + ... + 1/n)$. Knoblauch [16] showed that this is also an approximate lower bound in the sense that the ratio of the expected sum of men's rankings of their assigned mates and $(n + 1)((1 + 1/2 + ... + 1/n) - n)$ has limit 1 as n goes to ∞. Thus if the men's preferences are random then this sum is $\Theta(n \ln n)$ for large n. However, it is not hard to design instances where this sum is $\Theta(n^2)$. This is for instance true for any SM instance where the men have identical preferences.

2.1 Generalizations of SM We next review two generalizations of the SM problem. In the stable roommates (SR) problem one is given a set of n persons, each one with a complete ranking of all the others persons. The objective is now to pair two and two persons together, i.e. make them roommates, such that there is no pair (x, y) of persons where x is either unmatched or prefers y to its current partner, while at the same time y is either unmatched or prefers x to its current partner. Just like for the SM problem, such a solution is called stable. Unlike the SM problem there might not exist a solution for an SR instance. We also note that if some persons have only ranked a subset of the other participants we get the stable roommates problem with incomplete lists (SRI). Irving gave an algorithm for computing a stable solution to an SRI problem or to determine that no such solution exists [12]. This algorithm operates in two stages, where the first one is similar to the Gale-Shapley algorithm where each person makes, accepts and rejects proposals. The second phase of the algorithm is slightly more involved but does not change the running time of $O(n^2)$. For more information on the SR and SRI problems see [10, 17].

In the last generalization each person can be matched with more than one person in a final solution. More formally, we are looking for a stable solution to an SM, SMI, SR, or SRI instance where each person v_i is matched with at most $b(v_i)$ other persons, where $b(v_i) \geq 1$. Being stable again means that no two persons v_i and v_j would both obtain a better solution if they were to match with each other, either by dropping one of their current partners or if v_i has fewer than $b(v_i)$ partners or if v_j has fewer than $b(v_j)$ partners.

For the SM problem this gives us the many-to-many stable assignment problem (MMSA), where each "man" and "woman" can be matched with several participants of the opposite sex. This was solved by Baïou and Balinski [2] who presented a general algorithm based on modelling this as a graph searching problem. See also [3] for how to model the SM problem as a graph problem [3].

Applying the last generalization to an SR instance, we get the stable fixtures problem [13] for which Irving and Scott gave an $O(n^2)$ algorithm. Similarly to Irving's algorithm for SRI, this algorithm also consists of two stages, where the first stage is a natural extension of the Gale-Shapley algorithm to handle that each person can participate in multiple matches.

3 Matching Problems

We next explore the relationship between stable marriages and weighted matchings in graphs. A matching M on a graph $G(V, E)$ is a subset of the edges such that no two edges in M share a common end point. For an unweighted graph the object is to compute a matching of maximum cardinality. For an edge weighted graph a typical problem is to compute a matching M such that the sum of the weights of the edges in M is maximum over all matchings. Another variant could be to compute the maximum weight matching over all matchings of maximum cardinality.

We consider the GREEDY algorithm for computing a matching of maximum weight in an edge weighted graph where all weights are positive. This algorithm considers edges by decreasing weight. In each step the heaviest remaining edge (u, v) is included in the matching before removing any edge incident on either u or v. If the weights of the edges in G are unique or if a consistent tie breaking scheme is used then it follows that the solution given by GREEDY is also unique. In the following we will always assume that this is the case. It is well known that GREEDY guarantees a solution of weight no worse than 0.5 times the weight of an optimal solution. We label the problem of computing a greedy matching in an edge weighted graph as the GM problem.

Given an instance G of the GM problem one can construct an equivalent instance of the SRI problem by sorting the edges incident on each vertex u by decreasing weight, and letting this be the ranking of u's neighbors in the SRI instance. It is not hard to see that with this construction, a solution to the GM problem is equivalent to a stable solution of the corresponding SRI problem. Consider the heaviest edge (u, v) in the graph. This is included in the GM solution and the corresponding vertex pair must also be part of any solution to the SRI instance, otherwise this solution would not be stable as both u and v would prefer to match with each other over any other partner. We can thus include (u, v) in the solutions to both instances and also remove u and v from further consideration. For the GM problem this means that any edges incident on either u or v are removed and for the SRI instance u and v are removed from all ranking lists. One can then repeat the argument using the heaviest remaining edge, and it thus follows by induction that the two solutions are identical. It is also clear that the corresponding SRI instance always has a unique stable solution.

