Annales UMCS Informatica AI 2 (2004) 91-99

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

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

Parallel implementation of the k-connectivity test algorithm

Przemysław Sokołowski[a], Paweł Konieczka[a], Jakub Sochacki[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 University, Tulsa, OK 74106, USA*

Abstract

There exists a large number of theoretical results concerning fast parallel algorithms for graph problems, however, scarcely one finds reports of their practical implementation. In an attempt at partial filling this gap we discuss implementation of an algorithm performing the pretest for k-connectivity. This test is based, first, on the Scan-First Search algorithm introduced in [1]. Utilizing this procedure we decrease the size of the input graph by removing selected edges so that the resulting graph (certificate of k-connectivity) has only $O(kn)$ left. During this part of computations we can answer the question about k-connectivity negatively if a certificate cannot be generated. Afterwards, we can apply the test described in [2] to establish k-connectivity in the remaining cases.

1. Introduction

Let us start with defining the basic terms. An undirected graph G is defined as an ordered pair $G=(V, E)$, where V is the set of n vertices ($|V|=n$) and E is the set of edges ($|E|=m$). Furthermore, for an edge $e \in E$, $e = \{a, b\}$ is an unordered pair, where a, b are the vertices of G. We say that H is a spanning subgraph of G if $V(H) = V(G)$ and $E(H)$ is the subset of $E(G)$. T is a spanning tree of G if T is a connected, acyclic subgraph of G (graph is connected *iff* any two vertices of G can be linked by a path). The certificate of k-connectivity C is a spanning subgraph of G such that C is k-connected *iff* G is k-connected.

According to Menger's theorem [3] graph is k-connected *iff* there exist k vertex-disjoint paths between any two vertices in V. This theorem gives an easy algorithm for checking k-connectivity. Unfortunately, this method is inefficient

*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.

for any larger number of vertices and/or edges. As a part of the solution one can try to reduce the size of the graph by removing some edges (minimizing the degree of vertices) in such a way as to keep the k-connectivity unchanged. We will achieve this goal by trying to find the certificate of k-connectivity of G. If we are not able to generate certificate of k-connectivity we will establish that G is not k-connected (and our main problem is solved). However, it is still possible to find the certificate for a graph that is not k-connected so we need to check further the k-connectivity of the found certificate.

Finally, let us note that while there exist many ways of representing graphs, we will utilize a matrix-based one. Here, a graph G is represented by a binary matrix $M=\{a_{ij}\}$ of size $n*n$ (an incidence matrix of G), where $a_{ij} = 1$ *iff* there exists an edge $\{i,j\}$ in E, otherwise $a_{ij} = 0$. Because G is undirected, matrix M is symmetric so the complete information about the structure of the graph is stored in the upper (lower) triangle of M.

In our literature search we have not found any parallel implementation of any test for k-connectivity. There exists a number of theoretical results concerning parallel methods for establishing k-connectivity but they are based on rather unrealistic assumptions (for exapmle a polynomial number of processors).

We proceed as follows. In the next section we introduce the Scan-First Search algorithm that is used to establish the certificate of k-connectivity. We follow with the description and analysis of experimental results obtained on the 12 processor SGI Power Challenge parallel computer as well as on homogenous clusters of 17 PC's.

2. Scan-First Search algorithm

The crucial part of the algorithm is able to generate the certificate of k-connectivity for a given graph G. We achieve this goal by iterating the Scan-First Search (SFS) method (see [1] for its detailed description). In i-th round a spanning forest F_i of G_i is generated. Then every tree in F_i is numbered using the prenumber algorithm. Thanks to this numbering it is possible to generate a spanning forest F_i (for every vertex v we choose its neighbour u with the smallest number given by the prenumber algorithm). In the next iteration graph $G_{i+1} = (V, E(G_i) - E(F_i))$ is obtained form G_i by removing edges of forest F_i. It can be shown that after k steps graph $C = (V, \bigcup_i E(F_i))$ is the certificate of k-connectivity of G (where, $G_1 := G$).

