A COMPUTATIONAL STUDY OF PARALLEL ALGORITHMS FOR THE ALL-PAIRS SHORTEST PATH PROBLEM

RIZOS SAKELLARIOU [1]
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, U.K.
E-mail: *rizos@cs.man.ac.uk*

Abstract

This paper presents experiences from the implementation of parallel algorithms for the all-pairs shortest path problem. A brief survey of the problem is given and efficient techniques for exploiting the memory hierarchy of parallel computers are described. Comparing the algorithms at a practical level, the importance of issues such as synchronisation and data locality is established.

1 Introduction

The problem of finding the shortest path in a graph is one of the most important problems in operations research. It arises in many applications and, as a result, has been the subject of numerous studies which provide solutions for both sequential [7] and parallel [3] computers. This paper deals with algorithms for solving the all-pairs shortest path problem (i.e. finding the shortest paths between all pairs of nodes in a graph) on parallel computers. While theoretical measures of the performance of these algorithms are rather well described, it is well known that for parallel implementations of any algorithms there are architecture features which may affect their performance. Moreover, the recent trend towards scalable Non-Uniform Memory Access (NUMA) parallel computers has resulted in dramatic changes in the relative costs of communication and computation, therefore making a re-examination of these algorithms at the implementation level desirable.

2 Parallel algorithms for the all-pairs shortest path problem

Let $G = (V, E)$ be a directed graph with n nodes and m arcs. Let l be a length function; we denote the length of the arc (i, j) by l_{ij}. The length of a path is the sum of the lengths of its arcs. Given a distinguished node s of G, the *single-source shortest path problem* (SSSP problem) is that of computing for each node $v \in V$

[1] The author is supported by the Greek State Scholarships Foundation.

```
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if length(i,k) + length(k,j) < length(i,j)
               then length(i,j) = length(i,k) + length(k,j)
        endfor
    endfor
endfor
```

Figure 1: The Floyd-Warshall algorithm

the length of a minimum-length path from s to v, while the *all-pairs shortest path problem* (APSP problem) is that of computing the length of a minimum-length path between all pairs of nodes.

The classical sequential algorithm for solving the APSP problem is that of Floyd-Warshall [5] which is illustrated in figure 1. An array *length* of dimension $n \times n$ holds the best known shortest distances between every pair of nodes. Initially, $length(i,j)$ is the length of the arc (i,j) if it exists, 0 if $i = j$, and ∞ otherwise. The expected running time is $O(n^3)$. The parallelism analysis of this algorithm is straightforward. For a given value of k there are no dependences, hence the loops for i and j can be executed in parallel, while the outermost loop carries dependences and must be executed sequentially. Thus, given an $O(n^2)$ number of processors, the running time becomes $O(n)$. At the implementation level, various techniques have been proposed for mapping the algorithm onto different models of parallel computers (for a list of references the reader is referred to [9, 16]). In the next section, a new technique suitable for NUMA parallel computers will be described.

Another approach to the APSP problem is to solve n SSSP problems. There are various sequential algorithms for solving the latter problem. According to one classification scheme [7], all these algorithms can be derived by a single prototype procedure, the main difference between them being in the particular data structure used to implement the set of candidate nodes. Such a prototype algorithm is illustrated in figure 2. In this algorithm, Q is the set of candidate nodes, r is the root node and d_i holds the length of the path from r to i. How nodes are inserted, selected and removed from Q affects the execution time of the algorithm.

Based on an idea attributed to D' Esopo and Pape [15], Pallottino suggested the use of two queues connected in series for the implementation of Q, resulting in an algorithm with complexity $O(n^2m)$ or $O(n^4)$ in the case of complete graphs [7]. Except in this latter case, the algorithm generally shows a good behaviour in computational experiments [7, 13].

