Annales UMCS Informatica AI 2 (2004) 81-89

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

# The implementation and analysis of parallel algorithm for finding perfect matching in the bipartite graphs

Maciej Chróśniak[a], Jakub Dworniczak[a], Karol Ziarko[a], Marcin Paprzycki[ab*]

[a]*Department of Mathematics and Computer Science, Adam Mickiewicz University, Umultowska, 61-614 Poznań, Poland*
[b]*Computer Science Department, Oklahoma State Uniwersity, Tulsa, OK 74106, USA*

**Abstract**

There exists a large number of theoretical results concerning parallel algorithms for the graph problems. One of them is an algorithm for the perfect matching problem, which is also the central part of the algorithm for finding a maximum flow in a net. We have attempted at implementing it on a parallel computer with 12 processors (instead of the theoretical $O(n^{3.5}m)$ processors). When pursuing this goal we have run into a number of practical problems. The aim of this paper is to discuss them as well as the experimental results of our implementation.

## 1. Introduction

Development of parallel algorithms for the graph problems is a peculiar area. On the one hand, there exists a large body of research (and literature) that presents theoretical algorithms developed for a number of equally theoretical models of parallel computers (see [1] and references listed there). On the other hand, there exist almost no results where parallel graph algorithms have been implemented on the existing parallel machines.

One of the sub-areas where such a situation is very clear is when the algorithms for finding perfect matching in graphs are considered. This problem has very well defined real-life applications. For instance, finding perfect matching in the bipartite graphs is a core of an algorithm for finding a maximum flow on the net [1,2]. Existing approaches to finding perfect matching in a graph are mainly based on the RNC algorithms. Namely, these are probabilistic algorithms computed in polilogarithmic time using a polynomial number of

* Corresponding author: *e-mail address*: marcin@cs.okstate.edu. The research at Adam Mickiewicz University was sponsored by a scholarship from the Fulbright Commission. The computer time grant from the Poznań Supercomputing and Networking Center is kindly acknowledged.

processors [1-4]. Karp, Upfal and Widgerson were the first to propose an RNC algorithm for solving this problem [3]. However, in our work we have decided to follow a more elegant (and claimed to be simpler and more efficient) algorithm of Mulmuley, Vazirani, and Vazirani [4], which can be summarized as follows (for all the remaining details as well as theoretical background see [1,4-7]):

Let $G$ be a graph with a set of vertices $V$ and edges $E$: $G = (V, E)$, $|V| = n$, $|E| = m$

1. For each edge $e_{ij} = (i,j) \in E$ select randomly a number $w_{ij} \in [0,...,2*m]$.
2. Form the Tutte matrix of $G$ (or Edmonds matrix for bipartite graphs), assign weight $2^{wij}$ for each $e_{ij} \in E$ (a result of a new matrix $A$ is created).
3. Compute in parallel the determinant $det(A)$ and the adjoint $D$ of $A$.
   - the adjoint matrix $D$ has the following form:

$$D = \left[ d_{ij} \right]_{nxn},$$

$$d_{ij} = (-1)^{i+j} \cdot \det\left( A_{ij} \right).$$

   - $A_{ij}$ is a matrix obtained from $A$ by deleting the $i$-th row and $j$-th column.
4. Let $2^w$ be the highest power of 2 that divides $det(A)$.
5. For each edge $e_{ij} \in E$ compute $(det(A_{ij})2^{wij})/ 2^w$.
6. If this value is odd then include $e_{ij}$ in the matching.

In [4] it is shown that this algorithm is computed in $O(log^2 n)$ steps using $O(n^{3.5}m)$ processors. This result is based on the parallel integer matrix inversion algorithm proposed by V. Pan in [8]. This result brings some interesting consequences when one considers implementing this algorithm. Let us consider a graph with $|V| = n = 80$ vertices and $|E| = m = 156$ edges. In this case the proposed algorithm can be completed in $(\log_2 80)^2 \approx 40$ steps when implemented on 714,396,886 processors. Obviously, these numbers are based on the *bigO* complexity functions and thus do not provide us with exact values. However, they are presented to show the practical absurdity of a perfectly reasonable theoretical result. Not only the most powerful existing computer has fewer than 10000 processors and the largest number of processors existing ever in a single machine was about 65000, but also one should ask how reasonable are thecomplexity functions involving 714 million of processors as far as, for instance, their connectivity and communication are concerned. Finally, observe how small a graph how large a computer are required and try to extrapolate the required computational power for realistic sizes of the networks for which flow problems are considered in practice.

