A Comparison of two Parallel Iterative Algorithms for Solving Path Problems

Robert Manger

Department of Mathematics, University of Zagreb, Zagreb, Croatia

Path problems are a family of optimization and enumeration problems posed on a directed graph. General algorithms for solving path problems can be designed as counterparts of the traditional iterative methods for solving linear systems. In this paper two parallel iterative Gauss-Seidel-like algorithms for solving path problems are compared. Theoretical results are listed, which estimate the computational complexity of both algorithms. Experiments are presented, where the algorithms have been tested on randomly generated graphs and with different numbers of available processors. Some situations are identified, where one of the algorithms becomes superior to the other.

Keywords: directed graphs, path problems, parallel algorithms, iterative methods, complexity, experiments.

1. Introduction

Path problems are frequently encountered in operations research, computer science, electronics and other fields. These problems take different forms and variants, but in essence they all reduce to determination and evaluation of paths in a directed graph. The most famous example is the shortest path problem, stated as follows: in a graph whose arcs are given lengths determine the shortest path between two given nodes. A similar task is to find the most reliable path between two nodes, or the path of maximum capacity. In addition to these optimization problems, there are also some enumerative path problems, e.g. checking the existence of a path between two given nodes, or listing all possible paths, etc.

There are many ways how to treat path problems. The algebraic approach (Carré, 1979; Rote, 1990) tries to find a general formulation for the whole family. An abstract algebraic structure is chosen, so that each particular problem is described through an appropriate concrete instance of that structure. Also, algorithms for solving path problems are interpreted as counterparts of the classical methods for solving linear systems. In this way, iterative algorithms for solving path problems are introduced, which resemble the classical Jacobi or Gauss-Seidel method (Press, 1988). Special cases of such algorithms are known in various contexts under different names, e.g. the Bellman or Ford algorithm for shortest paths (McHugh, 1990).

The Gauss-Seidel algorithm is generally regarded as better than Jacobi, since it usually requires less iterations. However, a possible drawback of Gauss-Seidel is that it seems to be inherently sequential, and therefore not very suitable for parallel computing. Still, parallel versions of Gauss-Seidel can be defined, as we have shown in a previous paper (Manger, 1993). But frankly, all those versions make some kind of a compromise: either with the original Gauss-Seidel idea, or with the efficiency of parallel computing. It is very hard to say, which version should be regarded as the best one.

The aim of this paper is to make a comparison of different, parallel Gauss-Seidel algorithms for solving path problems. In fact, two representative algorithms from (Manger,

1993), using completely different parallelization strategies, will be evaluated. The first algorithm is a straightforward parallelization: it performs a single iteration step very efficiently, but it slightly degrades the original Gauss-Seidel computational procedure, thus requiring more iterations. The second algorithm is a more sophisticated parallelization: it is computationally equivalent to the sequential Gauss-Seidel, but its execution of a single iteration step is not so efficient. Our comparison should reveal relative advantages and drawbacks of the two parallelization strategies, in order to decide which strategy is better and when.

The paper is organized as follows. Section 2 reviews the algebraic approach to path problems, and shows through an example, how a concrete path problem can be interpreted as an equation. Section 3 introduces our two parallel iterative Gauss-Seidel-like algorithms for solving such equations; some basic properties of the algorithms are highlighted. Section 4 lists the available theoretical results, which estimate and compare the computational complexity of both algorithms; it is demonstrated by examples that those results cannot generally be improved. The next Section 5 presents our experiments and summarizes their results; in those experiments the two algorithms have been tested and compared on randomly generated graphs, and with different numbers of available processors. The final Section 6 gives conclusions.

2. Algebraic Approach to Path Problems

Suppose that we want to solve a *single-destination path problem*, in a given directed graph with n nodes. That means, we are interested only in paths terminating in a specified (fixed) node. Then, according to (Carré, 1979), the problem can be reduced to solving a vector equation of the form

$$\mathbf{y} = A \circ \mathbf{y} \vee \mathbf{b}. \qquad (1)$$

Here, A is an $n \times n$ matrix specifying the graph, $\mathbf{b}$ is a vector of length n pointing to the destination node, and $\mathbf{y}$ is an unknown vector (also of length n) describing the solution of the original problem. Entries of A, $\mathbf{b}$ and $\mathbf{y}$ are elements of a suitably chosen set P, which is called a *path algebra*. P is equipped with two binary operations, $\vee$ and $\circ$, satisfying certain algebraic properties. $\vee$ is a replacement for standard addition, and $\circ$ serves as multiplication. Neutral elements exist for both operations, and they are called the zero and the unit element respectively. Similarly, as in ordinary linear algebra, the "scalar" operations $\vee$ and $\circ$ induce the corresponding matrix and vector operations.