The above construction implies that the solution given by GREEDY is stable in the sense that there does not exist an edge $(u, v) \notin M$ such that the weight of M would increase if (u, v) was added to M while removing any edges incident on either u or v from M. This observation is often stated as that the solution given by GREEDY does not contain any augmenting path containing three or fewer edges. An augmenting path of length k is a path containing k edges starting with an edge in M and then alternating between edges not in M and edges in M, with the property that if one was to replace all the edges on the path that belong to M with those that are not in M then the weight of the solution would increase.

We next proceed to show that the solution given by GREEDY can also be obtained by solving an associated SMI (or SM) instance. To the best of our knowledge this result has not been shown previously.

Given an instance of the GM problem on an edge weighted graph $G(V, E)$. We define an SMI instance G' from G as follows. Let L and R be the sets of men and women respectively, both of size $n = |V|$. Any man l_i will include exactly those r_j in its ranking where there is an edge $(v_i, v_j) \in E$. As the edges in G are not directed, this also means that l_j will rank r_i. Similarly, any woman r_j will include exactly those l_i in her ranking where there is an edge $(v_i, v_j) \in E$. Both men and women order their lists by decreasing weight of the corresponding edges in G. Thus every $(v_i, v_j) \in E$ gives rise to four rankings in G'. We call the two pairs (l_i, r_j) and (l_j, r_i) for the corresponding pairs of (v_i, v_j).

LEMMA 3.1. *Given a graph G with SMI instance G' as described above. Let further M be the greedy matching on G. Then the pairs in G' corresponding to the edges in M make up the unique solution to the SMI problem on G'.*

Proof. The proof is by induction on the edges of M considered by decreasing weight. Let (v_i, v_j) be the edge of maximum weight in G. Then $(v_i, v_j) \in M$ and it also follows from the construction of G' that l_i will rank r_j highest. Similarly, l_i will also have the highest ranking among the men ranked by r_j. Thus (l_i, r_j) must be included in any stable solution of G'. A similar argument shows that the edge (l_j, r_i) will also be included in such a solution.

Assume now that the pairs in G' corresponding to the $k \geq 1$ heaviest edges in M must be included in any stable solution and consider the two pairs (l_s, r_t) and (l_t, r_s) corresponding to the $k + 1$st heaviest edge (v_s, v_t) in M.

It is clear that any solution where l_s is matched to a woman that he has ranked after r_t while at the same time r_t is matched to a man that she has ranked after l_s, cannot be stable as both l_s and r_t would be better of if they were to match with each other. Thus if (l_s, r_t) is not included in a stable solution at least one of l_s and r_t must be matched to a partner which he or she has ranked higher than the other one of $\{l_s, r_t\}$. Assume therefore that l_s is matched to r_u and that l_s has ranked r_u higher than r_t, implying that the weight of (v_s, v_u) is greater than the weight of (v_s, v_t) in G. But since $(v_s, v_t) \in M$ it follows that $(v_s, v_u) \notin M$. Thus v_u must be matched to some other vertex v_z in M. And since $(v_s, v_u) \notin M$ the weight of (v_u, v_z) must be greater than that of (v_s, v_u). By the induction hypothesis the pairs in G' that correspond to the k heaviest edges in M must be included in any stable solution in G'. It therefore follows that l_z must be matched to r_u in any stable solution on G' contradicting that l_s is matched to r_u. A similar argument shows that r_t cannot be matched to any man in L to which she gives higher priority than she gives to l_s. Thus the pair (l_s, r_t) must be in any stable solution in G'. The argument for why (l_t, r_s) also must be included in a stable solution is analogous. It follows that any pair in G' corresponding to an edge in M must be part of a stable marriage in G'.

