

# R-Kleene: A High-Performance Divide-and-Conquer Algorithm for the All-Pair Shortest Path for Densely Connected Networks

Paolo D'Alberto and Alexandru Nicolau

Department of Computer Science  
University of California at Irvine \*

**Abstract.** We propose a novel divide-and-conquer algorithm for the solution of all-pair shortest-path problem for directed and dense graphs with no-negative cycles. We propose *R-Kleene* a compact and in-place recursive algorithm inspired by Kleene's algorithm. R-Kleene delivers better performance than previous algorithms for randomly-generated graphs represented by highly-dense adjacency matrices, in which the matrix components can have any integer value. We show that R-Kleene, unchanged and without any machine tuning, yields consistently between 1/7 and 1/2 of the peak performance running on five very different uniprocessor systems.

## 1 Introduction

The **all-pair shortest-paths problem** (APSP) is a well studied and basic problem in graph theory but it is also a crucial and real problem in large networks such as sensor networks, switch networks or complex targeting systems.

Consider the scenario where many thousands of nodes are located across a large area and every node has a processor with little memory space and computational power. In this scenario, the computation of APSP is neither feasible nor practical by a single node, nonetheless it is a key feature for the efficient data routing and broadcasting. Despite the node-processor computational/memory limitations, a node in the network is able to determine the locations and distances of its neighbors rather easily. Such a local information can be coded, sent

---

\* Authors: [paolo,nicolau]@ics.uci.edu. This work has been supported in part by NSF Contract Number ACI 0204028

on the network and collected by an observer node such as a satellite, a global router or a computer cluster. Then, the observer node may construct the adjacency matrix, compute the solution and send the result back on the network where each node will store the necessary local information.

Any network is naturally represented by a **directed graph** and we formalize the APSP as follows. Given a graph  $G = (V, E)$  where  $V$  is a set of nodes and  $E$  is a set of directed edges, we label every node in the graph by an integer  $\iota \in [0, n - 1]$  where  $n = |V|$  ( $n = |V|$  is the **cardinality** of the set  $V$ ), and an edge in  $E$  is defined by a unique ordered pair of integers  $(i, j)$  with  $i, j \in [0, n - 1]$ . In fact, we assume that there is at most one directed edge connecting two nodes and therefore, the graph has no more than  $|E| = m \leq n^2$  edges. To represent  $G$ , we use an adjacency matrix,  $\mathbf{A} \in \mathbb{Z}^{n \times n}$ . That is, an entry in row  $i$  and column  $j$  in matrix  $\mathbf{A}$ ,  $a_{i,j} \in \mathbb{Z}$ , stands for the cost to reach node  $j$  from node  $i$  through the edge  $(i, j) \in E$ . Also, we assume that  $a_{i,i} = 0$  for every  $i \in [0, n - 1]$  –i.e., there is no cost to stay in one node– and if there is no direct edge from node  $i$  to node  $j$ , then  $a_{i,j} = \infty$ . An **elementary path** [1] of length  $k$  from node  $i$  to  $j$  (in a graph  $G$ ) is a sequence of  $k$  edges connecting  $i$  to  $j$  with no nodes repeated. In fact, we denote an elementary path as the set of edges  $\mathbf{P}_k(\mathbf{i}, \mathbf{j})$ . The **cost of an elementary path**  $P_k(i, j)$  is denoted as  $\mathbf{C}[\mathbf{P}_k(\mathbf{i}, \mathbf{j})]$  and it is equal to  $\sum_{\ell=0}^{k-1} a_{\iota_\ell, \iota_{\ell+1}}$ , where  $(\iota_\ell, \iota_{\ell+1}) \in P_k(i, j)$ . Thus, the solution to the APSP problem is the **matrix closure**  $\mathbf{A}^*$  such as for all  $i, j \in [0, n - 1]$  the matrix element  $a_{i,j}^*$  is  $\min_{k \in [0, n-1]} C[P_k(i, j)]$ .

In this paper, we propose a recursive **divide-and-conquer** (D&C) algorithm inspired by Kleene’s algorithm [2], which is optimal in the number of addition–comparison operations and memory accesses,  $\Theta(n^3)$ . Indeed, fast **matrix-multiply** (MM) algorithms [3,4] that could speed up performance as proposed by Zwick [5] are not applicable; in fact, we assume that the adjacency-matrix entries have no constraint and, thus, they can be any positive or negative integer –as we shall explain in Section 2. Though, our approach is based on MM, however it is independent of the algorithm used to code MM. So we may apply R-Kleene in combination with architecture-specific highly-tuned MMs –see Section 3– but this is beyond the scope of

this paper. We have two major contributions as we summarize them in the following.