To illustrate these ideas, let us consider the graph in Figure 1 whose arcs are given "lengths". Suppose that we want to solve a *shortest distance problem*, i.e. we want to determine the length of the shortest path from any node to node 5 (denoted by a double circle). Then the equation (1) should be given as follows.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} \infty & \infty & 3 & \infty & \infty \\ 1 & \infty & \infty & 8 & \infty \\ 3 & 1 & \infty & \infty & 1 \\ \infty & 2 & 7 & \infty & \infty \\ \infty & \infty & 6 & 2 & \infty \end{bmatrix} \circ \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} \vee \begin{bmatrix} \infty \\ \infty \\ \infty \\ \infty \\ 0 \end{bmatrix}.$$

The set P consists here of real numbers extended with ∞. The operation $\vee$ is the standard minimum, and $\circ$ is the conventional summation. The zero in P is ∞, and the unit element is 0. The matrix A is in fact the adjacency matrix of our graph, i.e. the (i,j)-th matrix entry is the length

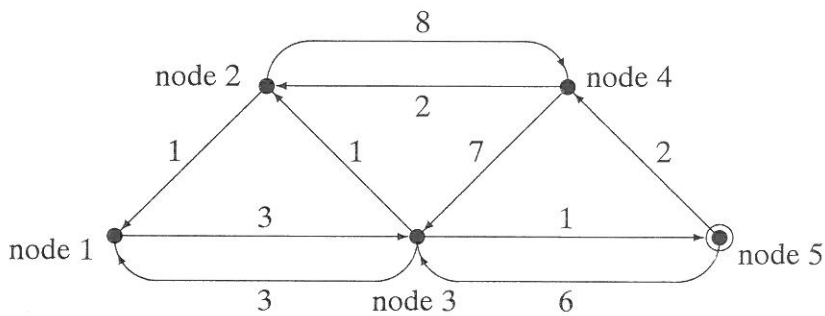

Fig. 1. A simple shortest distance problem.

of the arc from node i to node j. The vector $\mathbf{b}$ is the 5-th unit vector, corresponding to our chosen destination node 5. It is easy to check that the vector

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 1 \\ 7 \\ 0 \end{bmatrix}$$

is a solution of our equation. From this vector we read the solution of our path problem, i.e. the i-th vector element is the shortest distance from node i to node 5. Many other examples of path problems and corresponding path algebras can be found in (Carré, 1979; Manger, 1990).

3. Two Parallel Iterative Algorithms

As in ordinary linear algebra, the equation (1) can be solved by iteration. Thus an initial approximate solution vector $\mathbf{y}^{(0)}$ is chosen and put into the right-hand side of (1) in order to evaluate the next solution vector $\mathbf{y}^{(1)}$. The obtained vector $\mathbf{y}^{(1)}$ is used in the same way to evaluate $\mathbf{y}^{(2)}, \ldots$ and so on. After some number of iterations, the process will converge and an exact solution will be reached.

In the *Jacobi* iterative method, each element of $\mathbf{y}^{(k)}$ is computed by using strictly the elements of $\mathbf{y}^{(k-1)}$. The *Gauss-Seidel* method, on the other hand, evaluates an element of $\mathbf{y}^{(k)}$ by using the most recent values of the other elements, no matter if they belong to $\mathbf{y}^{(k-1)}$ or already to $\mathbf{y}^{(k)}$. So in the Gauss-Seidel method, the final value of $\mathbf{y}^{(k)}$ depends on the order in which particular elements of $\mathbf{y}^{(k)}$ have been evaluated.

Now we present two parallel iterative algorithms for solving the equation (1). Both algorithms are based on the Gauss-Seidel type of iteration, and they are designed for a *multiprocessor* built of many independent processors with a common memory (Quinn, 1987).

Algorithm 1.
(∗ *P is any path algebra. m processors are available. Input: an* $n \times n$ *matrix* $A=[a_{ij}]$ *over P, and a vector* $\mathbf{b} = [b_i]$ *of length n over P.* ∗)

$k := 0$;
$l := \lceil n/m \rceil$;
$\bar{m} := \lceil n/l \rceil$;
for $i := 1$ **to** n **do**
  $y_i^{(0)} := b_i$;
repeat
  $t := true$;
  $k := k + 1$;
  for all $s \in \{1, 2, \ldots, \bar{m}\}$ **do in parallel**
    for $i := (s-1)l + 1$ **to** $\min\{sl, n\}$ **do**
      begin
        $y_i^{(k)} := (\bigvee_{j=1}^{(s-1)l} a_{ij} \circ y_j^{(k-1)}) \vee$
          $(\bigvee_{j=(s-1)l+1}^{i-1} a_{ij} \circ y_j^{(k)}) \vee$
            $(\bigvee_{j=i}^{n} a_{ij} \circ y_j^{(k-1)}) \vee b_i$;
        if $y_i^{(k)} \neq y_i^{(k-1)}$ **then**
          $t := false$
      end
  until $t = true$
(∗ *Output: the vector* $\mathbf{y} = [y_i^{(k)}]$. ∗)

