J Braz Comput Soc (2013) 19:161–166
DOI 10.1007/s13173-012-0089-z

ORIGINAL PAPER

Parallel transitive closure algorithm

C. E. R. Alves · E. N. Cáceres · A. A. de Castro Jr. · S. W. Song · J. L. Szwarcfiter

Received: 4 December 2011 / Accepted: 24 August 2012 / Published online: 25 September 2012

Abstract Using the BSP/CGM model, with p processors, where $p \ll n$, we present a parallel algorithm to compute the transitive closure of a digraph D with n vertices and m edges. Our algorithm uses $\log p + 1$ communication rounds if the input is an acyclic directed graph labeled using a linear extension. For general digraphs, the algorithm requires p communication rounds in the worst case. In both cases, $O(M/p)$ local computation is performed per round, where M is the amount of computation needed to compute a sequential transitive closure of a digraph. The presented algorithm can be implemented using any sequential algorithm that computes the transitive closure of a digraph D. We have implemented the algorithm using the Warshall transitive closure algorithm on two Beowulf clusters using MPI. The implementation results show its efficiency and scalability. It also compares favorably with other parallel implementations. The worst case (communication rounds) for a digraph was derived through an artificially elaborated example. However, in all our test cases, the algorithm never requires more than $\log p + 1$ communication rounds and presents very promising implementation results. The algorithm also can be applied to any $n \times n$ matrix that represents a binary relation.

Keywords Transitive closure · Parallel algorithms · Scalable algorithms · BSP/CGM algorithms

This work was partially supported by CNPq Proc. Nos. 30.1652/09-0, and CNPq, FUNDECT-MS and CAPES.

C. E. R. Alves
Universidade São Judas Tadeu, São Paulo, SP, Brazil
e-mail: c.e.r.alves@gmail.com

E. N. Cáceres (✉)
Universidade Federal de Mato Grosso do Sul,
Campo Grande, MS, Brazil
e-mail: edson@facom.ufms.br

A. A. de Castro Jr.
Universidade Federal de Mato Grosso do Sul,
Ponta Porã, MS, Brazil
e-mail: amaury.ufms@gmail.com

S. W. Song
Universidade de São Paulo, São Paulo, SP, Brazil

S. W. Song
Universidade Federal do ABC, Santo André, SP, Brazil
e-mail: song@ime.usp.br

J. L. Szwarcfiter
Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
e-mail: jayme@nce.ufrj.br

1 Introduction

There are many problens where we need to answer reachability questions. That is, can one get from city A to city D in one or more roads? The computation of the transitive closure of a directed graph can solve the above problem and it is fundamental to the solution of several other problems, such as shortest path and ancestor relationship problems. Several variants of the transitive closure problem exist [8].

The problem of computing the *transitive closure* of a directed graph (*digraph*) was first considered in 1959 by Roy [18]. Since then, a variety of sequential algorithms to solve this problem have been proposed. The basic sequential solutions usually employ the adjacency matrix of the digraph, considered as a Boolean matrix. Using this approach, Warshall [23] presented an $O(n^3)$ algorithm.

The best sequential algorithms that solve this problem have $O(mn)$ time complexity, where m and n are, respectively, the number of edges and vertices of the digraph (Simon [19] and Habib et al. [9]). It should be mentioned that the computation of transitive closures can be done asymptotically faster by matrix multiplication techniques

Parallel algorithms for this problem have been presented for PRAM [11,13], for arrays, trees and meshes of trees [15], for highly scalable multiprocessors [20,21], for BSP/CGM [5,7,20], and for other architectures [2,8,12,14,16,17]. The aproaches used by these algorithms include single and all pairs shortest paths [8,12], breadth first search [5], matrix multiplication and path decomposition [7]. Some of these algorithms were implemented on hypercubes [12], cluster of workstations [5,7] FPGAs [2] and GPGPUs [3,10,14].