- First, we formulate our algorithm as a recursive MM where the result is computed in-place for dense adjacency matrices. As such, we are able to replicate the classic properties of MM, such as space complexity, I/O complexity and register utilization. In fact, Kleene’s algorithm and Floyd-Warshall algorithm [2,6,7] impose a specific computation order that exploits little parallelism and register reuse. In contrast, R-Kleene leads to an in-place implementation that yields an efficient register utilization and exposes a larger number of independent operations, and, in turn, better performance. (This idea can be taken a step further and we may apply the same optimization and fine tuning used by software libraries such as ATLAS [8]; but this is beyond the scope of this work.) In fact, we exploit an inherent property of the application that exhibits a loose order of the computation. This is property of the application and it is not a general approach as the one proposed in [9] for the study of DAG consistency.
- Second, we present a quantitative measure of R-Kleene performance using row-major and Z-Morton data layout [10], and we present an upper bound to the performance achievable by a recursive algorithm such as R-Kleene. We achieve this by collecting experimental results for our algorithm and other 3 representative algorithms on five different systems and for any adjacency matrices of sizes ranging from 200 to 5,000 ( $7 \leq \log_2 n \leq 12$ ). In fact, we show that R-Kleene achieves good, predictable, scalable and portable performance.

The remainder of the paper is organized as follows. In Section 2, we introduce the related work. In Section 3, we introduce our D&C algorithm. In Section 4, we discuss our experimental results. We draw our conclusions and future work in Section 5.

## 2 Related Work

The literature available for APSP and its solutions is copious and, for the sake of explanation, we may distinguish four main categories:

APSP algorithms for dense graphs, for sparse graphs, for static networks and dynamic networks (changing in time).

For a general overview of the static approaches, Zwick [11] presents a complete survey about the algorithms for the computation of the exact distances in graphs and also about the *lingering open problems* in the topic. We may notice that the most efficient algorithms presented in the survey are based on Dijkstra’s algorithm [12]. In practice, the complexity of an APSP algorithm is briefly summarized by the number of **comparison–addition** operations –*compadd* for short– and Dijkstra’s algorithms perform  $O(mn) \leq O(n^3)$  *compadds*.

Our original contribution is a static solution of the APSP problem for dense graphs and Floyd-Warshall algorithm [6,7] was our starting point. This algorithm has the same complexity as Dijkstra’s (i.e.,  $O(n^3)$ ) but it is often preferred for its *practical* performance for dense adjacency matrices [1]. In practice, Floyd-Warshall algorithm is an in-place algorithm that constructs the shortest paths during the computation using a dynamic-programming approach. To exploit data locality, the algorithm can be reorganized as Kleene’s algorithm [2], which can be seen as the blocked Floyd-Warshall algorithm, where the classic matrix multiply is used as basic routine. Notice that if the adjacency matrix has values that belong to a finite and small set, the domain where APSP is defined, a semi-ring, can be extended to a ring. Thus, we may use fast MMs (e.g., Strassen’s or Coppersmith’s [3,4]) as proposed by Zwick [5] and other authors. However, in the scenario assumed throughout this work, we cannot use such fast MMs.

Our work is similar to the work by Park et al. and Penner et al. [13,14], because we investigate the performance for APSP algorithms for dense adjacency matrices using different data layouts. However, our algorithm is 100% faster (this is a major result) because it utilizes more efficiently data in registers, reducing the number of memory accesses and, thus, improving performance.

In parallel and independently of our work, Sibeyn [15] presents recursive and blocked algorithms for APSP. In summary, he presents an alternative proof of correctness for the algorithms presented in this paper; however, he collected a preliminary set of experimental results that is in direct contrast to ours because he did not exploit fully the algorithm properties. In practice, while we explain every

algorithm-engineering step using an algebraic proof, which is based only on the properties of the closed semi-ring, he finds the same high-level algorithm using a classic presentation (theorem/proof) where the proofs are colloquial. More importantly, he does not exploit the implications of the theoretical results: in fact, he produces a limited and preliminary set of experimental results where his codes do not take advantage of the features of the recursive algorithms; thus, he concludes that recursive/blocked algorithms do not perform as well as the basic algorithm. In contrast, we show that our recursive algorithm outperforms the basic Floyd-Warshall and all previous algorithms.