Algorithm 2.
(∗ *P is any path algebra. m processors are available. Input: an* $n \times n$ *matrix* $A = [a_{ij}]$ *over P, and a vector* $\mathbf{b} = [b_i]$ *of length n over P.* ∗)

$l := \lceil n/m \rceil$;
$\bar{m} := \lceil n/l \rceil$;
for $r := 1$ **to** n **do**
  begin
    $y_r := b_r$;
    $w_r := b_r$;
    $z_r := zero\ in\ P$
  end ;
for $i := 1$ **to** n **do**
  for all $s \in \{1, 2, \ldots, \bar{m}\}$ **do in parallel**
    for $r := (s-1)l + 1$ **to** $\min\{sl, n\}$ **do**
      if $r \leq i$ **then**
        $w_r := w_r \vee a_{ri} \circ y_i$
      else
        $z_r := z_r \vee a_{ri} \circ y_i$;
repeat
  $t := true$;
  for $i := 1$ **to** n **do**
    begin
      $\bar{y}_i := w_i \vee z_i$;
      if $\bar{y}_i \neq y_i$ **then**
        begin
          $y_i := \bar{y}_i$;
          $t := false$;
          for all $s \in \{1, 2, \ldots, \bar{m}\}$ **do**
          in parallel

```
                    for r := (s − 1)l + 1 to
                    min{sl, n} do
                      if r ≤ i then
                        w_r := w_r ∨ a_ri∘y_i
                      else
                        z_r := z_r ∨ a_ri∘y_i
              end
        end
    until t = true
(* Output: the final value of the vector y =
[y_i]. *)
```

Algorithm 1 is a straightforward parallelization of the sequential Gauss-Seidel. However, due to parallel computing, the obtained sequence of vectors $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(k)}, \ldots$, is not necessarily the same as in the sequential case. In fact, if the number of processors m is equal to n, Algorithm 1 degrades into a Jacobi type of iteration. For $1 < m < n$ Algorithm 1 acts as a hybrid of Jacobi and Gauss-Seidel.

Algorithm 2 uses a more sophisticated parallelization, where the inner loop of a single iteration has been parallelized instead of the outer loop. In this way, the original (sequential) Gauss-Seidel order of evaluation has been preserved, i.e. the same sequence of approximate solution vectors is obtained for any number of processors.

It can be shown (Manger, 1993) that both Algorithms 1 and 2 are correct when applied to path problems, i.e. they terminate in a finite number of iterations, and they really find the solution corresponding to our original path problem.

4. Estimates of Computational Complexity

In order to compare our algorithms for computing in a path algebra P, we introduce a theoretical measure of performance called *computational complexity*. This is the time required by a given algorithm to solve a given problem, provided that one computational operation with elements of P (i.e. $\vee$, $\circ$, equality test) takes one unit of time, and all other operations take no time. Computational complexity is usually expressed as a function of n (number of graph nodes) and m (number of processors).

The computational complexity of Algorithms 1 and 2 has already been studied in (Manger, 1993). Also relevant are some theorems from (Carré, 1979) treating iterative methods in general. All those results combined together can be summarized as follows.

1. The computational complexity of a single iteration in Algorithm 1 is at most $\lceil n/m \rceil (2n + 1)$.
2. Algorithm 1 never requires more than n iterations to terminate.
3. The computational complexity of a single iteration in Algorithm 2 is at most $2n(\lceil n/m \rceil + 1)$.
4. Algorithm 2 never requires more iterations to terminate than Algorithm 1 (for the same problem and the same number of processors).

To be precise, statements 2 and 4 above are not generally true, but only if Algorithms 1 and 2 are used with correct input data describing a meaningful path problem. Moreover, the graph involved must have an additional property called absorptivity (Carré, 1979). Still, almost all path problems lead to absorptive graphs, and therefore in our context the presented results are practically always valid.

Remember that with one processor both algorithms are computationally equivalent to the original (sequential) Gauss-Seidel method, while with more processors only Algorithm 2 is still guaranteed to be equivalent. Thus, if only one processor is used, then both algorithms need the same number of iterations for the same problem. As more processors are added, the number of iterations possibly increases for Algorithm 1, but remains unchanged for Algorithm 2.

As we can see, the estimates of complexity implied by the quoted results are too vague to determine which of Algorithms 1 and 2 is better. Namely, Algorithm 1 seems to execute a single iteration faster, but perhaps requires more iterations. Also, it is hard to compare our two algorithms only on the basis of worst-case complexities, since those values can be unequally pessimistic in each concrete case. An additional property of Algorithm 2 is that it can sometimes skip almost a whole iteration (if $\bar{y}_i = y_i$). This

Fig. 2. Two simple path existence problems.

