A Parallel Approach to the Stable Marriage Problem

Jesper Larsen
DIKU – Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK 2100 Copenhagen Ø
e-mail:friberg@diku.dk

February 3, 1997

Abstract

This paper describes two parallel algorithms for the stable marriage problem implemented on a MIMD parallel computer. The algorithms are tested against sequential algorithms on randomly generated and worst-case instances. The results clearly show that the combination fo a very simple problem and a commercial MIMD system results in parallel algorithms which are not competitive with sequential algorithms wrt. practical performance.

1 Introduction

In 1962 the *Stable Marriage Problem* was introduced by *David Gale* and *Lloyd Shapley* in a paper entitled "College Admissions and the Stability of Marriage". Informally, a stable marriage is a one-to-one matching of a set of men to a set of women, such that for any pair of a man and a woman, not both prefer the other to their assigned partner.

This paper describes the development of two parallel algorithms for the stable marriage problem. In section 2 the basic concepts of the stable marriage problem are introduced. In section 3 two sequential algorithms are described and in section 4 we describe the parallel algorithms. Section 5 contains the experimental results. Th paper is based on [Lar94] in which all details of the algorithms and their implementations are described.

The constructed programs (sequential as well as parallel) are available on request. The sequential programs are written in C, while the parallel programs additionally use the CSTools communication primitives [Mei92]. Also available are the generator programs for generating worst-case instances and randomly generated instances.

2 Basic concepts and notation

Formally, an instance of the stable marriage problem of size n consists of two disjoint sets M and W of size n. Throughout this paper we will let n denote the size of the problem, even though the problem size is actually n^2, as the input is two $n \times n$-matrices. The set M is called the *set of men*, and W the *set of women*. Associated with each *person*, that is each member in M and W, is a *preference list*. The preference list for a man is a strictly ordered list of **all** the women, and vice versa.

Given a problem-instance a *matching* $\mathcal{M}$ is a pairing of a man m to a woman w. If m and w are matched in the matching $\mathcal{M}$, we call m and w *$\mathcal{M}$-partners* (or *partners* in $\mathcal{M}$ or just *partners*), and write $w = p_{\mathcal{M}}(m)$ and $m = p_{\mathcal{M}}(w)$. A pair (m, w) is said to *block* a matching $\mathcal{M}$, if

- m and w are not partners in $\mathcal{M}$, but
- m prefers w to $p_{\mathcal{M}}(m)$ and
- w prefers m to $p_{\mathcal{M}}(w)$.

The pair (m, w) is then called a *blocking pair*.

If a matching contains at least one blocking pair the matching is called *unstable*, otherwise it is called *stable*. The terms *matching* and *marriage* will be used interchangeably throughout the paper. The stable marriage problem is to the determine a stable matching for a given instance.

3 Two Sequential Algorithms

First we present two sequential algorithms for the stable marriage problem. The first is the previously mentioned Gale-Shapley algorithm published in the original paper on stable marriages. The second is an algorithm by *Tseng* and *Lee* (hereafter referred to as the Tseng-Lee algorithm) presented in [TL84]. This algorithm is interesting because it uses the divide-&-conquer-paradigm to construct the solution, a paradigm which exhibits some degree of parallelism.

3.1 The Gale-Shapley algorithm

The Gale-Shapley algorithm will always favor one sex over the other, depending on the parts the sexes plays in the algorithm. We will speak of a *man-oriented* or a *woman-oriented* algorithm.

We describe the man-oriented version here. The woman-oriented can be derived from the man-oriented by reversing the roles of the sexes.

Initially all persons are *free*. In each iteration a free man proposes to the first woman on his preference list which he has not previously proposed to. If the woman is free the two persons are matched. Is the woman engaged she chooses the man

with her highest preference. The rejected man becomes free and therefore has to try the next woman on his preference list. The algorithm continues by considering a free man in each iteration and terminates when all men become engaged. Note that while each engagement of a man, who is engaged more than once, is less desirable to him, the women get successively more favorable engagements. Proof of the correctness and termination of the Gale-Shapley algorithm is given in [GI89].