In this work, we propose an algorithm that is oblivious of the graph structure and, therefore, it performs  $n^2$  *compadd* per node (but it is still optimal for dense and generic graphs). This approach seems less efficient than the algorithms presented by Cherkassky et al. [16]. In fact, Cherkassky et al. show that algorithms based on Dijkstra’s algorithm achieve an average of 1 *compadd* per node when applied to a representative set of sparse graphs. This is a large work difference obviously. However, we shall show in Section 3, that our algorithm can be parallelized naturally, and used efficiently in multiprocessor systems, offering an appealing performance edge.

To conclude this short review, we may notice that all of the previous approaches are *static* solutions, because they assume that the network is fixed and unchangeable. However, in the scenario where nodes can be introduced/removed dynamically, and the APSP solution must be updated correspondingly, then the solution must be dynamic as well; for example of dynamic approaches see Demetrescu et al. [17,18].

### 3 A Recursive D&C Algorithm, R-Kleene

Recursive algorithms are extremely appealing for the design of matrix computation because of their natural cache locality exploitation and for their intuitive formulation [19,20,21]. In this section, we present a recursive D&C algorithm derived from Kleene’s algorithm Figure 1.(a). Notice that Kleene’s algorithm was originally designed to solve the **transitive closure** (TC) of an adjacency matrix. That is, finding whether or not there is a path connecting two nodes in

directed graph. However, Kleene’s algorithm is also a **standard** algorithm/solution for the APSP. In fact, in a closed semi-ring TC and APSP are the same problem and Kleene’s algorithm is a solution (when the scalar operators  $*$  and  $+$  are specified as in the following paragraph) and it determines every edge of the (shortest) path directly [2].

A brief description of Kleene’s algorithm follows. We divide the basic problem into two sub-problems, and we solve each subproblem directly using the Floyd-Warshall algorithm, see Figure 1.(a). Then, we perform several MMs to combine the results of the subproblems using a temporary matrix. Formally, matrix multiplication  $\mathbf{E}+ = \mathbf{F}\mathbf{G}$  (with  $\mathbf{E}, \mathbf{F}, \mathbf{G} \in \mathbb{Z}^{n \times n}$ ) is simply  $e_{i,j}+ = \sum_{k=0}^{n-1} f_{i,k} * g_{k,j}$ ; where the scalar addition of two numbers is actually the minimum of the two numbers –i.e.,  $a+b = \min(a, b)$ – and the scalar multiplication of two numbers is the (regular arithmetic) addition –i.e.,  $a * b = a + b$ .

<pre> Kleene(J) { /*        A B   */ /* J =   C D   */  1: A = FLOYD_WARSHALL(A); 2: A += A*A*A; 3: B += A*A*B; 4: C += C*A*A; 5: D += C*A*B;  6: D = FLOYD_WARSHALL(D); 7: A += B*D*C; 8: B += B*D*D; 9: C += D*D*C; 10: D += D*D*D; } </pre>	<pre> R-Kleene(J) { /*        A B   */ /* J =   C D   */  1: A = R-Kleene(A); 2: B += A*B; 3: C += C*A; 4: D += C*B;  5: D = R-Kleene(D); 6: B += B*D; 7: C += D*C; 8: A += B*C; } </pre>	<pre> J += J*J { /*        A B   */ /* J =   C D   */  A += A*A; B += A*B; C += C*A; D += C*B;  D += D*D; B += B*D; C += D*C; A += B*C; } </pre>
(a)	(b)	(c)

**Fig. 1.** (a) Kleene, (b) R-Kleene and (c) (Self) Matrix Multiply

In this case, the MM is defined in a closed semi-ring and using the properties within, we reorganize the algorithm in Figure 1.(a) and obtain the R-Kleene algorithm in Figure 1.(b). Thus, R-Kleene is a solution for APSP in a closed semi-ring.

In the following, we explain in seven steps how to achieve the algorithm in Figure 1.(b):

1. We start by noticing that the computations on line 2 and 10 in Figure 1.(a), (i.e.,  $\mathbf{A}+ = \mathbf{A} * \mathbf{A} * \mathbf{A}$  and  $\mathbf{D}+ = \mathbf{D} * \mathbf{D} * \mathbf{D}$ ) can be removed because they do not have any effects on matrix  $\mathbf{A}$  and  $\mathbf{D}$  respectively, because each is a matrix closure.