It only remains to show that once the pairs corresponding to the edges in M have been included in the solution M' to G', then it is not possible to match any other pairs in G'. If M' contains a pair (l_i, r_j) in addition to the pairs corresponding to the edges in M then $(v_i, v_j) \notin M$ and neither v_i nor v_j can be matched in M. But since l_i has ranked r_j (and vice versa) it follows that $(v_i, v_j) \in E$ and that M can be expanded with (v_i, v_j). This contradicts that M is maximal and the result follows.

We next consider the b-matching problem which is a generalization of the regular weighted matching problem similar to the many-to-many stable assignment problem and the stable fixtures problem. A b-matching on G is a subset of edges $M \subseteq E$ such that every vertex $v_i \in V$ has at most $b(v_i)$ edges in M incident on it. The objective is to compute the b-matching of maximum weight. A 0.5 approximation can again be computed using the greedy algorithm that selects edges by decreasing weight and whenever $b(v_i)$ edges incident on v_i have been selected, the remaining edges incident on v_i are removed [20]. Setting $b(v_i) = 1$ for all $v_i \in V$ gives a regular (one) matching. See [21] for a survey of algorithms for computing optimal b-matchings.

It is straight forward to see that the stable fixtures problem is also a generalization of greedy b-matching. Given an instance of the greedy b-matching problem, one can also construct an equivalent many-to-many stable assignment instance by setting the bounds $b(l_i)$ and $b(r_i)$ equal to $b(v_i)$. A proof similar to that of Lemma 3.1 shows that these two problems have equivalent solutions.

3.1 Algorithmic Similarities As a consequence of the fact that the solution given by GREEDY can be obtained by either solving a properly designed instance of SMI or SRI, any algorithm that solves either of these two problems can also be used to compute a greedy weighted matching. This process can be simplified as it might be possible to run an SMI or SRI algorithm directly on the original graph. Let G be an instance of GM and G' its corresponding SMI instance. Also let $\{r_{s_1}, r_{s_2}, \ldots, r_{s_f}\}$ and $\{l_{t_1}, l_{t_2}, \ldots, l_{t_g}\}$ be the ranked lists of the man l_i and the woman r_i respectively. Then if follows from the construction of G' that $f = g$ and that $s_k = t_k$ for all k. Thus any proposal made to r_i could be handled directly by l_i as he has the same information as r_i. It follows that one can merge l_i and r_i into one node v_i that handles making, accepting, and rejecting proposals related to l_i and r_i. In this way both the Gale-Shapley and the McVitie-Wilson algorithm can be used directly on edge weighted general graphs to compute greedy matchings, but now using edge weights to rank potential partners.

Irving's algorithm [12] for solving the SRI problem consists of two stages, of which the first is exactly this algorithm of making, accepting, and rejecting proposals on a general graph. If the rankings in an SRI instance are based on edge weights from a GM instance then the first phase will produce the greedy solution which is stable, thus making the second phase of the algorithm redundant.

Previous efforts at designing fast parallel greedy matching algorithms have been based on the notion of dominant edges. These are edges that are heavier than any of their neighboring edges. Preis showed that an algorithm based on repeatedly including dominant edges in the matching while removing any edges incident on these will result in the same solution as GREEDY [25]. Based on this observation Manne and Bisseling developed the pointer algorithm [18], which was further enhanced by Manne and Halappanavar in the SUITOR algorithm [19]. We note that the SUITOR algorithm is identical to the McVitie-Wilson algorithm applied to a general edge weighted graph, while the pointer algorithm has strong resemblances to the Gale-Shapley algorithm as outlined in Algorithm 1.

The same type of relationship also holds true between the greedy b-matching problem and the many-to-many stable assignment problem. The algorithm presented in [2] can be instantiated to solve the b-matching problem using a Gale-Shapley type algorithm where a vertex v will accept the $b(v)$ best offers at any given time. We note that this is the same algorithm as the one presented by Georgiadis and Papatriantafilou [8] and also by Khan et al. [14] for computing a greedy b-matching. In [14] the authors experiment with what they call *delayed* versus *eager* rematching of rejected suitors. The difference between these two variants is the same as that between a Gale-Shapley and a McVitie-Wilson style algorithm.