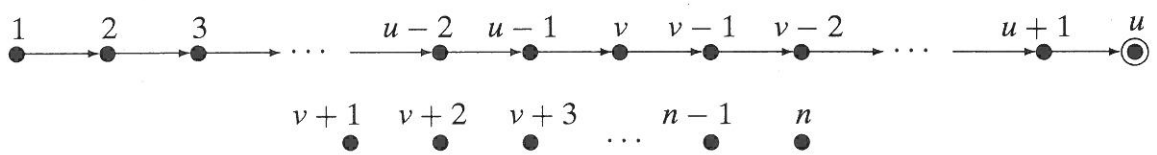

Fig. 3. Another path existence problem.

property can have a significant impact on overall performance, although it is not captured by our estimates.

One may argue that a more precise comparison of Algorithms 1 and 2 can be made simply by "sharpening" the available estimates. But we claim that statements 1-4 cannot generally be improved. Namely, it is obvious that the upper bound in statement 1 and 3 respectively cannot be lowered, since it reflects the actual number of algebraic operations required for a complete graph (a dense matrix and vector). A little bit more difficult is to demonstrate that the range given by statements 2 and 4 respectively cannot be made tighter; we will show this through a set of examples.

In each of the following examples we will consider a *path existence problem*. So our task will be to determine all nodes in a given graph that are connected by a path to a given destination node. Such a problem can again be reduced to a vector equation of the form (1), but now the path algebra P must be the Boolean algebra, consisting of only two elements 0 and 1, with the operations max and min as $\vee$ and $\circ$ respectively.

Let us first consider the problem presented by Figure 2 — left; as before the destination node is denoted by a double circle. It is easy to see that for this problem both Algorithms 1 and 2 terminate after exactly n iterations, no matter how many processors are employed. Thus Algorithm 1 can really need as many as n iterations, and Algorithm 2 can really need as many iterations as Algorithm 1.

Let us next consider the problem illustrated by Figure 2 — right; the destination node is again denoted by a double circle. For this problem Algorithm 1 executes n iterations if n processors are used, while Algorithm 2 executes only 2 iterations. Thus with a different number of processors Algorithm 1 may require a considerably different number of iterations, ranging between 2 and n. On the other hand, with the same number of processors Algorithm 2 may require much less iterations than Algorithm 1.

The previous two problems can be combined into a more general problem shown in Figure 3. The graph of Figure 3 can be constructed for any u, v, n such that $1 \leq u \leq v \leq n$. It can be checked that, for this problem and with n processors, Algorithm 1 needs v iterations, while Algorithm 2 needs $\min\{u+1, v\}$ iterations. So, we see that almost all situations implied by statements 2 and 4 together are possible.

5. Experiments and Results

To enable exact testing, we have made two simple C programs implementing Algorithm 1 and 2 respectively. These programs are in fact sequential, but each of them genuinely simulates the behaviour of the corresponding parallel algorithm. In addition to computing, each program also measures the number of iterations required, the total computational complexity, and the average complexity per iteration. Since the specification of the currently chosen path algebra P has been isolated in a separate module,

destination	# of processors	# of iterations Alg 1 : Alg 2	total complexity Alg 1 : Alg 2	avg cmplx per iter Alg 1 : Alg 2
node 1	1	2 : 2	32 : 22	16.00 : 10.00
	2	3 : 2	27 : 18	9.00 : 8.00
	3	4 : 2	30 : 17	7.50 : 8.00
	5	4 : 2	17 : 16	4.25 : 7.50
node 2	1	3 : 3	49 : 28	16.33 : 8.67
	2	3 : 3	27 : 24	9.00 : 7.67
	3	3 : 3	20 : 23	6.67 : 7.00
	5	3 : 3	11 : 22	3.67 : 7.00
node 3	1	2 : 2	34 : 23	17.00 : 10.00
	2	3 : 2	33 : 19	11.00 : 8.50
	3	3 : 2	24 : 19	8.00 : 9.00
	5	4 : 2	20 : 17	5.00 : 8.00
node 4	1	4 : 4	72 : 52	18.00 : 12.50
	2	4 : 4	37 : 45	9.25 : 11.00
	3	4 : 4	29 : 41	7.25 : 10.00
	5	5 : 4	21 : 38	4.20 : 9.25
node 5	1	3 : 3	47 : 33	15.67 : 10.67
	2	4 : 3	33 : 27	8.25 : 8.67
	3	4 : 3	27 : 27	6.75 : 8.67
	5	5 : 3	21 : 25	4.20 : 8.00

Tab. 1. Experimental results — shortest distance problems given by Figure 1.

any of the two programs can easily be adjusted to solve various concrete path problems, e.g. path existence, shortest distances, . . ., etc.

In our programs, the computational complexity is measured simply by counting operations in each of the "parallel" processes. More precisely, only non-trivial operations with elements of P are taken into account, i.e. operations with both operands non-null. This measurement method is appropriate for prospective professional implementations of our algorithms, where sparse data structures should be used for matrices and vectors (Tewarson, 1973). Such structures do not store zeros, and therefore allow trivial operations to be easily recognized and avoided. It is still assumed, however, that a non-trivial operation always produces a non-trivial result; this is namely true in most path algebras.