2.1. Parallelization of the SFS algorithm

Let us now describe the details of our parallel implementation of the SFS algorithm:

1. To generate the spanning forest we use the Kruskal algorithm (description can be found in [3,4]; in [4] a distributed version of this algorithm somewhat similar to our approach is presented – we have not found any information about actual implementation of any of these algorithms). The idea of this algorithm is to build simultaneously trees from each vertex of G. If it is possible, trees are connected into larger trees so as a result we obtain a spanning forest (if graph G is connected, as a result we obtain a spanning tree). In order to parallelize the process, every processor is assigned some fraction of vertices of G (a given processor has knowledge only about the edges between the assigned vertices) and it is building "its" forest on those vertices. After each processor computes its forest, it exchanges results of computations with another processor and then both can generate larger forest using edges between their vertices. Because the algorithm is deterministic, after exchange both processors generate the same result.
2. Preorder numbering is a standard procedure which is described, for instance, in [3]. The main problem is its parallelization. We used a simple load-balancing approach: all trees in the forest are sorted in a non-decreasing order with regard to the number of vertices and the first (largest) tree is assigned to the first free processor, the second free processor receives the second tree and so on.

2.2. Implementation details of Kruskal's algorithm

Let us now provide further details of our parallel implementation of the Kruskal algorithm:

1. Our approach is based on the master-slave model of parallel computing (with the master only managing computations and the slaves performing all "useful" calculations). Initially, the master distributes parts of the coincidence matrix (in this paper we consider graphs with $n = h*2^m$ vertices) between p slave processors (p is a power of 2). Every slave receives n/p vertices – n/p rows of the coincidence matrix – so that every edge starting in a given vertex is known while other edges (between assigned vertices and others) cannot be used to grow the tree at first. Next all necessary computations are performed independently by slaves.
2. Since the forest is built in an iterative fashion (see above), after each iteration appropriate parts of the coincidence matrix are exchanged between processors. Since at the end of the first run of Kruskal every processor has his own copy of the matrix, in subsequent steps of checking k-connectivity there is no need to send the complete matrix – master sends only information about the removed edges so that processors can update their matrices of interest. In i-th iteration processors exchange computed

forests and parts of matrix (in every iteration the whole matrix is filled with missing rows). This exchange process is depicted in Figure 1.

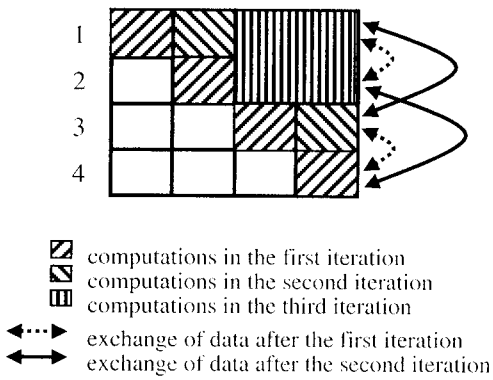

Fig. 1. Data exchange pattern for parallel Kruskal

The formula presented below as a C pseudocode is used to find processor to contact and exchnage data with:

```
if ( ( 0 <= (proc_nr - 1) % (count * 2) ) && ( (proc_nr - 1) % (count * 2) < count) )
        {with_who = proc_nr + count;}
else
        {with_who = proc_nr - count;}
```

where: count – iteration number, proc_nr – processor number, with_who – number of processor to contact.