In this paper, we present a parallel algorithm to compute the transitive closure of a digraph using the BSP/CGM model with p processors each with n^2/p local memory. One of the big issues when designing BSP/CGM algorithms is the number of communication rounds. If the input graph is an acyclic digraph with its vertices labeled with a so-called linear extension our algorithm requires $\log p + 1$ communication rounds and uses $O(n^3/p)$ local computation. For general digraphs, the algorithm requires p rounds. The main idea of the algorithm is the partitioning of the digraph into p pieces and the construction of a local transitive closure for each one of these pieces. These local transitive closures can be obtained by any sequential transitive closure algorithm. We have implemented the algorithm using the well-known Warshall's [23] transitive closure algorithm. Besides the fact that it uses more local computation time than other algorithms, it is very easy to deal with the communication between the processors in this algorithm. Without the linear extension labeling, we can no longer guarantee the bound of $\log p + 1$ communication rounds and uses p communication rounds in the worst case. Nevertheless, in our experiments, all the test graphs required less than $\log p + 1$ communication rounds. We also present a particularly elaborated example where the algorithm uses more than $O(\log p)$ communication rounds to solve the transitive closure problem. The algorithm was tested with two Beowulfs parallel computers, each with 64 nodes, using the MPI library. The obtained speedups indicate efficiency and scalability of the algorithm. The execution times of our algorithm are better than other parallel implementations such as [3,14,16,17]. Preliminary versions of the results (for acyclic graphs and general digraphs) presented in this paper have appeared in [4] and [1].

2 Notation, terminology and computational model

Let $D = (V, A)$ be a digraph, with vertex set V and edge set A, $|V| = n$ and $|A| = m$. Let $S \subseteq V$. Denote by $D(S)$ the digraph formed exactly by the (directed) edges of D, having at least one of its end points in S. If Q is a path in D denote its length by $|Q|$. If D is acyclic, a *linear extension* L of D is a sequence $\{v_1, \ldots, v_n\}$ of its vertices, such that $(v_i, v_j) \in A \Rightarrow i < j$.

The *transitive closure* of D is the digraph D^t, obtained from D by adding an edge (v_i, v_j), if there is a path from v_i to v_j in D, for all $v_i, v_j \in V$.

In this paper, we use the BSP/CGM (Bulk Synchronous Parallel/Coarse Grained Multicomputer) Model [6,22]. Let N be the input size of the problem. A BSP/CGM computer consists of a set of p processors $P_1, \ldots, P_p$ with $O(N/p)$ local memory per processor and each processor is connected by a router that can send/receive messages in a point-to-point fashion. A BSP/CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. The term *coarse grained* means the size of the local memory is much larger than $O(1)$. Dehne et al. [6] define "much larger"' as $N/p \gg p$.

In a computing round, we usually use the best sequential algorithm in each processor to process its data locally. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. Each processor p_i sends/receives at most $O(N/p)$ data in each communication round. In the BSP/CGM model, the communication cost is modeled by the number of communication rounds, which we wish to minimize.

3 The parallel algorithm for transitive closure

Algorithm 1 below is a parallel algorithm [4] for constructing the transitive closure of D, with n vertices and m edges, using p processors, where $1 \le p \le n$. It uses a sequential algorithm to compute the transitive closure in each processor.

The following proposition asserts the correctness of the algorithm.

Algorithm 1 Parallel transitive closure algorithm

```
Require: (i) An acyclic digraph $D = (V, A)$, with $|V| = n$ vertices and $|A| = m$. (ii) $p$
    processors.
Ensure: $D^t$, the transitive closure of $D$.
1: Let $S_1, \ldots, S_p$ be a partition of $V$, whose parts have cardinalities as equal as possible, and
    where each $S_j$ is formed by consecutive vertices in $L$.
    For $j = 1, \ldots, p$, assign the vertices of $S_j$ to processor $j$;
2: In parallel, each processor $j$ sequentially:
    . constructs the digraph $D(S_j)$ from $D$
    . computes the transitive closure $D^t(S_j)$ of $D(S_j)$
    . includes in $D$ the edges of $D^t(S_j) \setminus D(S_j)$
3: After all processors have completed Step 2, verify if $D^t(S_j) = D(S_j)$, for all processors $j$.
    If true, the algorithm terminates and $D = D^t$ is the transitive closure of the input digraph.
    If false, go to Step 2.
```