The programs have been tested on a number of path existence and shortest distance problems. Different numbers of processors have been tried for each problem. The measured iteration counts and computational complexities are summarized in Tables 1-4. The first two tables present the exact results for few smaller but specially designed problems. The next two tables contain average results for populations of larger problems generated randomly.

Table 1 shows the values for the shortest distance problem given by Figure 1, and also for the remaining four problems that use the same graph but another destination node. These examples, although very simple, clearly illustrate some general characteristics of Algorithms 1 and 2. We see that with one processor Algorithm 2 always has a considerably smaller complexity than Algorithm 1. With more proces-

sors, Algorithm 2 performs quite predictably, i.e. the number of iterations remains constant, and the complexity slowly decreases. At the same time, the behaviour of Algorithm 1 is more unstable, since its number of iterations can increase. For the second sample problem (destination node 2) Algorithm 1 never requires more iterations than Algorithm 2, thus, with more processors, it becomes much faster. In the first or third example Algorithm 1 is always slower than Algorithm 2, due to a greater number of iterations. Finally, in the fourth or fifth example Algorithm 1 requires few more iterations, but still ends up faster than Algorithm 2.

Table 2 shows the measurements for the path existence problem of Figure 3. The parameters v and n have been set to 10, and few different values for u have been tried. We see that the number of iterations required by our algorithms can really be regulated through u and v, as claimed earlier. Another interesting thing to note is, that for this particular problem Algorithm 2 never manages to employ more than one processor, i.e. its complexity does not change with more processors.

Table 3 summarizes the results for 50 path existence problems, all of the same size $n = 100$. The problems have been divided into 5 groups of 10, according to their graph density (percentage of non-zeros in the adjacency matrix A). For each problem, the graph structure with the desired density was generated randomly. Also, the destination node was chosen randomly. Entries in Table 3 in fact correspond to groups, not

value of u	# of pro -cessors	# of iterations Alg 1 : Alg 2	total complexity Alg 1 : Alg 2	avg cmplx per iter Alg 1 : Alg 2
1	1	2 : 2	29 : 20	14.50 : 9.50
	2	3 : 2	24 : 20	8.00 : 9.50
	3	4 : 2	27 : 20	6.75 : 9.50
	4	5 : 2	26 : 20	5.20 : 9.50
	5	6 : 2	21 : 20	3.50 : 9.50
	10	10 : 2	19 : 20	1.90 : 9.50
4	1	5 : 5	74 : 44	14.80 : 8.60
	2	6 : 5	47 : 44	7.83 : 8.60
	3	6 : 5	44 : 44	7.33 : 8.60
	4	7 : 5	38 : 44	5.43 : 8.60
	5	7 : 5	26 : 44	3.71 : 8.60
	10	10 : 5	19 : 44	1.90 : 8.60
7	1	8 : 8	101 : 59	12.62 : 7.25
	2	8 : 8	67 : 59	8.38 : 7.25
	3	9 : 8	51 : 59	5.67 : 7.25
	4	9 : 8	47 : 59	5.22 : 7.25
	5	9 : 8	33 : 59	3.67 : 7.25
	10	10 : 8	19 : 59	1.90 : 7.25
10	1	10 : 10	109 : 64	10.90 : 6.30
	2	10 : 10	75 : 64	7.50 : 6.30
	3	10 : 10	60 : 64	6.00 : 6.30
	4	10 : 10	51 : 64	5.10 : 6.30
	5	10 : 10	36 : 64	3.60 : 6.30
	10	10 : 10	19 : 64	1.90 : 6.30

Tab. 2. Experimental results - path existence problems of Figure 3 ($v = n = 10$).

to particular problems. Most of those entries are averages computed over a whole group. A score is simply the number of members of a group where one algorithm outperformed the other. Thus Table 3 shows a typical behaviour of Algorithms 1 and 2 on a path existence problem. This behaviour is expressed in dependence on the graph density and on the number of available processors.