A formal proof follows. First,  $\mathbf{AA}$  is equal to  $\mathbf{A}$ ; in fact, consider  $\mathbf{F} = \mathbf{AA}$ ,  $f_{i,j} = \sum_{k=0}^{n-1} a_{i,k} * a_{k,j}$ ; because  $\mathbf{A}$  is matrix closure,  $f_{i,j} = a_{i,j} * a_{j,j} + \sum_{k \neq j} a_{i,k} * a_{k,j} = a_{i,j} + \sum_{k \neq j} a_{i,k} * a_{k,j} = a_{i,j} + \sum_{k \neq j} a_{i,j} = a_{i,j}$ , therefore  $\mathbf{F} = \mathbf{A}$ . Moreover, because  $+$  is idempotent (i.e.,  $a + a = a$ ) we have that  $\mathbf{A} + \mathbf{A}$  is equal to  $\mathbf{A}$ .

2. Notice that the property that  $\mathbf{AA} = \mathbf{A}$ —when  $\mathbf{A}$  is matrix closure—is applied elsewhere such as on line 3 in Figure 1.(a), which becomes the operation on line 2 in Figure 1.(b).
3. Moreover, consider the computation on line 5 in Figure 1.(a)  $\mathbf{D}+ = \mathbf{CAB}$ , this is equivalent to  $\mathbf{D}+ = \mathbf{CB}$ —Figure 1.(b), on line 4.

A formal proof follows. First, we show that  $\mathbf{CA} \equiv \mathbf{C} + \mathbf{CA}$ ; in fact, consider  $\mathbf{F} = \mathbf{CA}$ , then  $f_{i,j} = \sum_{k=0}^{n-1} c_{i,k} * a_{i,j} = c_{i,j} * a_{j,j} + \sum_{k \neq j} c_{i,k} * a_{i,j}$ ; because  $a_{j,j} = 0$ , we have  $f_{i,j} = c_{i,j} + \sum_{k \neq j} c_{i,k} * a_{i,j} = c_{i,j} + \sum_{k=0}^{n-1} c_{i,k} * a_{i,j}$ , so  $\mathbf{CA} \equiv \mathbf{C} + \mathbf{CA}$ . We conclude by noticing that  $\mathbf{D}+ = (\mathbf{C} + \mathbf{CA})\mathbf{A}(\mathbf{B} + \mathbf{AB})$  is equivalent to  $\mathbf{D}+ = (\mathbf{CA} + \mathbf{CAA})(\mathbf{AB} + \mathbf{AAB}) = (\mathbf{CA})(\mathbf{AB}) = (\mathbf{C} + \mathbf{CA})(\mathbf{B} + \mathbf{AB})$ .

4. Similarly, we obtain the simplified computation  $\mathbf{D}+ = \mathbf{CB}$  (i.e., line 4 in Figure 1.(b)).
5. Moreover, following a similar reasoning, we may postpone the computation of line 7 in Figure 1.(a), as last on line 8 in Figure 1.(b),
6. The last step is to apply the idea recursively on  $\mathbf{A}$  and  $\mathbf{D}$ .

If we look at the algorithm in Figure 1.(b), ultimately, this is *similar* to the recursive algorithm for MM, Figure 1.(c). Though, Kleene's algorithm and the algorithm proposed by Park et al. [13]—which is basically the algorithm in Figure 1.(c)—impose a strict order to the function calls; however, the MM algorithms, in general, and the MMs in our algorithm do not require to apply the same order of the function calls—e.g., in Figure 1.(c). In fact, we shall explain shortly that our algorithm is bound loosely to the function call order.

It is evident that the computations on line 2 and 3 in Figure 1.(b) can be executed in any order and in parallel. We now show that the MMs involved in the computation have no restrictions and they may exploit a different order than the one specified in Figure 1.(c). This is important because certain low-level-optimizations for the solution of MM need to rearrange the computation order –which, in general, is quite different from the algorithm in Figure 1.(c)– so as to exploit better locality and, thus, performance. Indeed, we show in Remark 1 that we may choose any order and, thus, the order that exploits data reuse in registers.

*Remark 1.* In a closed semi-ring the matrix multiplications  $\mathbf{B}+ = \mathbf{A}\mathbf{B}$  or  $\mathbf{B}+ = \mathbf{B}\mathbf{A}$  can be done in place and in any evaluation order if  $\mathbf{A}$  is a matrix closure.