4 Experiments

As shown in Section 3 much of the theory that has been developed lately for greedy matching algorithms are mainly restricted versions of previous results from the theory of stable marriages. However, the work on greedy matchings has to a large extent been driven by a need for developing scalable parallel algorithms for use in scientific applications.

This has lead to the implementation of Gale-Shapley and McVitie-Wilson type matching algorithms on a large variety of architectures, including distributed memory machines [4, 18], multicore computers [11, 15, 19], and GPUs [1, 23].

There has been less emphasis on implementations and developing working code for the stable marriages problem. We believe that much of the work done on greedy matchings can easily carry over to developing efficient code for stable marriage problems. To show the feasibility of this we have developed shared memory implementations of both the Gale-Shapley and McVitie-Wilson algorithms. We used OpenMP to parallelize the Gale-Shapley algorithm and both OpenMP and CUDA for parallelizing the McVitie-Wilson algorithm.

In weighted matching both endpoints of an edge (u, v) evaluates the importance of the edge to the same number, i.e. the weight of the edge. Whereas in the stable marriage problem both u and v assign their own ranking of the other. Thus the main difference between greedy matching algorithms such as those presented in [18, 19] and the Gale-Shapley algorithm is that in the latter, a man who makes a proposal evaluates his chance of success based on the woman's ranking, instead of on a common value. Another difference is that it is typically not assumed in weighted matching problems that the neighbor list of a vertex is sorted by decreasing weight. It was shown in [19] that when this is the case, then it both simplifies the algorithm and also speeds up the execution considerably.

Our parallelization strategy for the McVitie-Wilson algorithm using OpenMP closely follows that of the SUITOR algorithm as presented in [19], while our CUDA version of the same algorithm is a simplified version of the SUITOR algorithm used in [23]. In both of our OpenMP algorithms the set of men is initially partitioned among the threads who then each run a local version of the corresponding algorithm until completion. A thread will first search the list of the current man to locate the woman he gives highest priority and where the woman also prefers him to her current suitor (if any). If such a woman is discovered the thread will use a compare-and-swap (CAS) operation to become the new suitor of the woman. In this way it is assured that no other thread has changed the suitor value. If the CAS operation succeeds the previous suitor (if any) is treated according to the current strategy and is inserted in a local stack (McVitie-Wilson) or a local queue (Gale-Shapley). If the CAS operation failed because some other thread had already changed the suitor value, then if the current man can still beat the new the suitor then the thread will retry with a new CAS operation, otherwise it will continue searching for the nest eligible woman.

There is a difference between the algorithms in how they can handle load imbalance. For the parallel Gale-Shapley algorithm it is possible to synchronize the threads after each round of proposals and then redistribute the unmarried men to the threads before moving on to the next round. However, synchronization tends to be costly, and experiments done on greedy matching problems indicate that this is typically not worth the effort. For the McVitie-Wilson algorithm one can load balance the algorithm by using one of the dynamic load balancing strategies in OpenMP such as *dynamic* or *guided* in the initial assignment of men to threads. This strategy was used successfully in experiments for the SUITOR matching algorithm on graphs with highly varying vertex degrees [19].

For the CUDA algorithm we assign one thread to each man in our implementation of the McVitie-Wilson algorithm. Each thread then executes the algorithm similarly to the OpenMP version using a CAS operation to assign a man as the suitor of a particular woman. Using only one thread per man allows for a larger number of thread blocks which the runtime environment can balance across the device. But as the threads within one physical warp operate in SIMD, the running time of all threads in the same warp will be equal to the maximum execution time of any of the threads. Similarly, the threads within the same thread block will not release resources until all threads in the block have finished executing. It would have been possible to statically assign multiple men to each thread or to design a dynamic load balancing scheme with the aim of evening out the work load. But this would have resulted in a more complicated algorithm and as our main goal is proof of concept we did not pursue this.

Implementing the Gale-Shapley algorithm on the GPU presents additional challenges compared to the McVitie-Wilson algorithm. In a Gale-Shapley algorithm the threads would have to be grouped so that each thread group operates on one common queue, where the size of the group could be either a subset of threads in a warp or all the threads in one thread block. As the number of free men monotonically decreases between rounds, there should initially be more men than threads assigned to the same queue, something that would complicate the algorithm. Also, having several threads operate on one common queue would require synchronization which can be time consuming on the GPU. For this reason we chose not to implement the Gale-Shapley algorithm using CUDA.