Theorem 1 *If the input digraph D is acyclic and labeled with a linear extension L of D, Algorithm 1 computes the transitive closure of D. Moreover, it requires at most $1 + \lceil \log p \rceil$ iterations of Step 2.*

Proof Denote by D_i the digraph D at the end of the i-th iteration of Step 2. Let D_0 be the input digraph and D_i^t the transitive closure of D_i, $i = 0, 1, \ldots$. Since $D_i(S_j)$ is a subgraph of D_i, it follows that all edges of the transitive closure $D_i^t(S_j)$ of $D_i(S_j)$ also belong to D_i^t, and hence to D_0^t. Consequently, in order to show that the algorithm correctly computes the transitive closure D_0^t of D_0, it suffices to show that every edge of D_0^t is also an edge of some $D_i^t(S_j)$.

With this purpose, let $(v, w) \in A(D_0^t)$. We show that $(v, w) \in A(D_i^t(S_j))$, for some i, j. Because (v, w) is an edge of D_0^t, D_0 contains a path $z_1, \dots, z_\ell$, from $v = z_1$ to $w = z_\ell$. For each k, $1 \le k \le \ell$, denote by $P(z_k)$ the processor to which z_k is assigned. Because the assignment of the vertices to the processors obeys a linear extension ordering, it follows that $P(z_1), \dots, P(z_\ell)$ is a non-decreasing sequence. Therefore, vertices assigned to a same processor are consecutive in the sequence. Consequently, after completing the first iteration of Step 2, we know that D_1 contains a $v - w$ path Q_1, formed solely by (a subset of) vertices of $z_1, \dots, z_\ell$, assigned to distinct processors. Hence $|Q_1| \le p$. If $|Q_1| = 1$ then $(v, w) \in A(D_1^t(s_j))$, implying the correctness of the algorithm. Otherwise, let z'_{k-1}, z'_k, z'_{k+1} be three consecutive vertices in Q. Let $P(z'_k) = j$. Consequently, $(z'_{k-1}, z'_k), (z'_k, z'_{k+1}) \in A(D_1(S_j))$. The latter means that at the end of the second iteration of Step 2, $(z'_{k-1}, z'_{k+1}) \in A(D_2)$. Consequently, D_2 contains a $v - w$ path Q_2, formed by a subset of vertices of Q_1 satisfying $|Q_2| = \lceil |Q_1|/2 \rceil$. By induction, it follows that $|Q_{\lceil log|Q_1|+1 \rceil}| = 1$, that is, $(v, w) \in A(D^t_{\lceil \log |Q_1|+1 \rceil}(S_j))$, as required. Moreover, no more than $1 + \lceil \log p \rceil$ iterations of Step 2 are needed.

We now analyze the complexity of the algorithm. Basically, the algorithm consists of at most $1 + \lceil \log p \rceil$ parallel computations of a sequential transitive closure algorithm. We employ a sequential algorithm whose complexity is the product of the number of vertices and edges of the digraph [9,19]. Consider a worst case example, where D consists of a single path. Then D^t is a complete acyclic digraph. In this case, each processor j may compute the transitive closure $D^t(S_j)$ of a digraph $D(S_j)$, where $|V(D(S_j))| = n$ and $|A(D^t(S_j))| = (n/p)(n - n/p) = O(n^2/p)$. Since at most $1 + \lceil \log p \rceil$ iterations are required, the overall complexity is $O(\frac{n^3 \log p}{p})$.

Finally, observe that each processor handles at most $O(n^2/p)$ directed edges. Therefore, the algorithm fits into the BSP/CGM model.

If the vertices of the digraph D are not labeled with a linear extension, the bound of $\log p + 1$ communication rounds, however, is no longer valid. In fact, there exist cases where more than $\log p + 1$ communication rounds are required. For instance, consider $p = 4$ and a graph that is the following linear list of $n = 16$ vertices (each labeled with two digits): $11 \to 31 \to 41 \to 21 \to 32 \to 12 \to 42 \to 33 \to 22 \to 43 \to 13 \to 23 \to 34 \to 14 \to 24 \to 44$. Assume that each vertex labeled with digits ij is stored in processor i. It can be shown that Algorithm 3 will need more than $\log 4 + 1 = 3$ communication rounds. The above is an elaborated example. In practice, however, this algorithm behaves efficiently, as shown in our experiments.