Every woman makes at most $(n-1)$ rejections, and when the last free woman gets a proposal the algorithm stops, that is the last free woman will not reject anyone at all. The number of rejections is therefore maximum $(n-1)*(n-1) = n^2-2n+1$. Every iteration involves at least one rejection, unless all women have an engagement. This gives a worst-case complexity of $n^2 - 2n + 1 + 1$ iterations. The structure of such a worst-case can be found in [GI89].

3.2 The Tseng-Lee algorithm

The algorithm of *Tseng* and *Lee* uses the well known *divide-&-conquer* paradigm. Having an instance $\mathcal{P}$ of the stable marriage problem, $\mathcal{P}$ is in the *dividing step* divided into two subproblems $\mathcal{P}_1$ and $\mathcal{P}_2$, where $\mathcal{P}_1$ contains the men from 1 to $\frac{n}{2}$ (assuming for the moment that n is even), and $\mathcal{P}_2$ contains the men from $\frac{n}{2}+1$ to n.

These subproblems are then solved recursively, resulting in two "stable submatchings". There are no conflicts between the men and the women in either of the two submatchings, and they can in a sense be regarded as *locally* stable if we "forget" about the unassigned women in each submatching. It is now the responsibility of the *conquer* or *merging step* to solve any conflicts between pairs of the two subsolutions, thereby establishing one single solution.

If there are no conflicts between the two submatchings then they will constitute a stable matching and we are finished. Otherwise the conflicts involved have to be solved. If two men m and m' are engaged to the same woman w, she chooses between them by maintaining the engagement with the one she prefers and rejecting the other. We then assign the rejected man to the next woman on his preference list. This may generate new conflicts which are solved in the same way. The order in which the conflicts are solved is arbitrary. When all conflicts are solved the matching is stable.

Note that there may be more than two men in a given conflict because other rejected men may have been assigned to the same woman. The conflicts are nevertheless resolved principally the same way: the woman chooses the winner and the rest proceeds to the next woman on their preference lists.

The "bottom-case" of this recursive algorithm is a problem with one man and n women. zHere the man is assigned to his number one female choice.

The termination and correctness proofs are much like the proofs for the Gale-Shapley algorithm and are therefore omitted. As in the Gale-Shapley algorithm a man never proposes to the same woman more than once. Hence, a conflict resolved

at an earlier stage in the Tseng-Lee algorithm will not occur later, thereby ensuring an $O(n^2)$ worst-case time complexity. The worst-case instance for the Tseng-Lee algorithm is the same as for the Gale-Shapley algorithm, as the Tseng-Lee "degenerates" into the Gale-Shapley algorithm if the conflicts arise only at the top level of the divide-and-conquer, and the conflicts here are always between two men.

3.3 Running times

In order to test the programs we constructed two problem-generators. One generates random problem instances and another generates worst-case instances. Table 1 shows the running times in seconds of the randomly generated and the worst-case instances (average of 5 different instances).

n	*Gale-Shapley*		*Tseng-Lee*	
	random	worst-case	random	worst-case
250	0.01	0.16	0.08	0.38
500	0.01	0.76	0.30	1.68
750	0.02	1.88	0.67	3.92
1000	0.03	3.58	0.93	7.25
1250	0.04	5.70	1.75	7.25
1500	0.05	8.27	2.65	16.47
1600	0.05	9.42	2.56	18.72
1700	0.07	10.66	4.02	26.44

Table 1: Running times in seconds of the sequential algorithms.

The table clearly shows that the sequential Gale-Shapley algorithm is the best of the two, although the advantage drops from a factor 60 to a factor 2 working on the worst-case instance.

4 The Parallel Approach

Throughout this section p denotes the number of processors.

The parallel algorithms are implemented on the MEIKO parallel computer at DIKU. It is a message-passing parallel computer with distributed memory. The system consists of 16 Intel i860 processors, each equiped with 16 Mb of memory, and 32 T800 communication transputers. It should be noted that the MEIKO has a high startup latency, i.e. communication of small amounts of data are expensive in time.