As we are not aware of any sufficiently large publicly available data sets for the stable marriage problem, we have designed two different random data sets. The first set has been constructed to be relatively easy to solve, whereas the second set is intended to be more time consuming. We label these sets as *easy* and *hard* respectively. Each instance consists of n men and n women. In the *easy* data set each man is assigned a random number $\epsilon \in [0, 1]$ and then randomly picks and ranks $(1 + \epsilon) \ln n$ women. Each woman then ranks exactly the men that ranks her. With this configuration more than 98% of the participants were matched in every final solution and the total number of proposals is at most $2n \ln n$ with an average of $n \ln n$. In the *hard* data set each man has an identical complete random ranking of all the women. Similarly, all woman share the same random ranking of all the men. Thus there will always exist a complete stable solution and the total number number of considered women will always be $n(n+1)/2$. Moreover, in the *hard* instances there will be contention among the men for obtaining the same set of women, and thus cause substantial synchronization requirements for the parallel algorithms. One obvious difference between the datasets is that the *easy* instances will require more memory access as each participant has an individual ranking list, while for the *hard* instances all rankings are stored in two shared vectors of length n.

For each value of n we have generated 5 instances and for each of these we run each algorithm 3 times. For all timings we take the average of these 15 runs.

The OpenMP algorithms are run on a computer with two Intel Xeon E5-2699 processors and 252 Gbytes of memory. Each processor has 18 cores and runs at 2.30GHz. As the machine can use hyperthreading it is possible to use up to 72 threads. However, going beyond 36 threads did not give any additional speedup and thus we restrict our reporting to maximum 36 threads. The GPU is a Tesla K40m with 12 GB of memory, 2880 cores running at 745 MHz and has CUDA compute capability version 3.5.

In Figure 1 we present results from the *easy* instances when n varies from 5M up to 25M in steps of 5M. For the OpenMP algorithms the number of threads is set to 36. For most of these instances the running time stays well below one second. It is only for the $n=$25M instance that the GPU algorithm uses slightly more time than one second. This is also the largest *easy* instance that could be run on the GPU.

For smaller instances the GPU algorithm is the fastest one but as the problem size increases it is slowing down compared to the OpenMP algorithms. In general the OpenMP Gales-Shapley algorithm is faster than the OpenMP McVitie-Wilson algorithm with as much as 12%. For this setup one would expect that the graph displaying the time would resemble $n \ln n$ as the computing resources is the same for each instance. This is most true for the OpenMP algorithms where the time increases close to linearly with n, whereas the time grows faster than n for the GPU algorithm. This can be seen further in Figure 2 which shows the speedup of both McVitie-Wilson algorithms compared to their sequential counterpart. The OpenMP algorithm gives a constant speedup of about 9, while the speedup of the GPU algorithm starts out at about 18 but then drops sharply as the size of the instances increase. Thus this is most likely due to insufficient memory on the GPU.

Figure 3 shows running times of the OpenMP algorithms using 36 threads as n increases up to 125M. It can be observed that the tendencies for the smaller instances still