Regardless of the SSSP algorithm used, n runs, each one for a different source, can be executed independently at the same time, thus resulting to a running time

1. d_r=0; $d_i = \infty, \forall i \neq r$.
2. If $Q = \emptyset$ then **STOP** else **select** and **remove** a node i from Q.
3. Scan the set of the arcs (i, j) which have i as the first node and adjust d_j.
4. **Insert** into Q any successors of node i for which d_j has been improved.
5. Go to step 2.

Figure 2: A prototype algorithm for solving the SSSP problem

for solving the APSP problem in parallel which is theoretically the same as that for solving the SSSP problem sequentially. Using a parallel algorithm for the SSSP problem, one may expect an even better performance. However, it is not our intention to examine parallel algorithms for the SSSP problem; the interested reader is referred to [10, 12]. Also, beyond the scope of this paper is the examination of algorithms which attempt to exploit topological features of the graph such as sparsity and planarity [6, 14], or make other simplifying assumptions [11].

Comparing the two approaches to solving the APSP problem, one may observe that the Floyd-Warshall algorithm offers better theoretical performance, but it needs a considerable amount $O(n)$ of synchronisation. Instead, algorithms based on SSSP algorithms need no synchronisation but at the expense of more expensive computations. On the other hand, both approaches have a high storage requirement, $O(n^2)$, a common characteristic of all direct, and therefore parallelisable, approaches to the APSP problem.

3 Exploiting the memory hierarchy

While recent advances in processor technology have increased considerably the computational power available, improvements in memory technology have not followed at a similar rate. As a result, there has been an increase in the time needed for memory access, creating an imbalance between the rate at which computations can be performed and the rate at which data can be transferred from memory (*memory latency*). In order to overcome this problem, machine designers have created complex memory hierarchies. Many existing parallel computers, like the KSR1, the Dash and the Cray T3D, fit into this category. They all have the architectural characteristic of physically distributed main memory, where remote main memory is more expensive to access than local memory, and, because of that, they are known as NUMA machines. To take advantage of the memory hierarchy, programs need to be restructured; this process is known as *optimising for data locality*. For parallel computers, the issue is optimising simultaneously for parallelism and data locality; the problem is that these two kinds of optimisations may seriously conflict.

In recent years much research effort has been devoted to this problem, focusing mainly on how to transform nested loops to achieve optimum parallelism and data locality [18]. We describe here the transformations which were used for the optimisation of the Floyd-Warshall algorithm:

• **Loop interchange** is useful for achieving better memory performance when referencing arrays. For example, consider the following FORTRAN loop where arrays are stored columnwise:

```
        DO 10 I=1,N
           DO 10 J=1,N
10            A(I,J)=B(I,J)+C(I,J)
```

References to successive elements of A,B,C are a long distance apart and each one of them may require a transfer from the main memory to the cache. Interchanging the I and J loops we guarantee that there will be a reference to the main memory only once every cache line.

• **Loop tiling** or **strip mining** [19] is a transformation that transforms a loop nest of m loops to a nest of $2m$ loops. Thus, it partitions the iteration space into blocks whose reuse can be captured by the cache. The typical example is matrix multiplication. While the entire computation involves $2n^3$ arithmetic operations, it produces and consumes only $3n^2$ data values, hence there is considerable data reuse. Since usually an entire matrix does not fit in a local cache this implies a significant data transfer. Tiling provides a method for executing the algorithm over blocks of data which fit into the cache. For example, one way to perform blocked matrix multiplication is the following:

```
for kk=1,n step s1
    for jj=1,n step s2
        for i=1,n
            for k=kk,min(kk+s1-1,n)
                for j=jj,min(jj+s2-1,n)
                    c(i,j)=c(i,j)+a(i,k)*b(k,j)
```

Theoretically, s_1 and s_2 should be chosen in such a way that $3s_1s_2$ data elements will fit in the local cache, while still leaving enough parallelism for the two outermost loops. However, it has been shown [4] that, in practice, the optimum tile size is difficult to predict.