3. Experimental results

The above described approach to establishing k-connectivity was implemented in C with the MPI library used for parallelization. Tests were performed on two parallel machines. First, a shared memory SGI Power Challenge XL with 12 MIPS R8000 CPU's, running at 90MHz and 1Gbyte of RAM. Here, regardless of the fact that this is a shared memory computer, we still proceeded with the MPI-based parallelization and the native SGI-provided MPI implementation was used. Since our approach required utilization of power of 2 slave processors we have used up to 9 processors of this machine (1 master and 8 slaves). Our experiments were executed while other programs were running. As a result we have observed a substantial variation in execution times. In each case the best result out of multiple runs is reported, however, it cannot be guaranteed that these are trully the best possible results. Second, a homogeneus cluster of 17 PCs. Each of them had an Intel Pentium 4 processor running at 1.5 GHz and 256 Mbytes of RAM. PCs were connected by a Catalyst 6500 switch

with full duplex 100Mbit/s switching capability. Open source versions of LAM and MPI have been used. Here the experiments were performed at night on an "empty" system, however, we have still observed a relatively large variation in execution times. As above, the best results out of multiple runs are reported. Finally, since we have used a master-slave approach in the results reported here the "single-processor" mode consists really of two processors; one master and one slave. Therefore, we have decided to compute speedup as a ratio between time on ***two*** processors and time on ***p+1*** processors.

On the SGI we tested our program using complete graphs on 1024, 2048 and 3072 vertices (for bigger graphs lack of available memory caused page swapping which made our results practically useless). The results of our experiments are reported in Figures 2-4. On the cluster we used graphs of size 2048, through 6144 vertices and reports are presented in Figures 6-7.

We chose complete graphs because for the problem of establishing k–connectivity they constitute the worst case scenario. This is due to the fact that the preorder numbering takes a very long time (as shown in Figure 5). Furthermore, since the graph is connected, there is practically no parallelization available in this stage of the algorithm. This being the case, this constitutes a classical Amdahl-type bottleneck for the whole problem. We can therefore observe the overall worst possible scenario for the complete SFS algorithm. The parameter k was tested for two values: 20 and 30. Obviously, we made an assumption that for bigger k speedup would be greater. The choice was caused by the hardware characteristics of the SGI computer. 10-connectivity required "almost no time at all to complete" while 40-connectivity took "forever to complete."

First, in Figure 2 we present the speedup obtained for the Kruskal algorithm on the SGI computer for the graphs with 1024, 2048, 3072 vertices and for 1, 2, 4, and 8 slave processors. We follow with Figures 3 and 4 that represent that speedup of the complete SFS algorithm for $k = 20$ and 30 respectively.

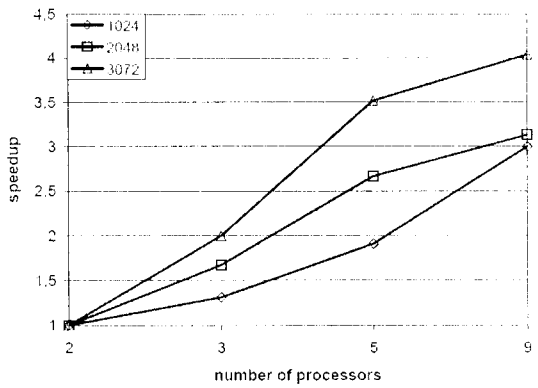

Fig. 2. SGI Power Challenge; speedup of Kruskal algorithm for the graphs of sizes 1024, 2048, 3072

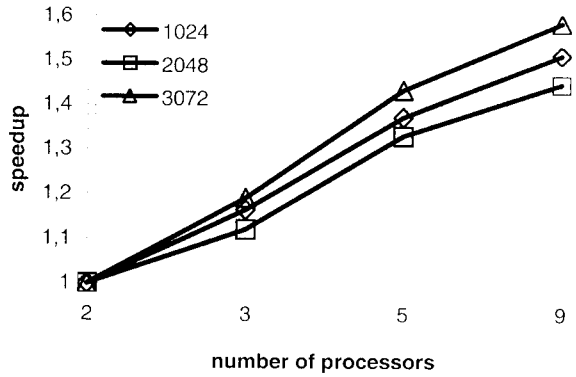

Fig. 3. SGI Power Challenge; speedup in the whole program for generating certificate of 20-connectivity for the graphs of sizes 1024, 2048, 3072