## 2. Proposed implementation

While the theoretical estimates presented in [4] are highly unrealistic, we have decided to proceed with an attempt at an implementation of the proposed algorithm on an existing parallel machine. Our goal here was to establish its

realistic performance characteristics. To achieve this goal we have adjusted the original algorithm. First, in step (2) it is necessary to compute $det(A)$ and $n^2$ determinants of $det(A_{ij})$, $i, j = 1...n$. To achieve this goal we have used the matrix inversion; namely: $D = det(A)*(A^{-1})^T$ and the Gaussian elimination (complexity $O(n^3)$ [9]). Proceeding along this path we can compute $A^{-1}$ and $det(A)$ in a simple way (after reducing the matrix to an upper triangular form). However, due to the standard numerical "deficiencies" of operations on real numbers, the Gaussian elimination calculates only approximate values of the solution. At the same time, for the proposed algorithm to work, we need the exact values to know which edges belong to the matching (step 6 of the algorithm). That is also the reason why we could not use well-known libraries for linear algebra calculations (i.e. BLAS, LAPACK) that are efficient in matrix inversion – they use floating point numbers. To solve this problem we have decided to implement the Gaussian elimination based on the rational numbers and for this purpose to utilize the GMP (GNU Multiple Precision, [10]) library.

### 2.1 Details of parallelization

Our approach to parallelization follows the standard approach to parallelization of matrix computations described in [9]. However, since our approach involves rational numbers we cannot apply well-known blocking techniques that became a staple of high-performance matrix algorithms [9]. Instead we proceed with a simple master-slave model, where the master is active and takes part in the work of the whole group. In the main part of the algorithm, where the differences between the execution time of individual jobs can be the largest, we have used dynamic load balancing. The master tries to ensure availability of tasks for the slaves. It "puts aside" next job before beginning his part of computation. In this way, employees have next job in reserve and when they finish current one, they can take next even though the manager is busy.

More precisely, in the algorithm we can distinguish two parts of computing the inverse matrix (finding solution to the system of equations $A*X = I$ where $A$, $X$, $I \in R^{n \times n}$, and $I$ is the unit matrix). In the first part we apply Gaussian elimination to reduce matrix representing a given graph to the upper triangular form. Here, we perform independent simultaneous operations on rows distributed by the manager. In the second part, we back solve in parallel $n$ the systems representing the $n$ columns of the identity matrix obtaining the inverse of $A$.

## 2. Experimental setup

We have implemented the proposed algorithm in C. In order to make the algorithm work in parallel we used the POSIX threads. This solution was "imposed" by utilization of rational numbers. With the POSIX threads we avoid

moving around very large numbers (results of Gaussian elimination performed on rational numbers, see below). On the other hand, this solution restricted our implementation to parallel computers with shared memory (or virtual-shared memory). Furthermore, we had to organize access to the shared data which is somewhat more complicated by implementation of dynamic distribution of jobs. This made us ensure appropriate synchronization of calculation units (master and slaves) that was realized by using critical sections and special structures such as flags of access and progress.

We have experimented with our code on a 12-processor SGI Power Challenge XL. This computer has shared memory and MIPS R8000 processors and runs IRIX version 6.2 operating system. Our code was compiled using MIPSPro C compiler with the optimization level – O2. Because of usage of threads we had to utilize clock based on daytime (we could not locate a special clock for threads). To reduce the effect of machine workload we have run multiple experiments (minimum of three) and in each case we report the best obtained time.

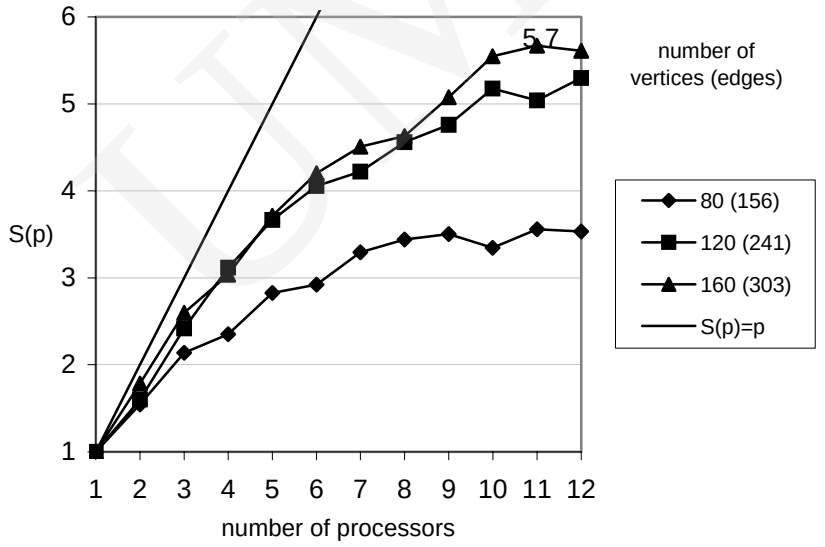


Fig. 1. Speedup of the solution process for p = 1, 2, …, 12 processors

Table 1. Times (in minutes) required for finding the perfect matching for the increasing number of processors