Loop interchange and strip mining can be applied only to parallelisable nested loops, i.e. loops that do not carry any dependences [1, 18]. Hence, leaving intact the outermost loop, a strip mined version of the Floyd-Warshall algorithm is illustrated in figure 3. We assume that s_1, s_2 are integer divisors of n. It is worth observing that for $s_1 = 1, s_2 = 1$ a kji version of the original Floyd-Warshall algorithm results, while for $s_1 = 1, s_2 = n$ we get the kij as in figure 1.

```
for k = 1 to n do
    for j = 1 to n step s2 do in parallel
        for i = 1 to n step s1 do in parallel
            for jj = j to j + s2 − 1 do
                for ii = i to i + s1 − 1 do
                    if length(ii,k) + length(k,jj) < length(ii,jj)
                       then length(ii,jj) = length(ii,k) + length(k,jj)
                    endif
                endfor
            endfor
        endfor
    endfor
endfor
```

Figure 3: Strip-mined version of Floyd-Warshall algorithm

4 Experimental results

A series of experiments were performed on a virtual shared memory parallel computer, the KSR1. The KSR1 is a physically distributed memory computer, with each processor having a 32-Mbyte cache memory and a 0.5-Mbyte subcache [8]. The configuration used contains 64 processors.

Two different algorithms were implemented using KSR FORTRAN: *Floyd, Floyd tiled*; a third algorithm, *Pape*, based on Bertsekas' implementation of Pape's algorithm [2], was modified to solve the APSP problem, parallelised and adapted to the KSR FORTRAN. Two types of graphs were considered: *complete* and *sparse* graphs.

Complete graphs: Three different size complete graphs were generated, each one having 100, 300 and 500 nodes; for each size, two graphs were generated based on different arc lengths: in the first case we used the formula $l_{ij} = |i - j|$, hence the length of each arc was already the shortest path between the respective nodes, while in the second case we used the formula $l_{ij} = 2|i - j| - 1, i \neq j$ and therefore only arcs (i, j) such that $|i - j| = 1$ had already the value of the shortest path. The results are illustrated in table 1. As expected, the execution time of the Floyd-Warshall algorithm was not affected by the formula used to compute l_{ij}, therefore no distinction is made in the table. Regarding Pape's algorithm, for each value of n there are two lines, one for each formula. It should be noted also that all the results are the best time achieved over 3 runs.

The main observations from the table can be summarised as follows:

The tiled version of the Floyd-Warshall algorithm results in a better performance only for a small number of processors (< 16). It must be noted here that a different series of experiments were carried out in order to determine the values

Algo-rithm	n	Number of Processors							
		1	2	4	8	16	24	32	48
Floyd	100	0.70	0.50	0.49	0.69	0.69	0.80	0.87	0.93
	300	18.92	10.29	5.98	4.20	4.22	4.41	4.62	5.35
	500	87.91	46.66	25.69	15.64	14.16	12.30	13.58	13.85
Floyd (tiled)	100	0.59	0.49	0.52	0.68	0.82	0.95	1.02	1.12
	300	13.55	8.04	5.11	4.56	4.98	5.01	5.23	5.42
	500	62.48	36.61	21.08	13.87	13.72	13.65	14.50	15.11
Pape	100	2.06	1.28	0.70	0.46	0.47	0.51	0.60	0.66
	100	26.91	19.39	11.10	7.54	5.12	4.00	3.30	2.56
	300	51.91	29.48	16.56	8.84	4.48	4.16	3.93	2.96
	300	2556.10	2164.33	1244.05	766.63	669.29	425.02	382.01	254.12
	500	256.64	130.39	88.23	40.10	21.49	12.45	9.54	9.75
	500	25773.0	18845.3	11293.4	5564.3	3274.9	2861.9	2283.2	1834.0

Table 1: Complete graphs: running time of various algorithms in seconds

of s_1, s_2 that give optimum performance. As the number of processors increases, the optimum values become lower, gradually tending to $s_1 = 1, s_2 = 1$, i.e. the original version of the Floyd-Warshall algorithm. However, in the table, only results obtained for $s_1, s_2 > 1$ are presented. For further discussion, we refer to [17].