4.1 Parallelization of the Gale-Shapley algorithm

The parallel version of the Gale-Shapley algorithm is developed by dividing the execution in phases.

In the first phase (the proposing phase) *all* free men propose to the woman highest on their preference lists simultaneously. In the second phase (the rejection phase) the women (in parallel) evaluate their proposals keeping the best offer and rejecting the remaining men.

Now the *free* men proceed to the next woman on their preference list. The algorithm continues alternating between the two phases until no man is unmatched at the end of the rejection phase.

Implementing this two-phase algorithm on the MEIKO parallel computer, it is important to keep in mind that the MEIKO has a high start-up latency vs. integer operation ratio. We must therefore restrict the number of communications in our parallel algorithm. We have adopted the *master-slave* approach, where the *master*-processor is in control of the data-flow of the entire algorithm (handling the free men), and the remaining processors (denoted *slaves*) are responsible for rejecting/engaging the men for some specified women.

The master is "packing" the proposals and sending the packets to the appropriate slave processors thereby minimizing the number of communication. The women each keep the best man, and return the others to the master (again packed in an array). When no men are returned to the master the algorithm terminates. A drawback of this approach is that the master processor becomes a bottleneck of the system.

4.2 Parallelization of the Tseng-Lee algorithm

In the parallelization of the Tseng-Lee algorithm a problem of size k is divided into two "subproblems" each of size $\frac{k}{2}$. These two subproblems is solved independently by two processors. The solutions of the two subproblems are then send to the parent processor and merged together using the merging-step described in section 3.2.

As mentioned earlier it is possible that all computation is made in the root processor, as no conflicts has to be resolved in the subproblems. For such an instance the parallel algorithm "degenerates" into a sequential one, but as reported in [TL84] a theoretical analysis showns that the probability of experiencing of the worst case in randomly generated instances is extremely small.

The partitioning is stopped and the problem solved sequentially if the problem reaches some critical size. Unfortunately our experiments show that the best threshold is not to divide the problem at all, that is, to solve the problem sequentially!

If one considers the above implementation the utilization of processors does not seem to be optimal. Once a processor has sent it's data down to the children it is idle waiting for the results. A way to utilize the processors better is to let more *processes* (that is instances of the program) work on the same *processor*. When a process is sending down it's two subproblems the left child will be a process residing on the same processor while the right child will be a process on another processor.

This algorithm will only be implemented for *complete* binary trees, that is for 1, 2, 4, 8 and 16 processors. Note that having more processors on the same processor decreases the size of the problems that can be solved as more processes has to shared

the memory.

With the "smart" configuration some processors will have a larger workload than others. We therefore have to consider balancing the load of the processors in order to make them work on approx. the same amount of data. Here a simple way of balancing the load is implemented. Instead of dividing the problem in two equally sized subproblems, one problem is made larger than the other.

5 Experimental results

In testing the parallel algorithms we concentrate the testing solely on complete processor-trees. The instances used are the same as for the sequential algorithms.

An entry **oom** (out of memory) indicates that the given problem could not be solved due to insufficient amounts of memory available, while **ter** (terminated) indicates that all the generated instances for the problem had been running for at least 5 minutes before they were stopped. All running times are presented in seconds.

p	n							
	250	500	750	1000	1250	1500	1600	1700
Randomly generated instances								
3	0.25	0.70	1.28	1.12	1.83	2.83	1.71	oom
7	2.38	5.16	6.48	7.47	10.49	13.94	9.35	oom
15	2.27	5.54	9.44	8.81	9.94	16.50	14.32	oom
Worst-case instances								
3	44.83	182.14	ter	ter	ter	ter	ter	oom
7	45.73	184.55	ter	ter	ter	ter	ter	oom
15	49.60	200.33	ter	ter	ter	ter	ter	oom

Table 2: The results of the parallel Gale-Shapley algorithm

In Table 2 the execution times for the parallel Gale-Shapley algorithm are presented. Comparing the times for the parallel Gale-Shapley algorithm with the times for the "normal" parallel Tseng-Lee algorithm shown in Table 3, it is obvious that the parallel Gale-Shapley algorithm is extremely sensitive to the number of conflicts. The difference between the times for the randomly generated instances and the worst-case instances are several orders of magnitude higher for the parallel Gale-Shapley algorithm than for the "normal" parallel Tseng-Lee algorithm.