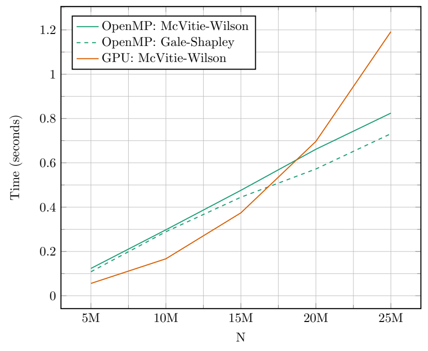

Figure 1: Running time on the *easy* dataset

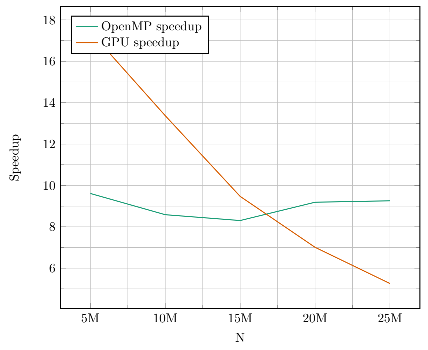

Figure 2: Speedup on the *easy* dataset

remain true for the larger ones. We note that the worst running time is only marginally larger than four seconds on the largest instance. Figure 4 shows the speedup of the OpenMP algorithms compared to the sequential Gale-Shapley algorithm for the three largest instances when using $t = 1, 9, 18, 27$ and 36 threads. The Gale-Shapley algorithm outperforms the McVitie-Wilson algorithm in almost all instances and reaches a speedup of approximately 15 on the $n = 75$M instance. We note that the relative performance of the two algorithms could have been different if there was a more uneven load balance between the threads.

Figure 5 shows the running time on *hard* instances where n increases from 100K up to 500K. These problems measure how well the algorithms and hardware can handle contention for shared resources. As the dataset only consists of two vectors we can run the problems using all three codes, the only limiting factor being time. Since the total amount of work grows as $O(n^2)$ on these instances it is to be expected that they will require more time than the *easy* ones. From

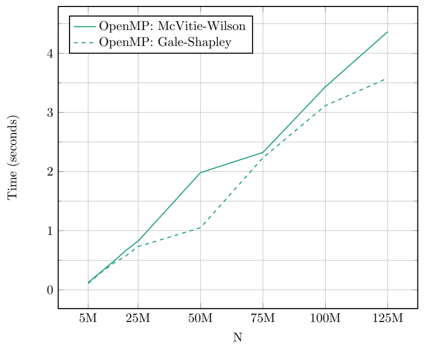

Figure 3: Running time of OpenMP algorithms on large *easy* instances

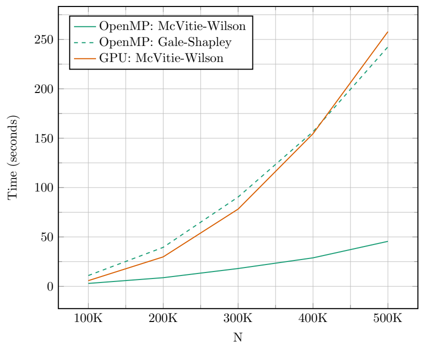

Figure 5: Running time on *hard* datasets

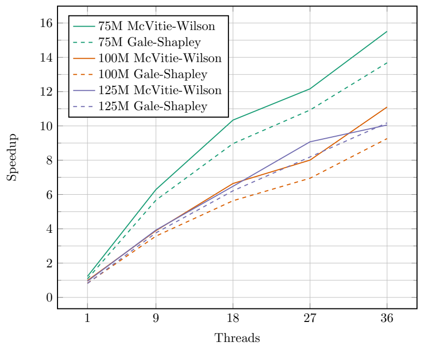

Figure 4: Speedup on large *easy* datasets

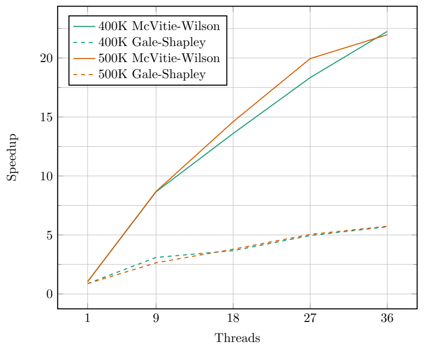

Figure 6: Speedup on *hard* datasets

the figure it can be observed that there is little difference in the running time between the Gale-Shapley OpenMP code and the McVitie-Wilson GPU code, which both take close to 250 seconds on the largest instance. However, the McVitie-Wilson OpenMP code performs considerably better, and is a factor of 5 times faster on the largest instance. This difference is also displayed in Figure 6 which gives the speedup of the same instances.