Corollary 1 *For a general digraph, Algorithm 1 computes correctly the transitive closure of the input digraph. Moreover, it requires at most $O(p)$ communication rounds.*

4 An implementation of the transitive closure BSP/CGM algorithm

Algorithm 2 Warshall's algorithm

```
Require: Adjacency matrix $M_{n \times n}$ of digraph $D$
Ensure: $D^t$ the transitive closure of $D$
for $k \leftarrow 1$ until $n$ do
    for $i \leftarrow 1$ until $n$ do
        for $j \leftarrow 1$ until $n$ do
            $M[i,j] \leftarrow M[i,j]$ or $(M[i,k]$ and $M[k,j])$
        end for
    end for
end for
```

Consider a digraph $D = (V, E)$ where $|V(D)| = n$ and $|E(D)| = m$. Let us apply Warshall's algorithm to this digraph. Let the vertices be $1, 2, \dots, n$. The input to the algorithm is an $n \times n$ Boolean adjacency matrix M of D, that is, the entry $M_{i,j}$ is 1 if there is a directed edge from vertex i to vertex j and 0 otherwise. Warshall's algorithm consists of a simple nested loop that transforms the input matrix M into the output matrix. The main idea is that if entries $M_{i,k}$ and $M_{k,j}$ are both 1, then we should set $M_{i,j}$ to 1. This following describes this method. The complexity is $O(n^3)$, as already mentioned.

Algorithm 3 Parallel Transitive Closure for a General Digraph

```
Require: Adjacency matrix $M$ stored in the $p$ processors: each processor $q$ $(1 \le q \le p)$ stores
    submatrices $M[(q-1)\frac{n}{p}+1..q\frac{n}{p}, 1..n]$ and $M[1..n, (q-1)\frac{n}{p}+1..q\frac{n}{p}]$.
Ensure: Transitive closure of digraph $D$ represented by the transformed matrix $M$.
    Each processor $q$ $(1 \le q \le p)$ does the following.
repeat
    for $k = (q-1)\frac{n}{p}+1$ until $q\frac{n}{p}$ do
        for $i = 0$ until $n-1$ do
            for $j = 0$ until $n-1$ do
                if $M[i,k] = 1$ and $M[k,j] = 1$ then
                    $M[i,j] = 1$ (if $M[i,j]$ belongs to processor different from $q$ then store it for
                    subsequent transmission to the corresponding processor.)
                end if
                Send stored data to the corresponding processors.
                Receive data that belong to processor $q$ from other processors.
            end for
        end for
    end for
until no new matrix entry updates are done
```

Algorithm 3 is an implementation of Algorithm 1 by incorporating a parallelized version of Warshall's Algorithm. First,