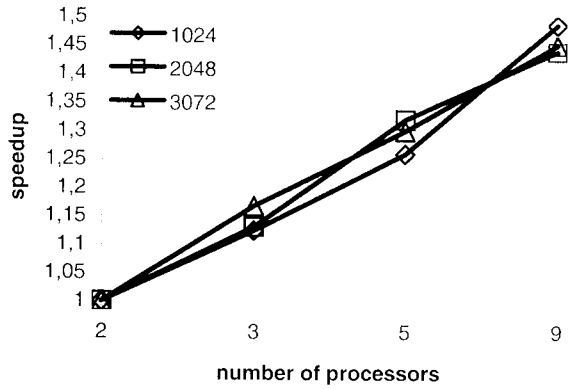

Fig. 4. SGI Power Challenge; speedup in the whole program for generating the certificate of 30-connectivity for the graphs of sizes 1024, 2048, 3072

Three observations can be made. First, speedup of Kruskal algorithm is relatively good. The efficiency reaches almost 50% for 8 slave processors. Second, speedup for $k = 20$ and 30 is very similar but slightly worse for $k = 30$. Third, the speedup of the whole process is substantially worse than that of Kruskal algorithm. The latter two facts can be explained by the performance of the second phase of the algorithm, which is illustrated in Figure 5.

Since we are experimenting with the worst case scenario of a complete graph the second phase of the algorithm is performed almost sequentially and is repeated as many times as the value of k (prenumbering is repeated in every iteration of the algorithm). This explains the superior performance of the ***20***-connectivity case, where the Amdahl-bottleneck repeats only 20, instead of 30 times.

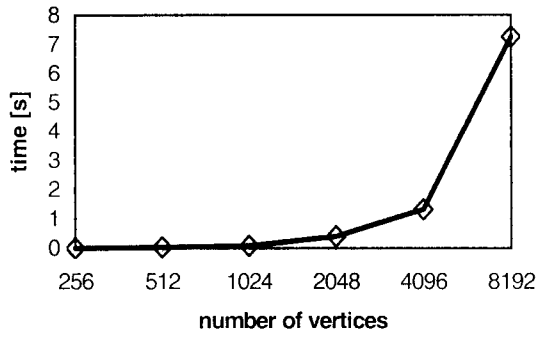

Fig. 5. SGI Power Challenge; time of numbering trees specified size with prenumber algorithm by one processor

The results of experiments on the described above homogeneous cluster of 17 PC's are depicted in Figures 6 and 7. First, the Kruskal algorithm and then the whole SFS algorithm for ***30***-connectivity.

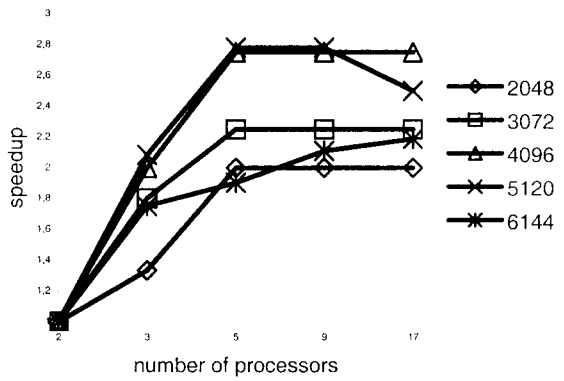

Fig. 6. PC Cluster; speedup in Kruskal implementation for the graphs of sizes 2048, ..., 6144

The results are very interesting and conforming to some of the earlier observations. The speedup of Kruskal's algorithm is much smaller for the cluster than for the SGI parallel computer. This fact can be attributed to its information exchange phases (Section 2.2, point 2). While we have been using MPI based parallelization, the SGI is a shared memory computer and SGI provided native MPI can manage information exchanges as prescribed in our code very efficiently. At the same time the cluster, while connected by a relatively fast switch, cannot move data around fast enough to obtain satisfactory speedup. In this case even further increase in the graph size does not result in substantial

gains in speedup as such gains are counterbalanced by the amount of data exchanged between processors.