We believe that the difference between the OpenMP algorithms can be explained by how the algorithms handle the large number of rejections. While the Gale-Shapley algorithm has to store each rejected man to memory and retrieve a new one, the McVitie-Wilson algorithm can continue working on the rejected man without needing to access relatively slow memory. The poor performance of the McVitie-Wilson GPU algorithm compared to the OpenMP one is most likely due to how the machines handle contention for shared resources. The GPU algorithm utilizes several thousand concurrent threads that, at least initially, will be competing for matching their man with the same set of women. Synchronizing this will lead to a much larger strain on the system compared to the OpenMP algorithm where there is never more than 36 concurrent threads.

Finally, in figures 7 and 8 we show the number of considered proposals per second for both *easy* and *hard* datasets on the OpenMP algorithms. For each instance this number is given as the sum over each man of his ranking of his final partner and then divided by the total time. In sparse graph algorithms this is often referred to as the number of traversed edges per second (TEPS) and is, among other things, used to rank the performance of computers in the Graph500 challenge [9, 22].

For the *easy* instances the TEPS rate starts out at 10M for one thread and then increases to somewhere between 100M to 140M for 36 threads. Thus the efficiency when using 36 threads lies somewhere in the range of 30% to 40%. For the *hard* instances the TEPS rate for the McVitie-

Wilson algorithm starts out at about 150M and increases up to 2.75 billion when using 36 threads for an efficiency rate of about 50%. As already noted the Gale-Shapley algorithm does not scale well on these instances. Comparing the TEPS rate between the *easy* and the *hard* instances when using the McVitie-Wilson algorithm on the same number of threads it can be observed that the maximum TEPS rate is more than a factor of 20 larger for the *hard* instances. This is most likely because the *hard* instances are not limited by access to memory as the whole dataset only consists of two vectors.

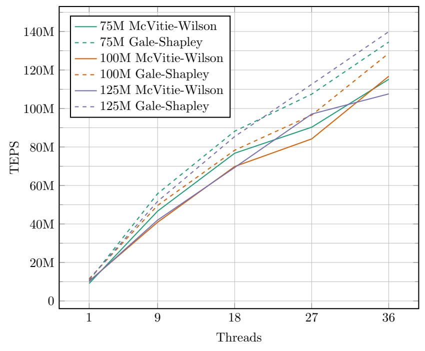

Figure 7: TEPS for *easy* datasets

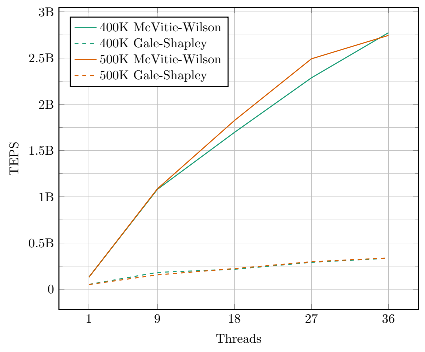

Figure 8: TEPS for *hard* datasets

5 Conclusion

In his book Manlove [17] lists some of the most noteworthy open problems related to SM. One of these is to determine if the SM problem is in the complexity class NC or not, that is, to determine whether the problem can be solved by an algorithm with polylogarithmic running time when using a polynomial number of processes. Efforts at designing such algorithms has mainly resulted in parallel algorithms requiring at least n^2 processes, and are thus mainly of theoretical interest [7, 28].

We are only aware of one previous attempt at implementing a parallel version of the Gale-Shapley algorithm and this did not result in any speedup [27]. Quinn [26] argues that one cannot expect a large speedup from a parallel Gale-Shapley style algorithm in practice as the algorithm cannot run faster than the maximal number of proposals made by any one man. We note that for a random instance the average number of proposals made by each man is in fact $O(\log n)$.

While the question of developing asymptotically faster parallel algorithms than those presented in Section 4 is of interest from a theoretical point of view, we believe that this is less relevant for a practitioner. To begin with the running time of the Gale-Shapley algorithm is linear in the instance size. Thus moderate sized problem can already be solved rapidly. In addition, our current experiments on the SMI problem as well as previously experiments on GM problems shows that Gale-Shapley type algorithms scale well. One reason for this is that the size of the instance n is typically much larger than the number of threads or processes used.