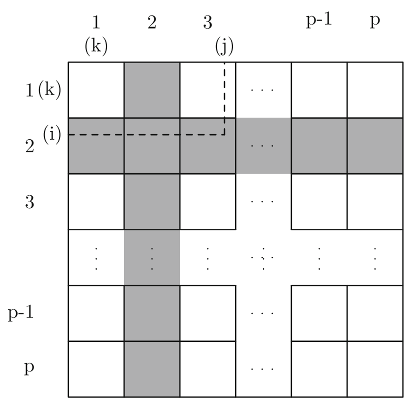


Fig. 1 Matrix *M* partitioned into *p horizontal* and *p vertical stripes*

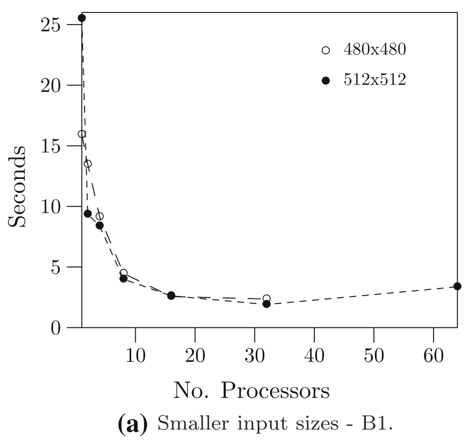
partition and store the rows and columns of the adjacency matrix M in the p processors as follows. Divide the adjacency matrix M into p horizontal and p vertical stripes. Assume for simplicity that n/p is an integer, that is, n divides p exactly. Each horizontal stripe has thus n/p rows. Likewise each vertical stripe has n/p columns. This is shown in Fig. 1. Consider processors numbered as $1, 2, \ldots, p$. Then processor q stores the horizontal stripe q and the vertical stripe q. Notice that each adjacency matrix entry is thus stored in two processors.

Observe the following property of the way the matrix M is stored in the processors. Both $M[i, k]$ and $M[k, j]$ for any k are always stored in a same processor (see Fig. 1). This makes the test indicated at line 5 of the algorithm very easy to perform. If both $M[i, k]$ and $M[k, j]$ are equal to 1 then $M[i, j]$ must also be made equal to 1. In this case, the update of $M[i, j]$ may be done immediately if it is also stored in processor q. Otherwise, the update will be done in the next communication round.

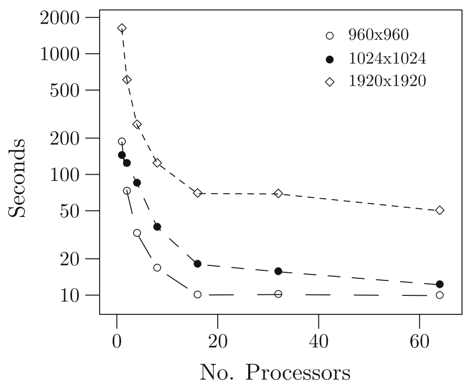
5 Experimental results of Algorithm 3

We implemented Algorithm 3 on two Beowulf clusters cluster of 64 nodes. The first one (B1) consisting of low cost microcomputers with 256MB RAM, 256MB swap memory, CPU Intel Pentium III 448.956 MHz, 512KB cache. The nodes are connected through a 100 Mb fast-Ethernet switch. The second one (B2) consisting of 256 processors (64 nodes with 4×2.2 GHz Opteron Cores) with 8 GB RAM per node. All nodes are interconnected via a Foundry SuperX switch using Gigabit ethernet. Our code is written in standard ANSI C using the MPI library.

The test inputs consist of randomly generated digraphs where there is a 20% probability of having an edge between two vertices. We tested graphs with $n = 480, 512, 960, 1{,}024, 1{,}920, 2{,}048, 4{,}096$ and $6{,}144$ vertices. In all the tests the number of communication rounds required were less than $\log p$.



(a) Smaller input sizes - B1.



(b) Larger input sizes - B1.

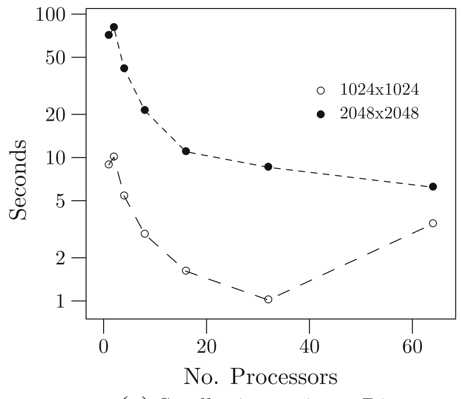
Fig. 2 Execution times for various input sizes—B1

The times obtained for different problem sizes are shown in Figs. 2 and 3, while the corresponding speedups are shown in Fig. 4. Notice the speedup curve (B1) for the input size of 1,920×1,920. It presents two ascending parts with different slopes. The first ascending part of the speedup can be seen to be superlinear. This is due to the memory swap overhead which is more severe for the sequential case ($p = 1$) which has to store the entire $n \times n$ input matrix. On the other hand, the parallel cases deal with input matrices of smaller size $n/p \times n/p$.