| $\|V\|(\|E\|)\backslash p$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 80 (156) | 0.88 | 0.57 | 0.41 | 0.37 | 0.31 | 0.30 | 0.27 | 0.26 | 0.25 | 0.26 | 0.25 | 0.25 |
| 120 (241) | 10.12 | 6.32 | 4.18 | 3.25 | 2.76 | 2.49 | 2.40 | 2.22 | 2.13 | 1.96 | 2.01 | 1.91 |
| 160 (303) | 28.19 | 15.79 | 10.85 | 9.28 | 7.59 | 6.70 | 6.25 | 6.09 | 5.55 | 5.08 | 4.97 | 5.02 |

## 3. Experimental results

The first series of experiments was devoted to finding perfect matching in the bipartite graphs. Due to the relatively long time of computations (the SGI Power Challenge is an almost 10 year old technology) we have experimented with relatively sparse graphs (the first of them is exactly the graph mentioned in the introduction to illustrate the purely theoretical value of some well-known algorithms). In Table 1 and Figure 1 we present the time and speedup obtained for three graphs and for $p = 1, 2, \ldots, 12$ processors. Speedup is calculated using a standard formula $S(p) = T_1/T_p$, where $T_1$ – time on one processor and $T_p$ – time on $p$ processors; which is reasonable since we utilize all processors, including the master.

The obtained results are satisfactory. On 11 processors we have obtained a speedup of 5.7 and thus efficiency above 50%. We also observe that as the size of the graph increases, the overall parallel performance of the code improves. Obviously, as the time of computation increases, synchronization has less impact on the procedure in comparison with the time of independent calculation performed independently by processors.

Note that the proposed algorithm is very sensitive to the density of the graph. We have experimented with the increasing number of edges for a fixed number of (80) vertices and found that the total time increases from less than a minute for 83 edges to almost 30 minutes for 202 edges. This is directly related to the fact that for the increasing number of vertices, (the magnitude of weights assigned to edges is from the range $[2^0,..., 2^{2*m}]$, where $m = |E|$ (see below).

Separately, we have experimented with general, non-bipartite graphs (as the proposed approach can find the perfect matching in any graph). Figure 2 and Table 2 represent the time of computation and speedup for 80 vertices and 155 and 156 edge general and bipartite graphs and for $p = 1, 2, \ldots, 12$ processors.

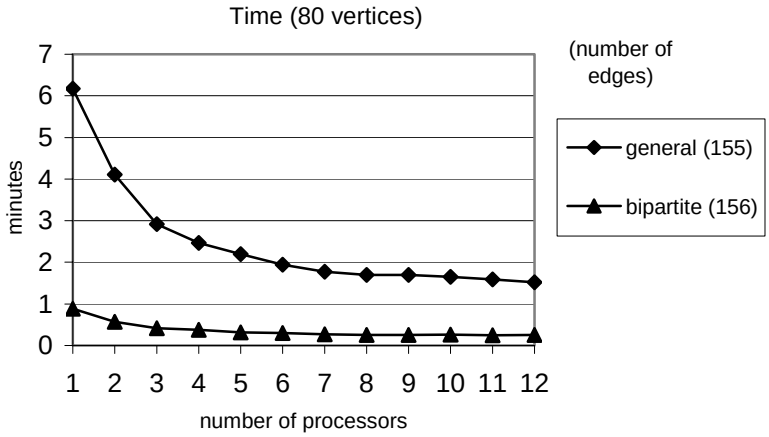


Fig. 2. Computation time (in minutes) for p = 1, 2, …, 12 processors

Table 2. Speedup of finding perfect matching for general and bipartite graphs

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| general (155) | 1.00 | 1.50 | 2.12 | 2.51 | 2.81 | 3.17 | 3.48 | 3.63 | 3.64 | 3.74 | 3.88 | 4.05 |
| bipartite (156) | 1.00 | 1.55 | 2.14 | 2.35 | 2.83 | 2.92 | 3.29 | 3.44 | 3.50 | 3.34 | 3.56 | 3.53 |

The results are similar to those obtained for the bipartite graphs. The only difference is that the time is substantially longer. By the same token, the obtained speedup is somewhat better for the general graphs.