One notable difference between the formulations of the GM and the SMI problem is that for GM it is not assumed that the neighbor lists are initially sorted by decreasing weight in the same way as priority lists are ordered in SM. Thus work on developing parallel algorithms for the GM problem has focused on how one should search the neighbor lists. Suggested solutions include sorting the lists initially, searching through the list each time a new candidate is needed, or something in between. All of these strategies result in a running time that is superlinear in the input size. However, Preis's algorithm for GM has linear running time [25], but is more complicated and not suitable for parallel execution. We therefore ask if it is possible to design a linear time algorithm for the SM problem if the priority lists are not sorted, but instead given as real valued numbers such that $p_i(j)$ gives the value that person i assigns to person j of the opposite sex.

References

[1] B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the multicore, Challenge II*, volume 7174, pages 108–119. LNCS, Springer, 2012.

[2] M. Baïou and M. Balinski. Many-to-many matching: stable polyandrous polygamy (or polygamous polyandry). *Discrete Applied Mathematics*, 101(1-3):1–12, 2000.

[3] M. Balinski and G. Ratier. Of stable marriages and graphs, and strategy and polytopes. *SIAM Review*, 39(4):575–604, 1997.

[4] Ü. V. Çatalyürek, F. Dobrian, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel al-

gorithms for matching and coloring. In *IPDPS Workshops*, pages 1971–1980, 2011.

[5] L. B. Wilson D. G. McVitie. The stable marriage problem. *Communications of the ACM*, 14(7):486–490, 1971.

[6] L. S. Shapley D. Gale. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[7] T. Feder, N. Megiddo, and S. A. Plotkin. A sublinear parallel algorithm for stable matching. *Theor. Comput. Sci.*, 233(1-2):297–308, 2000.

[8] G. Georgiadis and M. Papatriantafilou. Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists. *Algorithms*, 6(4):824–856, 2013.

[9] Graph 500. `http://www.graph500.org`.

[10] D. Gusfield and R. W. Irving. *The stable marriage problem*. The MIT press, 1989.

[11] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perf. Comput. App.*, 26(4):413–430, 2012.

[12] R. W. Irving. An efficient algorithm for the "stable roommates" problem. *J. Algorithms*, 6(4):577–595, 1985.

[13] R. W. Irving and S. Scott. The stable fixtures problem - A many-to-many extension of stable roommates. *Discrete Applied Mathematics*, 155(16):2118–2129, 2007.

[14] A. Khan, A. Pothen, M. M. A. Patwary, N. R. Satish, N. Sundaram, F. Manne, M. Halappanavar, and P. Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM J. Sci. Comput.*

[15] A. M. Khan, D. F. Gleich, A. Pothen, and M. Halappanavar. A multithreaded algorithm for network alignment via approximate matching. In *SC*, page 64, 2012.

[16] V. Knoblauch. Marriage matching: A conjecture of Donald Knuth. Economics Working Papers, http://digitalcommons.uconn.edu/econ_wpapers/200715, 2007.

[17] D. Manlove. *Algorithmics of matching under preferences*. World Scientific, 2013.

[18] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *PPAM'08*, volume 4967 of *LNCS*, pages 708–717. Springer, 2008.

[19] F. Manne and M. Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 519–528, 2014.

[20] J. Mestre. Greedy in approximation algorithms. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2006.

[21] M. Müller-Hannemann and A. Schwartz. Implementing weighted b-matching algorithms: Insights from a computational study. *J. Exp. Algorithmics*, 5, December 2000.

[22] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. Cray User's Group (CUG), 2010.

[23] Md. Naim, F. Manne, M. Halappanavar, A. Tumeo, and J. Langguth. Optimizing approximate weighted matching on nvidia kepler K40. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 105–114, 2015.

[24] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. CRC Press, 2012.

[25] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS'99*, volume 1563, pages 259–269. LNCS, Springer, 1999.

[26] M. J. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, 1987.

[27] J. L. Träff. A parallel approach to the stable marriage problem. In *Proceedings of the Nordic Operations Research Conference (NOAS '97)*, pages 277–287, 1997.

[28] C. White and E. Lu. An improved parallel iterative algorithm for stable matching. Extended Abstract, Companion of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing 2013).

[29] L. B. Wilson. An analysis of the marriage matching assignment algorithm. *BIT*, 12:569–575, 1972.