The Floyd-Warshall algorithm generally performs significantly better than Pape's algorithm. However, because of its high synchronisation requirements, the former shows a low speedup increase which, as the number of processor increases (and hence the grain size becomes smaller), becomes even lower and, at some point, the speedup starts to decrease. Instead, Pape's algorithm presents higher speedup increase and, in the case of graphs consisted of arcs whose length is the shortest path, outperforms the Floyd-Warshall algorithm when the number of processors becomes high (> 4 for $n = 100$, > 16 for $n = 300$, > 24 for $n = 500$). It is worth noting also the huge difference in Pape's algorithm running time for graphs of the same size but with different arc lengths, a characteristic rather neglected in the literature.

Sparse graphs: Three different size sparse graphs were generated, each one having 500, 1000 and 2000 nodes. Arcs were generated according to the following: For each pair of nodes $(i, j), i \neq j$, there exists an arc if $|i - j| \leq t$, t predefined constant. For a given t, a simple computation reveals that the number of arcs in a graph of size n is $m = 2tn - t(t+1)$. For $n = 500, n = 1000$, graphs with $t = 5$, $t = 20$, $t = 30$ were generated; for $n = 2000$, an additional graph with $t = 40$ was also generated. The results are illustrated in table 2. As expected, Floyd-Warshall algorithm was not affected by the number of arcs and therefore no distinction is made in the table.

For a small number of arcs, Pape's algorithm outperforms the Floyd-Warshall algorithm. As the number of arcs increases, Pape's algorithm becomes slower, a

Algo-rithm	n	t	Number of Processors 1	2	4	8	16	24	32	48
Floyd	500		86.5	45.9	24.2	14.1	11.1	12.1	9.6	10.2
	1000		698.6	370.0	193.0	103.8	65.0	54.6	48.4	45.1
	2000		12199.4	5037.2	2552.5	1126.1	461.4	344.4	274.9	252.0
Floyd (tiled)	500		62.4	36.5	21.0	13.9	13.6	13.8	14.6	15.3
	1000		490.8	281.1	148.9	87.6	71.0	68.4	65.2	64.7
	2000		11164.6	4858.5	1244.9	765.3	458.7	372.3	290.1	275.3
Pape	500	5	11.9	6.8	3.6	3.1	1.5	1.2	0.8	0.8
		20	93.8	62.6	34.4	19.7	10.4	7.2	5.3	4.1
		30	189.1	133.3	73.0	38.3	21.6	14.8	11.8	9.1
	1000	5	55.8	29.7	14.2	7.7	4.6	3.3	2.5	2.7
		20	390.4	265.7	143.8	75.4	41.2	31.4	21.7	21.6
		30	823.2	564.4	300.8	155.8	86.3	65.3	45.6	34.4
	2000	5	247.6	127.2	65.5	32.6	17.7	11.4	9.8	13.3
		20	1850.5	1172.1	566.8	301.5	161.7	112.2	83.9	58.6
		30	3653.5	2480.7	1288.2	632.3	341.4	229.1	180.4	135.3
		40	5878.1	3212.2	2151.9	1126.6	604.5	424.6	337.1	235.1

Table 2: Sparse graphs: running time of various algorithms in seconds

trend which is more obvious when the number of processors is relatively small. However, it can be observed that Pape's algorithm, due to its higher speedup increase, outperforms Floyd-Warshall when using 48 processors for $n = 500, t = 30$ and $n = 1000, t = 30$ despite the fact that it is significantly slower when using less processors.

Similar observations with the ones made for complete graphs can be made also for the tiled version of the Floyd-Warshall algorithm. It results in a better performance than the original algorithm for a relatively small number of processors (≤ 8 for $n = 500$, ≤ 16 for $n = 1000$, ≤ 24 for $n = 2000$).