### 3.1. Parallel versus sequential algorithm

While looking at the time required for solving the problem for relatively small and sparse graphs we came to conclusion that they are rather large. We have also observed the strong dependency of the algorithm on the length of the random number (number of edges) as and thus decided to compare the performance of our parallel code with that of a sequential method. We have selected a well-known Hungarian method [11]. This method has theoretical complexity $O(n^3)$ (similar to that of Gaussian elimination and computation of the adjoint – the core of the parallel algorithm). In Table 3 we present the time to find the perfect matching for the same three graphs as in Table 1. We report the time of the parallel algorithm on 1 processor, the best time on 2-12 processors and the time of the sequential algorithm (obtained on the same machine).

Table 3. Time for computing perfect matching of the parallel and sequential algorithms. Times of parallel algorithm are reported in minutes, while those of the sequential algorithm are reported in seconds

| \|V\| | parallel (1 proc.) | parallel (min) | sequential |
|---|---|---|---|
| 80 | 52.80 | 15.00 | 0.05 |
| 120 | 607.20 | 117.60 | 0.26 |
| 160 | 1691.2 | 303.60 | 0.91 |

The results are devastating as they show that, in spite of significant shortening of time by distribution of work between processors, it is hard to talk about competitiveness of parallel algorithm. We have therefore looked for the reason for such an enormous difference in performance. Initially, we have directed our attention to the fact that the processing time seems to be related to the number of edges and thus the size of the random numbers filling the adjacency matrix. The algorithm we have implemented, utilizes random numbers of size up to $2^{2|E|}$. Hence, to understand the effect of this fact, we decided to measure memory utilization of our program. The results for the three graphs that we have

experimented with are depicted in Figure 2 (these are approximate values gained from a tool listing existing processes in the operating system).

The chart shows memory usage at the beginning of computation, when large numbers were drawn and stored (min); and at the moment when the program took up the most amount of memory (max), usually near the end of algorithms execution. Let us observe that even the initial memory utilization is rather large, but it explodes as the program progresses. This increase of total memory usage between the initialization of computation and its completion is related to fast increasing sizes of rational numbers during calculation of the adjoint matrix *D*. This is an effect of utilizing rational numbers that increase in size as the Gaussian elimination is carried out. To observe this effect more closely, we have decided to study further the effect of application of rational numbers on efficiency of the proposed algorithm.

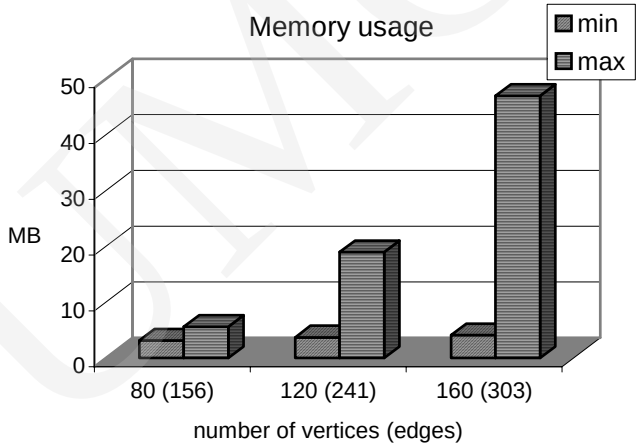


Fig. 3. Memory utilization of the parallel algorithm; the results in Mbytes

### 3.2. Additional analysis of operation on rational numbers

In the first series of experiments we have established the times of basic operations on rational numbers of size 60 Kb (typical size for our program) and compared them with these of floating-point numbers (long double – 8B). We have performed 10 thousand multiplications and subtractions. The total time for the rational numbers was 183.4 seconds and for the floating-point numbers 0.014 seconds. We have thus decided to check efficiency of operations on rational numbers while extending their size. These results determine the time (in sec.) of 10 thousand operations.

We used $n$ Kb to denote random numbers from the range of $[1..2^{1000n}]$. Based on the results presented in Figure 4 we can conclude that the time required to complete operations depend linearly on the size of rational numbers. Thus, while the size itself is not a problem in the early stages of the algorithm, it becomes so

as the size of the numbers increases, while the total number of arithmetical operations behaves like $O(n^3)$ magnifying the effect of increasing size of numbers.