All problems it Table 3 use a relatively sparse

graph density	# of pro -cessors	# of iterations Alg 1 : Alg 2	total complexity Alg 1 : Alg 2	avg cmplx per iter Alg 1 : Alg 2	score Alg 1 : Alg 2
1%	1	5.6 : 5.6	220.1 : 121.1	36.96 : 19.98	0 : 10
	2	7.3 : 5.6	172.8 : 115.9	21.52 : 19.16	0 : 10
	4	7.8 : 5.6	110.3 : 112.9	13.12 : 18.74	7 : 3
	7	8.2 : 5.6	87.2 : 111.5	9.89 : 18.52	8 : 2
	13	8.3 : 5.6	58.8 : 110.2	6.60 : 18.33	10 : 0
	25	8.4 : 5.6	43.0 : 109.5	4.84 : 18.23	10 : 0
	50	8.6 : 5.6	34.7 : 109.4	3.92 : 18.21	10 : 0
	100	8.6 : 5.6	28.3 : 109.4	3.25 : 18.21	10 : 0
3%	1	4.8 : 4.8	1677.5 : 799.4	349.32 : 166.38	0 : 10
	2	6.3 : 4.8	1019.8 : 680.8	161.06 : 141.59	0 : 10
	4	7.2 : 4.8	595.8 : 612.7	82.66 : 127.47	7 : 3
	7	7.7 : 4.8	391.0 : 577.6	50.72 : 120.19	10 : 0
	13	7.9 : 4.8	248.4 : 556.9	31.48 : 115.96	10 : 0
	25	8.0 : 4.8	149.3 : 542.8	18.82 : 113.05	10 : 0
	50	8.1 : 4.8	100.7 : 532.9	12.52 : 110.97	10 : 0
	100	8.1 : 4.8	71.9 : 529.2	8.99 : 110.22	10 : 0
5%	1	4.1 : 4.1	2783.5 : 1248.7	677.95 : 306.78	0 : 10
	2	4.8 : 4.1	1475.4 : 975.0	306.89 : 239.32	0 : 10
	4	5.2 : 4.1	763.5 : 821.8	145.93 : 201.22	7 : 3
	7	5.5 : 4.1	480.0 : 744.1	87.13 : 182.12	10 : 0
	13	5.7 : 4.1	299.0 : 685.2	51.80 : 167.50	10 : 0
	25	6.0 : 4.1	193.3 : 644.3	31.99 : 157.45	10 : 0
	50	6.0 : 4.1	115.7 : 616.2	19.13 : 150.53	10 : 0
	100	6.0 : 4.1	80.8 : 599.6	13.38 : 146.38	10 : 0
7%	1	3.6 : 3.6	3525.0 : 1618.2	977.20 : 453.77	0 : 10
	2	3.9 : 3.6	1625.5 : 1191.0	416.66 : 333.89	0 : 10
	4	4.3 : 3.6	832.7 : 951.8	191.83 : 266.19	7 : 3
	7	4.7 : 3.6	580.6 : 833.4	123.00 : 232.64	10 : 0
	13	4.8 : 3.6	329.1 : 746.6	68.34 : 208.30	10 : 0
	25	5.0 : 3.6	195.6 : 679.9	39.14 : 189.34	10 : 0
	50	5.0 : 3.6	115.4 : 629.8	23.05 : 175.15	10 : 0
	100	5.0 : 3.6	74.3 : 598.2	14.84 : 166.34	10 : 0
9%	1	3.1 : 3.1	3950.7 : 1974.7	1276.89 : 637.62	0 : 10
	2	3.6 : 3.1	2044.9 : 1375.4	565.42 : 443.86	0 : 10
	4	4.0 : 3.1	1098.1 : 1044.9	274.52 : 337.38	3 : 7
	7	4.1 : 3.1	651.1 : 879.3	158.42 : 284.05	9 : 1
	13	4.3 : 3.1	377.3 : 761.2	87.09 : 245.84	10 : 0
	25	4.4 : 3.1	218.1 : 670.1	49.12 : 216.35	10 : 0
	50	4.5 : 3.1	130.7 : 598.3	28.71 : 193.26	10 : 0
	100	4.5 : 3.1	82.6 : 545.1	18.20 : 176.02	10 : 0

Tab. 3. Experimental results — randomly generated path existence problems.

graph (density below 10%). Denser graphs have not been included, since they would produce very predictable results (similar to those for 9% density). Namely, dense graphs are too-well connected, and corresponding path existence problems become too simple. In such circumstances both algorithms finish in 2 or 3 iterations, and there is no real opportunity for their competition.

Table 4 presents the results for 50 shortest distance problems, all having the size $n = 100$.