As can be seen in Table 2 we had to terminate the algorithm after 5 minutes for many instances. Because the parallel Gale-Shapley algorithm uses more memory than the parallel Tseng-Lee algorithms it was not possible to test the instances of size 1700.

For the "normal" version of the parallel Tseng-Lee algorithm we report the results in the Table 3. The results for the "smart" Tseng-Lee algorithm are shown in the

Table 4. Here some instances were too big due to the fact that more processes have to share the same memory.

p	n							
	250	500	750	1000	1250	1500	1600	1700
Randomly generated instances								
3	0.28	1.10	2.73	5.44	6.60	11.62	11.04	13.32
7	0.41	1.54	3.42	5.90	9.50	14.87	15.73	18.15
15	0.53	1.89	4.06	7.06	11.36	16.05	18.95	20.84
Worst-case instances								
3	0.74	2.91	7.19	13.29	16.01	29.36	32.02	36.39
7	0.85	3.46	8.17	14.08	18.89	31.83	36.17	40.63
15	0.95	3.68	8.33	14.92	24.10	34.06	39.11	44.06

Table 3: The results of the "normal" parallel Tseng-Lee algorithm

p	n							
	250	500	750	1000	1250	1500	1600	1700
Randomly generated instances								
2	0.63	2.73	4.90	11.59	16.99	25.97	27.46	oom
4	0.96	3.89	8.50	15.51	24.29	37.07	oom	oom
8	1.36	4.78	9.87	17.75	27.64	oom	oom	oom
16	1.37	4.77	10.73	18.82	29.64	oom	oom	oom
Worst-case instances								
2	1.07	4.48	10.44	19.62	28.57	43.77	48.25	oom
4	1.26	5.82	12.88	23.12	37.40	52.58	oom	oom
8	1.54	6.24	14.07	25.70	40.10	oom	oom	oom
16	1.80	6.61	14.94	26.88	42.19	oom	oom	oom

Table 4: The results of the "smart" parallel Tseng-Lee algorithm

Comparing the results for the two parallel Tseng-Lee algorithms we conclude that the "smart" parallel Tseng-Lee algorithm is not so smart after all. If we compare the times of the two parallel Tseng-Lee algorithms where the number of processors is equal, we observe that the "smart" version is at least a factor 3 slower, and often the factor is around 4.5. Interestingly, if we compare the times where the number of *processors* in the "normal" version is equal to the number of *processes* in the "smart" version, the factor is 2.5. The reason herefore must be that more processes can be active at the same time on one processor, thereby slowing down computation.

Consider the "normal" version with three processors. The root first sends half the data to the left child, and then the other half to the right child. Notice that

while the root is sending data to the right child, the left has started finding a stable submatching. This means that the left child has an initial advantage over the right child, an advantage it keeps throughout the computation if the subproblems are "equally hard". Therefore we might be able to get better times if the left child gets a bit more to work on than the right child. In order to test this conjecture and see how much more work the left child must have we have run the "normal" version with three processors where the left child gets $50, 55, \ldots, 95\%$ of the data. This is done for the randomly generated instances of size $n = 500, 750, 1000$.

For $p = 3$ the load-balancing for $n = 1000$ improves the time by 42%, while the configuration with 7 processors at best is 13% faster.

It is notable that for $p = 3$ the best times are all achieved at 85%, and for $p = 7$ the best values are all obtained at 65%. This indicates that the best percentage is *independent* of the size of the test data. Secondly the two configurations obtain their best times with different percentages, which might hint that there is a dependency between the percentage and the number of processors.