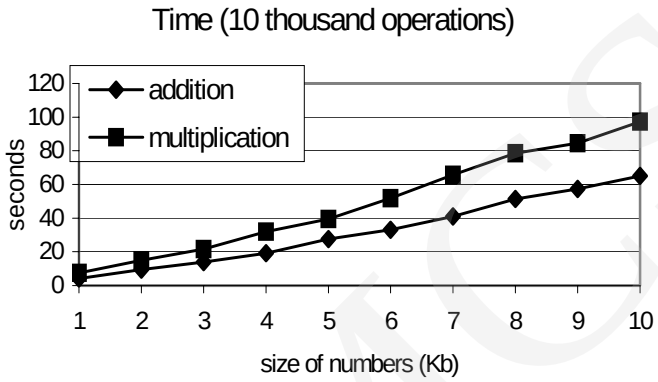


Fig. 4. Time of 10000 operations for the increasing size of rational numbers

### 3.3. The attempt at applying algorithm to the solution of the maximal flow problem

As we mentioned in the introduction, algorithm for finding perfect matching can be an intermediate step to find a maximum flow in a net. Since this is a randomized algorithm, it requires multiple steps. More precisely, in order to handle this problem we have to perform about $log_2|V|$ intermediate steps of finding perfect matching. Besides, each procedure finds matching with probability greater than 1/2 (if perfect matching does not exist algorithm always gives a correct answer). Thus, theoretically, we have a guarantee that the procedure of searching for maximum flow works correctly with probability greater than $1/|V|$.

However, during our experiments we noticed that the algorithm gives correct answers more often than it could be concluded form the theoretical estimation ($P>1/2$ for $C=2$). We have made 100 tests and gained the following results:

Table 4. Actual results of finding perfect matching for varying value of C; based on 100 test

| C | 2 | ¼ | 1/16 |
|---|---|---|---|
| P | 0.98 | 0.81 | 0.51 |

There is some optimistic accent here, because we have obtained the actual probability $P \approx 1/2$ for much smaller $C$ ($\approx 1/16$) than expected from the theoretical analysis. But it does not mean, of course, that we could actually use the algorithm with this value of $C$. There exists a technique of improving the probability of correct answer. If we want to do this we have to repeat the

algorithm for perfect matching several times or use larger numbers as the weights for edges at the beginning of the procedure. Both methods lead to an extension of the time of work.

Overall, these results are unsatisfactory taking into account the overall time that the proposed algorithm takes. That is why we have decided not to pursue the implementation of finding the maximum flow algorithm.

## 4. Summary

We have seen a large number of results concerning development of fast parallel algorithm for graph problems, in particular, for perfect matching. What is more, these algorithms, being probabilistic in nature, thanks to resignation from certainty of getting correct result, are usually more time-efficient. Unfortunately, to implement one of such algorithms, we had to modify the original algorithm and apply a method from linear algebra and utilize rational numbers. As it turned out, despite significant parallelization, our algorithm is not able to compete with a sequential one.

Graph algorithms are a domain in which it is not easy to find parallel solutions. Existence of good optimized sequential procedures makes it even more difficult to create competitive algorithms. Many attempts giving theoretically optimistic results do not take into account practical realization. They do not bring up such problems as ability of communication between processing units and performance that can be achieved in the required architecture.

## References

[1] Karp R.M., Upfal E., Widgerson A., *Constructing a perfect matching in Random NC*, Combinatorica, 6 (1986) 35.
[2] Mulmuley K., Vazirani U., Vazirani V., *Matching is as easy as matrix inversion*, In 19th ACM Symposium on Theory on Computing, 355-365, ACM Press, (1987).
[3] Karpiński M., Rytter W., *Fast Parallel Algorithms for Graph Matching Problems*, Oxford University Press, (1998).
[4] Diaz J., Serna M.J., Spirakis P., Toran J., *Paradigms for fast parallel approximability*, Cambridge University Press, (1997).
[5] Preparata F.P., Sarvate D.V., *An improved parallel processor bound in fast matrix inversion*, Inf. Process. Lett., 7 (1978) 148.
[6] Mahajan M., Vinay V., A *combinatorial algorithm for the determinant,* SODA97, (1997) 730.
[7] Galil Z., Pan V., *Parallel evaluation of the determinant and inverse of a matrix*, Inf. Process. Lett., 30 (1987) 41.
[8] Pan V., Fast *and Efficient Algorithms for the Exact Inversion of Integer Matrices*, Fifth Annual Foundations of Software Technology and Theoretical Computer Science Conference, (1985) 504.
[9] Golub G.H., Van Loan C.F., *Matrix Computations,* The Johns Hopkins University Press, (1997).
[10] www.swox.com/gmp – GNU Multiple Precision Arithmetic Library (GMP)
[11] Kuhn H.W., *The Hungarian method for the assignment problem,* Naval Research Logistics Quarterly, (1955).