graph density	# of pro -cessors	# of iterations Alg 1 : Alg 2	total complexity Alg 1 : Alg 2	avg cmplx per iter Alg 1 : Alg 2	score Alg 1 : Alg 2
10%	1	6.3 : 6.3	10861.3 : 4971.6	1715.59 : 792.59	0 : 10
	2	8.5 : 6.3	7375.6 : 3447.6	864.49 : 549.41	0 : 10
	4	9.1 : 6.3	3964.5 : 2610.2	434.09 : 415.23	0 : 10
	7	9.2 : 6.3	2431.8 : 2192.5	263.06 : 348.43	2 : 8
	13	9.2 : 6.3	1359.7 : 1905.4	147.11 : 302.61	10 : 0
	25	9.3 : 6.3	774.9 : 1687.5	83.03 : 268.13	10 : 0
	50	9.4 : 6.3	435.1 : 1509.6	46.05 : 239.60	10 : 0
	100	9.4 : 6.3	255.2 : 1349.3	27.03 : 213.81	10 : 0
30%	1	6.6 : 6.6	36252.4 : 15678.3	5477.71 : 2399.24	0 : 10
	2	8.3 : 6.6	22662.7 : 9280.1	2723.05 : 1419.43	0 : 10
	4	9.5 : 6.6	13061.7 : 5880.0	1370.17 : 898.62	0 : 10
	7	9.8 : 6.6	8094.2 : 4411.9	821.13 : 673.61	0 : 10
	13	10.1 : 6.6	4498.5 : 3327.2	443.02 : 507.42	0 : 10
	25	10.1 : 6.6	2334.8 : 2594.1	229.28 : 395.11	7 : 3
	50	10.1 : 6.6	1281.8 : 2069.8	126.10 : 314.74	10 : 0
	100	10.1 : 6.6	718.1 : 1586.5	70.88 : 240.64	10 : 0
50%	1	6.8 : 6.8	62743.0 : 25865.1	9212.84 : 3834.98	0 : 10
	2	9.0 : 6.8	41451.3 : 14537.6	4595.77 : 2154.67	0 : 10
	4	9.5 : 6.8	21862.3 : 8616.7	2293.68 : 1275.94	0 : 10
	7	10.2 : 6.8	14124.5 : 6091.7	1382.36 : 901.39	0 : 10
	13	10.1 : 6.8	7538.3 : 4251.4	744.03 : 628.91	0 : 10
	25	10.3 : 6.8	3990.5 : 3047.7	385.58 : 450.06	1 : 9
	50	10.4 : 6.8	2083.9 : 2161.6	199.43 : 318.67	7 : 3
	100	10.8 : 6.8	1183.5 : 1666.1	109.18 : 245.11	10 : 0
70%	1	7.1 : 7.1	92329.7 : 37156.4	12988.23 : 5241.43	0 : 10
	2	9.1 : 7.1	58751.6 : 20099.6	6444.97 : 2834.68	0 : 10
	4	10.2 : 7.1	33058.5 : 11386.5	3233.42 : 1605.63	0 : 10
	7	10.8 : 7.1	21073.6 : 7713.5	1946.11 : 1087.32	0 : 10
	13	10.9 : 7.1	11462.4 : 5086.9	1046.74 : 716.63	0 : 10
	25	11.2 : 7.1	5993.7 : 3294.3	533.09 : 463.95	0 : 10
	50	11.2 : 7.1	3090.4 : 2266.2	274.84 : 318.86	0 : 10
	100	11.2 : 7.1	1599.4 : 1752.0	142.33 : 246.25	9 : 1
90%	1	7.7 : 7.7	129456.9 : 47877.3	16769.96 : 6288.32	0 : 10
	2	10.0 : 7.7	83341.5 : 25246.8	8298.76 : 3314.06	0 : 10
	4	11.3 : 7.7	47214.4 : 13746.2	4157.17 : 1803.11	0 : 10
	7	11.6 : 7.7	29133.6 : 9006.4	2501.13 : 1180.99	0 : 10
	13	12.1 : 7.7	16339.8 : 5523.0	1344.99 : 723.08	0 : 10
	25	12.3 : 7.7	8394.0 : 3453.8	679.66 : 451.14	0 : 10
	50	12.3 : 7.7	4247.4 : 2418.1	343.60 : 315.02	0 : 10
	100	12.3 : 7.7	2162.2 : 1900.3	175.01 : 246.94	1 : 9

Tab. 4. Experimental results — randomly generated shortest distance problems.

The problems have again been arranged into 5 groups of 10, according to their graph density. As before, graph structures with desired densities, as well as destination nodes, were chosen randomly. In addition, arc lengths were generated as random numbers between 0 and 99. In this way, even a most complex path (consisting of many arcs) had a chance to be the shortest. Similarly as in Table 3, entries in Table 4 correspond to groups of problems, i.e. they represent averages and scores. In this way, Table 4 illustrates a typical behaviour of Algorithms 1 and 2, but now for the case of a shortest distance problem. As we can see, this behaviour is not quite the same as that for a path existence problem.

The problems in Table 4 cover a broad range of graph densities (from 10% to 90%). Dense graphs are more interesting for shortest distance than for path existence problems. Namely, a dense graph allows construction of many different paths connecting the same pair of nodes. To determine the shortest distance, it is not enough to find only one path between two given nodes (as in the case of path existence). Instead, all possible paths should be examined and their lengths should be compared. Consequently, shortest distance problems with denser graphs become more and more complex, even in terms of the number of iterations required.

Among other things, our experiments have also revealed an interesting property of Algorithm 1, which is usually called *superlinear speedup* (Helmbold and McDowell, 1990; Quinn, 1987). Thus, it can happen that, for some input data, the algorithm runs with m processors more than m times faster than with 1 processor. Concrete examples are visible in Table 3 (density 7%, number of processors 2 or 4). In fact, superlinear speedup is not as surprising as it looks at first sight. Namely, for each number of processors Algorithm 1 chooses a slightly different pattern of computation. It can happen that one of those patterns is "lucky enough" to find the solution after a relatively small total number of operations. Superlinear speedup occurs more frequently in path existence problems than in shortest distance problems. This fact can be explained by special properties of the Boolean path algebra. Still, the same phenomenon has been observed in few shortest distance problems, although due to averaging, Table 4 does not show it .