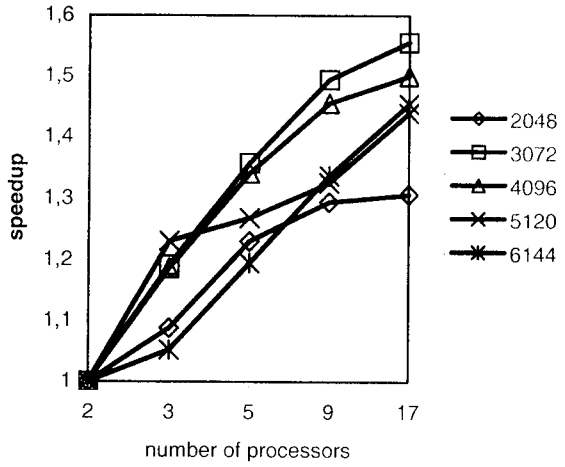

Fig. 7. PC Cluster; speedup in the whole program for generating certificate of 30-connectivity for the graphs of sizes 2048, ..., 6144

Overall the speedup of the SFS algorithm is similar on both computers, however an interesting trend can be observed. As the size of the graph increases, the overall speedup decreases. This is caused by the lack of parallelism in the second phase of the SFS algorithm. This can be related to the results presented in Figure 5. As the size of the graph increases, the time used by the prenumbering phase increases exponentially. This is combined with the iterative nature of the process thus further reducing any speedup gained in the Kruskal-phase of the SFS algorithm.

7. Concluding remarks

In this paper we have presented the results of our attempt at implementing parallel algorithm for establishing k-connectivity of the graph. We have implemented and tested the Scan-First Search algorithm. We have found that it is possible to implement the Kruskal's algorithm quite efficiently. This is especially the case for fast connection between processors (for instance, a shared memory of an SGI Power Challenge). At the same time, on a cluster connected over 100 Mbit/s network it was almost impossible to obtain satisfactory performance. The situation is much worse as far as the second phase of the SFS algorithm is concerned. We have used a rather simplistic approach to its parallelization and combined it with the worst case input data (a fully connected graph). This combination turned to be lethal to the performance of the code and suggests one of the places where a different approach needs to be found. Overall,

however, the very fact that we were able to implement sucesfully parallel graph algorithm and obtain speedup in the worst case input data should be treated as a success.

There exists a number of ways that the research reported here can be extended. First, from the results presented above one can conclude that the distributed model of computations is not the best for our problem (and possibly for most of the graph algorithms). Therefore it may be possible to improve performance of parallel Kruskal algorithm through a shared memory oriented implementation (for example using the OpenMP technology). The second research direction is an improvement in parallelization of the preorder algorithm, but for the time being we do not know how to do it. Finally, having this pretest for k-connectivity implemented we are now able to improve any k-connectivity test. Therefore, the next obvious step is to implement the algebraic algorithm for testing k-connectivity as described in [2].

Acknowledgments

We wish to thank Jerzy Jaworski, Jerzy Szymański and our families for help, patience and indulgence.

References

[1] Cheriyan J., Kao M.Y., *Thurimella R., Scan-First Search and sparse certificates: an improved parallel algorithm for k-vertex connectivity*, SIAM J. Comput., (1993).

[2] Linial N., Lovász L., Widgerson A., *Rubber bounds, convex embeddings and graph connectivity*, Combinatorica, (1988).

[3] Ross K.A., Wright C.R.B., *Matematyka dyskretna*, PWN, Warszawa, (1999), in Polish.

[4] Peleg D., *Distributed Computing: A Locality-Sensitive Approach*, SIAM, Philadelphia, (2000).