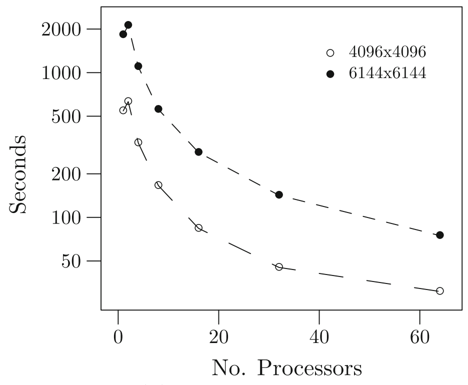
Using only one processor of Bewoulf B1 and the input size of 1,920×1,920 our implementation takes 1,614.60 s to compute the transitive closure, while Bewoulf B2 takes 70.95 s for the input size 2,048 ×2,048. As Fig. 4 shows, in both Bewoulfs the algorithm is scalable.

For comparison purposes, we have used the Bewoulf B1 since it uses a configuration that is similar to the other implementations [16,17]. Our implementation (Bewoulf B1) also considers input matrices that include the same input sizes used in [16,17], namely, 480 × 480, 960 × 960, and 1,920×1,920.

The results presented by Pagourtzis [16,17] were obtained on a cluster Hewlett Packard 720 with 20 nodes. The nodes

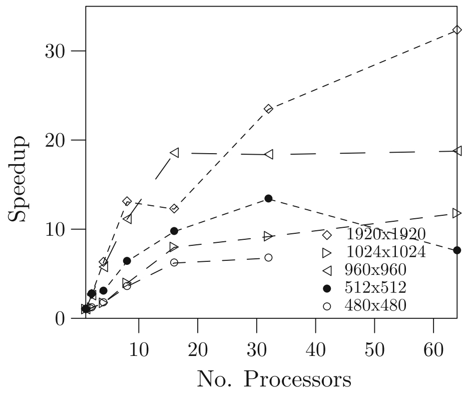


(a) Smaller input sizes - B2.

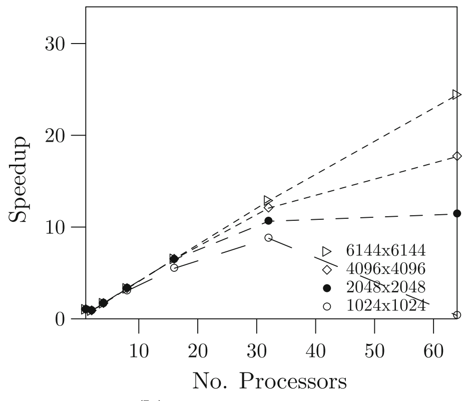


(b) Larger input sizes - B2.

Fig. 3 Execution times for various input sizes—B2



(a) Smaller input sizes - B1.



(b) Larger input sizes - B2.

Fig. 4 Speedups for various input sizes

were connected through a 10 MB Ethernet switch, and they used the PVM library (version 3.4.6.). They present tables and charts where the algoritms were executed with 36 work-processes (more than one process in each node).

Our implementation results using Beowulf B1 achieved better execution times. In particular we observe the 1,920×1,920 case where our implementation obtains very substantial speedups. We observe that in our tests we have used the MPI library with just one process in each node and our nodes were connected with a faster network.

We also considered matrices with equivalent sizes as those used in [3,14] (Beowulf B2). Our implementation results are competitive.

6 Conclusion

We have described a parallel algorithm for computing the transitive closure of a digraph, using p processors. Algorithm 1 is suitable for the BSP/CGM model. It employs at most $1 + \lceil \log p \rceil$ iterations of a sequential transitive closure algorithm if the vertices of the digraph are labeled using a linear extension. For general digraphs, the algorithm requires p communication rounds in the worst case.

Using the algorithm of Warshall to compute the local transitive closures we implemented the algorithm with MPI on Bewoulfs. It can also be viewed as a parallelized version of Warshall's algorithm. In fact, any other sequential transitive closure algorithm could be applied.

Although in theory for general digraphs the algorithm no longer guarantees the $O(\log p)$ bound on the number of communication rounds, all the graphs we tested used at most $\log p+1$ communication rounds. The implementation results are very promising and show the efficiency and scalability of the proposed modified algorithm, and compare favorably with other parallel implementations.