6. Conclusions

Both parallel algorithms considered in this paper solve a path problem by means of Gauss-Seidel iteration. However, each ot the two algorithms employs a different parallelization technique. Algorithm 1 uses a straightforward parallelization, which makes a slight compromise with the idea of the original (sequential) Gauss-Seidel method. Algorithm 2 uses a more subtle parallelization, which manages to reproduce genuinely the original method.

The available theoretical estimates of computational complexity turn out to be too vague to determine relative strengths and weaknesses of the two algorithms. Moreover, our examples show that these estimates cannot be made more precise, at least not in general. Therefore, an accurate evaluation and comparison of Algorithms 1 and 2 can be made only by experiments.

Our experiments show that Algorithm 1 is capable of employing many processors efficiently. However, with more processors Algorithm 1 can require more iterations. In some extreme cases, this overhead of iterations is so high that the total computational complexity becomes greater, although more processors are engaged. But usually, the degradation of performance is not too serious, and the overall speedup is still satisfactory.

Our experiments also demonstrate that Algorithm 2 is an outstandingly good sequential algorithm. Namely, with one processor Algorithm 2 requires considerably less operations to produce the same results as Algorithm 1. The reason for this is the already mentioned skipping capability, which relies on some special properties of path algebras. With more processors Algorithm 2 requires always the same number of iterations, but the total computational complexity drops down very slowly. So there is, in fact, no real potential to employ many processors. In some extreme cases, the algorithm is completely non-parallelizable. Usually however, there is some speedup, but never adequate to the number of processors.

Our experimental results clearly identify some situations when one of the algorithms is superior to the other. Namely, Algorithm 2 is always faster if only few processors are available.

With many processors (as many as graph nodes) Algorithm 1 usually becomes faster. So there usually exists a "turning point" (certain number of processors) where both algorithms perform similarly. However, the exact position of that point depends heavily on the problem involved.

The considered algorithms have been designed to solve single-destination path problems. But of course, they can be slightly modified to accommodate single-source problems. For all-pairs problems we recommend another family of parallel algorithms, which are based on Gaussian elimination rather than on iteration (Gayraud and Authie, 1992; Kung et al., 1987; Manger, 1992).

References

B. CARRÉ, *Graphs and Networks*. Oxford University Press, Oxford, 1979.

T. GAYRAUD AND G. AUTHIE, A parallel algorithm for the all pairs shortest path problem. In *Parallel Computing '91* (D. J. Evans, G. R. Joubert and H. Liddell, Eds.), pp. 107–114. Advances in Parallel Computing 4, North–Holland, Amsterdam, 1992.

D. P. HELMBOLD AND C. E. MCDOWELL, Modelling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, **1** (1990), 250–256.

S–Y. KUNG ET AL., Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Transactions on Computers*, **C–36** (1987), 603–614.

J. A. MCHUGH, *Algorithmic Graph Theory*, Prentice–Hall, Englewood Cliffs NJ, 1990.

R. MANGER, New examples of the path algebra and corresponding graph theoretic path problems. In *Proceedings of the 7th Seminar on Applied Mathematics, Osijek, Croatia, 13–15 Sep. 1989* (R. Scitovski, Ed.), pp. 119–128. University of Osijek, Croatia, 1990.

R. MANGER, A parallelization of the Jordan method for solving path problems. In *Proceedings of the 14th International Conference ITI, Pula, Croatia, 15–18 Sep. 1992* (V. Čerić and V. Dobrić, Eds.), pp. 491–496. University Computing Centre, Zagreb, Croatia, 1992.

R. MANGER, Parallel iterative algorithms for solving path problems. *Journal of Computing and Information Technology —CIT*, **1** (1993), 99–110.

W. H. PRESS ET AL., *Numerical Recipes in C — The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

M. J. QUINN, *Designing Efficient Algorithms for Parallel Computers*. McGraw–Hill, New York, 1987.

G. ROTE, Path problems in graphs. *Computing Supplement*, **7** (1990), 155–189.

R. P. TEWARSON, *Sparse Matrices*. Academic Press, New York, 1973.

Received: October, 1995
Accepted: October, 1996

Contact address:
Robert Manger
Department of Mathematics
University of Zagreb
Bijenička cesta 30
10 000 Zagreb
Croatia
Phone: +385 1 4555-720 / 119
Fax: +385 1 432-484
E-mail: manger@math.hr

ROBERT MANGER received the BSc. (1979), MSc. (1982), and PhD. (1990) degrees in mathematics, all from the University of Zagreb. For more than ten years he worked in industry, where he obtained practical experience in programming, computing, and designing information systems. Dr Manger is presently a lecturer in the Department of Mathematics at the University of Zagreb. His current research interests include parallel algorithms and neural networks. Dr Manger is a member of the Croatian Mathematical Society, Croatian Society for Operations Research and IEEE Computer Society.