The configuration for $p = 7$ can be viewed as a root with two trees of size three as children. In the previous runs all processors where using the same percentage, but as setups with three processors are running optimally at 85% we may get better times by fixing the children of the root in the $p = 7$ setup at 85% and changing the percentage only for the root. The results of this experiment gave an improvement of the algorithm by 6% compared to the previous load-balancing scheme, where the percentage was equal on all levels of the processor-tree, and the algorithm was 19% better than the normal version without load-balancing. With $p = 15$ we fix the children of the root to the configuration optimised above and vary the percentage at the root processor. The best times were achieved at 65% (which made it 13 to 21% better than the "normal" version without load-balancing), and it seems to be independent of the size.

We have tried to improve the "smart" algorithm by balancing the load. It did improve the times of the "smart" version, but they remained worse than the "normal" version and will therefore not be commented further.

As this algorithm (the "normal" version with different load-balancing at each level) looks to be the best parallel version of the Tseng-Lee algorithm we have run the algorithm for the remaining sizes and also for the worst-case instances. The results are given in Table 5. We will call the algorithm for the "optimal" version of the parallel Tseng-Lee algorithm. Note, although, that the algorithm is still slower than both the sequential ones.

For the randomly generated instances the "optimal" version is from 8 to 40% better than the "normal" version. Most of the improvements are about 20%. The algorithm is, nevertheless, still worse than the sequential versions and becomes worse as more processors are added.

In the remaining part of the paper the parallel Tseng-Lee algorithms commented upon are the "normal" and the "optimal" versions.

For each pair (p, n) we have calculated speedup and efficiency. In Table 6 the best

p	n							
	250	500	750	1000	1250	1500	1600	1700
Randomly generated instances								
3	0.26	0.98	2.09	3.84	5.87	8.28	9.56	11.36
7	0.36	1.41	3.02	5.27	8.13	11.25	12.59	15.52
15	0.44	1.56	3.46	6.22	9.66	13.62	15.42	18.01
Worst-case instances								
3	0.68	2.82	6.45	11.61	18.39	26.72	30.53	34.57
7	0.79	3.19	7.24	12.98	20.53	29.80	34.00	38.44
15	0.86	3.45	7.82	13.99	22.09	32.03	36.55	41.78

Table 5: The results of the "optimal" parallel Tseng-Lee algorithm

values for speedup and efficiency are given. For fixed p we have taken the greatest of the $S_p(n)$ respectively $E_p(n)$ values where $n = 250, 500, \ldots, 1500, 1600, 1700$. As can be seen, the sequential algorithms and especially the Gale-Shapley algorithm are always faster than the parallel algorithms no matter how many processors these use.

Algorithm		Random data		Worst-case	
	p	S_p	E_p	S_p	E_p
Parallel	3	$2.94 \cdot 10^{-2}$	$9.80 \cdot 10^{-3}$	$4.19 \cdot 10^{-3}$	$2.62 \cdot 10^{-4}$
Gale-Shapley	7	$5.37 \cdot 10^{-3}$	$7.67 \cdot 10^{-4}$	$4.14 \cdot 10^{-3}$	$2.59 \cdot 10^{-4}$
	15	$3.65 \cdot 10^{-3}$	$2.43 \cdot 10^{-4}$	$3.81 \cdot 10^{-3}$	$2.38 \cdot 10^{-4}$
"normal"	3	$9.12 \cdot 10^{-3}$	$3.04 \cdot 10^{-3}$	0.356	$2.22 \cdot 10^{-2}$
Parallel	7	$6.49 \cdot 10^{-3}$	$9.28 \cdot 10^{-4}$	0.302	$1.88 \cdot 10^{-2}$
Tseng-Lee	15	$5.29 \cdot 10^{-3}$	$3.53 \cdot 10^{-4}$	0.243	$1.52 \cdot 10^{-2}$
"optimal"	3	$1.02 \cdot 10^{-2}$	$6.38 \cdot 10^{-4}$	0.310	$1.94 \cdot 10^{-2}$
Parallel	7	$7.10 \cdot 10^{-3}$	$4.44 \cdot 10^{-4}$	0.278	$1.73 \cdot 10^{-2}$
Tseng-Lee	15	$6.41 \cdot 10^{-3}$	$4.01 \cdot 10^{-4}$	0.258	$1.61 \cdot 10^{-2}$

Table 6: The speedup and efficiency for the parallel Gale-Shapley and Tseng-Lee algorithms

With such bad speedup and efficiency the conclusions are very clear. The parallel algorithms are not even bad but appalling. They are not better than any of the two sequential algorithms, and the more processors we use the worse speedup and efficiency values we get.