References

1. Alves C, Cáceres E Jr, de Castro A, Song S, Szwarcfiter J (2003) Efficient parallel implementation of transitive closure of digraphs. In: Proceedings of the EuroPVM/MPI 2003. Lecture Notes in Computer Science, vol 2840. Springer, Berlin, pp 126–133
2. Bondhugula U, Devulapalli A, Fernando J, Wyckoff P, Sadayappan P (2006) Parallel fpga—based all-pairs shortest-paths in a directed graph. In: Proceedings of the 20th IEEE international parallel and distributed processing symposium. IEEE, New York

3. Buluç A, Gilbert JR, Budak C (2010) Solving path problems on the GPU. Parallel Comput 36(5–6):241–253
4. Cáceres EN, Song SW, Szwarcfiter JL (2002) A parallel algorithm for transitive closure. In: Proceedings of the 14th IASTED international conference on parallel and distributed computing and systems, Cambridge, USA, pp 114–116
5. Cáceres EN, Vieira CCA (2007) Bsp/cgm algorithms for the transitive closure problem. In: Proceedings of the 2007 high performance computing and simulation conference (HPCS 2007), Prague, Czech Republic, pp 718–723
6. Dehne F (1999) Coarse grained parallel algorithms. Algorithmica 24:173–426
7. Gibbons A, Pagourtzis A, Potapov I, Rytter W (2003) Coarse-grained parallel transitive closure algorithm: path decomposition technique. Comput J 46(4):391–400
8. Grama A, Kumar V, Gupta A, Karypis G (2003) An introduction to parallel computing: design and analisys of algorithms, 2nd edn. Addison-Wesley, Boston
9. Habib M, Morvan M, Rampon JX (1993) On the calculation of transitive reduction–closure of orders. Discrete Math 111:289–303
10. Harish P, Narayanan P (2007) Accelerating large graph algorithms on the gpu using cuda. In: High performance computing 2007. Lecture Notes in Computer Science, vol 4873. Springer, Berlin, pp 197–208
11. JáJá J (1992) Introduction to parallel algorithms. Addison-Wesley, Boston
12. Jenq J, Sahni S (1987) All pairs shortest paths on a hypercube multiprocessor. In: Proceedings of the 1987 IEEE international conference on parallel processing, pp 713–716
13. Karp RM, Ramachandran V (1990) Handbook of theoretical computer science, vol. A, chap. 17. MIT Press/Elsevier, Cambridge, pp 869–941
14. Katz GJ Jr, Kider JT (2008) All-pairs shortest-paths for large graphs on the gpu. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware. ACM, New York, pp 47–55
15. Leighton F (1992) Introduction to parallel algorithms and architectures: arrays–trees–hypercubes. Morgan Kaufmann Publishers, Burlington
16. Pagourtzis A, Potapov I, Rytter W (2001) Observations on parallel computation of transitive and max-closure problems. In: Cotronis Y, Dongarra J (eds) Proceedings Euro PVM/MPI 2001. Lecture Notes in Computer Science, vol 2131. Springer, Berlin, pp. 249–256
17. Pagourtzis A, Potapov I, Rytter W (2002) Pvm computation of the transitive closure: the dependency graph approach. In: Kranzlmuller D et al (ed) Proceedings Euro PVM/MPI 2002. Lecture Notes in Computer Science, vol 2474. Springer, Berlin, pp 217–225
18. Roy B (1959) Transitivité et connexité. C R Acad Sci Paris 249:216–218
19. Simon K (1988) An improved algorithm for transitive closure on acyclic digraphs. Theor Comput Sci 58:325–346
20. Tiskin A (2001) All-pairs shortest paths computation in the bsp model. In: Proceedings of the ICALP. Lectures Notes in Computer Science, vol 2076. Springer, Berlin, pp 178–189
21. Toptsis A (1991) Parallel transitive closure for multiprocessor. In: Proceedings of the international conference on computing and information. Lecture Notes in Computer Science, vol 497. Springer, Berlin, pp 197–206
22. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33:103–111
23. Warshall S (1962) A theorem on boolean matrices. J Assoc Comput Mach 11–12