As an empirical rule, one can say that Pape's algorithm has a better performance for very sparse graphs ($m/(n^2 - n) < 0.05$); when this ratio increases, the tiled Floyd-Warshall algorithm becomes better for small numbers of processors, with the original Floyd-Warshall being better for medium numbers of processors, and Pape's algorithm better for large numbers of processors. Further increase of this ratio enlarges the range that the tiled Floyd-Warshall algorithm is optimum while making Pape's algorithm advantageous for even higher numbers of processors.

5 Conclusions

Usually, the aftermath of such studies is to provide answers to questions of the type "which is the fastest or most efficient algorithm?". Based on experimental evidence, our study has already provided answers. However, the significant impact of the machine architecture should be noted; characteristics such as the synchronisation cost and the way that the memory hierarchy is built are the ones which determine the performance of various algorithms, as the number of processors increases, for graphs of similar structure. Given that these characteristics cannot be quantified, one has to rely on experimental evidence, something which makes the choice of an optimum algorithm a rather laborious task.

References

[1] U. Banerjee, *Loop Transformations for Restructuring Compilers*, Kluwer Academic Publishers, 1993.

[2] D. P. Bertsekas, *Linear Network Optimization: Algorithms and Codes*, MIT Press, 1991.

[3] D. P. Bertsekas, J. N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice-Hall, 1989.

[4] S. Carr, Memory-Hierarchy Management, Rice University, Department of Computer Science Technical Report, **TR92-190**, 1992.

[5] R. W. Floyd, "Algorithm 97: Shortest Path", *Communications ACM*, **5** (1962), p. 345.

[6] G. N. Frederickson, "Planar Graph Decomposition and All Pairs Shortest Paths", *Journal of ACM*, **38-1** (Jan. 1991), pp. 162–204.

[7] G. Gallo, S. Pallottino, "Shortest Path Algorithms", *Annals of Operation Research*, **13** (1988), pp. 3–79.

[8] *KSR Parallel Programming*, Kendall Square Research, 1991.

[9] V. Kumar, V. Singh, "Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem", *Journal of Parallel and Distributed Computing*, **13** (1991), pp. 124–138.

[10] P. Mateti, N. Deo, "Parallel Algorithms for the Single Source Shortest Path Problem", *Computing*, **29** (1982), pp. 31–49.

[11] A. Moffat, T. Takaoka, "An All Pairs Shortest Path Algorithm with expected time $O(n^2 \log n)$", *SIAM Journal of Computing*, **16-6** (Dec. 1987), pp. 1023–1031.

[12] T. Mohr, C. Pasche, "A Parallel Shortest Path Algorithm", *Computing*, **40** (1988), pp. 281–292.

[13] J. F. Mondou, T. G. Crainic, S. Nguyen, "Shortest-Path Algorithms: A Computational Study with the C programming language", *Computers and Operations Research*, **18** (1991), pp. 767–786.

[14] G. E. Pantziou, P. G. Spirakis, C. D. Zaroliagis, "Efficient Parallel Algorithms for Shortest Paths in Planar Graphs", *LNCS* **447** (1990), pp. 288–300.

[15] U. Pape, "Algorithm 562: Shortest Path Lengths", *ACM Transactions on Mathematical Software*, **6** (1980), p. 450–455.

[16] M. J. Quinn, N. Deo, "Parallel Graph Algorithms", *ACM Computing Surveys*, **16-3** (Sep. 1984), pp. 319–348.

[17] R. Sakellariou, Ph.D. thesis, in preparation, University of Manchester.

[18] M. E. Wolf, M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE Transactions on Parallel and Distributed Systems*, **2-4** (Oct. 1991), pp. 452–471.

[19] M. Wolfe, "Iteration space tiling for memory hierarchies", *Proceedings of the Third SIAM Conference on Parallel Processing*, Los Angeles, 1987.