Examining in detail an iteration in the sequential Gale-Shapley the only thing done is checking the next woman on a free man's preference list and hence if the woman is free make an assignment. If she is not free we test the two men against each other and then in the worst case making four operations (making a new assignment,

deleting the old assignment, setting the former fiance *free* and setting the new fiance as *engaged*). In the practical implementation this corresponds in worst-case to making two integer comparisons and three integer assignments. These operations are all integer operations and they are quickly performed by the processor. Hence the amount of data we distribute in the parallel algorithms must contain work enough to outweight the cost of the communication. For our algorithms this does not seem possible. We use to much time on communication compared to the time used for computation.

	n							
	250	500	750	1000	1250	1500	1600	1700
#conflicts	1383	2493	4064	5250	6634	7609	9404	10002
#conflicts/n	5.53	4.99	5.42	5.25	5.31	5.07	5.88	5.88
longest chain	39	40	47	50	51	54	57	57

Table 7: Number of conflicts and the longest chain of refusals in an randomly generated instance

Finally we measured the number of conflicts that occurred in the randomly generated instances, and from this we calculated the ratio between the number of conflicts and the size of the instances. Additionally we measured the maximum number of refusals a man was involved in. The average of the five runs are shown in Table 7.

Note that the ratio between the number of conflicts and the problem size is almost constant. This suggests that the sequential algorithms in the average case has linear time-complexity in n, which since the problem size is n^2 means that the algorithms are $O(\sqrt{n})$ algorithms and thus sublinear.

Hence, the parallel algorithms should show sub-linear behaviour in order to "beat" the sequential ones. This difficult goal was not obtainable and leading to the bad results. In fact the parallel Gale-Shapley algorithm seems to behave as an linear time algorithm in the average case, while both the of parallel Tseng-Lee algorithms seem to be quadratic even in the average case.

The number of refusals in the longest chain is important in two ways. Firstly it shows how small portions of the men's preference lists we are using. Hence, a lot of the data sent down in the parallel Tseng-Lee algorithm is never used, but it is not possible to know that *a priori*. Secondly, for the parallel Gale-Shapley algorithm, the longest chain is the number of startups that at least one man has to go through, as he is refused every time. The slave-processors has to make the same amount of startups also. Alone this amount of startups means that the parallel Gale-Shapley looses a lot of time compared to the parallel Tseng-Lee algorithm (with 16 processors it has to make 45 startups regardless of the problem size).

6 Conclusion

Our experiments clearly show that the parallel algorithms are slower than the sequential ones, and they get even slower the more processors we use. As stated earlier, the communication is too expensive compared to the amount of work to be done. Clearly the stable marriage problem is to simple to be solved with commercial parallel computers.

A warning was issued in [Qui85] by Quinn who writes: "The average case speedup of parallel stable marriage based on the McVitie and Wilson algorithm [like ours] will be small,...". Even so, it must be noted that although the sequential algorithms have a quadratic worst-case time bound in n, they are in fact linear in the size of the input data, because the input consists of two $n \times n$-matrices, and in the average case they are linear in n and $\sqrt{n}$ algorithms in the size of the input.

References

[GI89] Dan Gusfield and Robert W. Irwing. *The Stable Marriage Problem – Structure and Analysis.* MIT Press, 1989.

[Lar94] Jesper Larsen. A parallel approach to the stable marriage problem, 1994. Graduate project work.

[Mei92] Meiko Limited. *CSTools Tutorial For Portland C/860 Programmers*, 1992.

[Qui85] Michael J. Quinn. A note on two parallel algorithms to solve the stable marriage problem. *BIT*, 25:473 – 476, 1985.

[TL84] S. S. Tseng and R. C. T. Lee. A parallel algorithm to solve the stable marriage problem. *BIT*, 24:308 – 316, 1984.