A formal proof follows, however we explain the first case  $\mathbf{B}+ = \mathbf{A}\mathbf{B}$  only. Consider two elements in  $\mathbf{B}$  during the in-place computation. Without loss of generality, consider the entries  $b_{0,j}$  and  $b_{1,j}$ , which belong to the same column of  $\mathbf{B}$  and necessarily are needed for the computation of each other. Assume we compute first  $b_{0,j} = a_{0,m} * b_{m,j}$ , and then, we compute  $b_{1,j} = a_{1,n} * b_{n,j}$ . If we assume that  $b_{1,j}$  affects the previous shortest path, then we must recompute  $b_{0,j}$ , which should be  $\dot{b}_{0,j} = a_{0,1} * b_{1,j}$ . So we unfold the expression and we obtain  $\dot{b}_{0,j} = a_{0,1} * a_{1,n} * b_{n,j}$ ; because  $\mathbf{A}$  is matrix closure, we have  $a_{0,1} * a_{1,n} = a_{0,n}$ , thus, in the first evaluation we had  $b_{0,j} = a_{0,m} * b_{m,j} \leq a_{0,n} * b_{n,1} = \dot{b}_{0,j}$ . The following computation of  $b_{1,j}$  does not affect the computation of  $b_{0,j}$ , therefore we may perform the computation in any order and in-place.

Notice that this Remark assures that, as long as all terms are computed and written into the destination matrix without *write races*, the matrix multiplication can be successfully parallelized as it would be when source and destination operands are all non-overlapping matrices.

Intuitively, we can see that our algorithm inherits the properties of MM, and thus, it is **cache oblivious** achieving optimal data cache utilization at every cache level (asymptotically). We briefly repeat here the major results about the locality property of Kleene’s algorithms as previously investigated in [2]. In fact, Kleene’s algorithm is the first cache-aware algorithm with access complexity  $O(\frac{n^3}{\sqrt{S}})$  (also

I/O complexity), where  $S$  is the cache size in number of elements. Matrix Multiply has the same lower and upper bounds [22].

To prove that R-Kleene is asymptotically optimal, we determine the upper bound to the access complexity as follows. Suppose that  $S = s^2$ , matrix  $J$  has size  $n \times n$  and  $(n \bmod s) = 0$ . (The case for general square matrices is similar.) Recursively, R-Kleene divides the problem  $n \times n$  in smaller problems and it may compute the solution directly when the problem has size no larger than  $s \times s$ . Thus, the algorithm solves  $8^{\log n/s} = (\frac{n}{s})^3$  problems directly, each requiring up to  $2s^2$  memory accesses (i.e.,  $s^2$  reads from memory to cache and  $s^2$  writes from cache to memory), so we achieve a total of  $O(\frac{n^3}{\sqrt{S}})$  memory accesses, which is optimal asymptotically. Notice that because the algorithm accesses tiles of the adjacency matrix, a cache-aware layout can store such tiles continuously in memory improving the cache behavior of the algorithm. Such a layout reduces self/inter interference, therefore, cache conflicts further (see also [23,24,25]).

Previously [26,8], we developed techniques to improve the **leaf computation** of MM (where the recursion stops). In fact, we may exploit data reuse in registers and, therefore, we may achieve a sensible reduction of loads/stores for matrix multiplication. Indeed, we may reduce the memory accesses, for matrix multiply of matrices of size  $m \times m$ , from  $3m^3 + o(m^2)$  to  $\frac{2}{r}m^3 + o(m^2)$ , with  $1 \leq r^2 < R$  where  $R$  is the number of registers available. The fewer memory loads/stores in MM are, the higher the overall performance is, because MM is basic kernel.

## 4 Experimental Results

In this section, we discuss briefly the characteristics of the algorithms implemented, and how we measured overall performance. We conclude this section presenting the experimental results of the algorithms for every system separately. In practice, we compare R-Kleene versus other three algorithms on five systems using (very) different processors for dense adjacency matrices with random entries uniformly generated in an interval.

**R-Kleene** is our recursive algorithm in Figure 1.(b). The basic MM operation is a recursive algorithm that exploits data locality and aggressive data reuse in registers at the leaf computations so to

achieve near optimal performance. The algorithm assumes that the adjacency matrix is stored in a row-major format.

**Floyd-Warshall (FW)** is the classic algorithm based on a single loop nest. The algorithm assumes that the adjacency matrix is stored in a row-major format. In general, this algorithm is efficient only for small problem sizes and its performance degrades quickly as the problem size increases.

**Simple Recursive (Z-SR)** is the recursive algorithm in Figure 1.(c). This is the algorithm presented by Park et al. [13]– which these authors proposed for power of two matrices only – and, in fact, the performance presented in this work coincides with the performance previously published. The algorithm assumes that the adjacency matrix is stored in a generalized Z-Morton format [27]; that is, logical sub-matrices of the adjacency matrix are stored continuously in memory in a recursive fashion.

**ZR-Kleene** is the R-Kleene algorithm, however it assumes that the adjacency matrix is stored in a generalized Z-Morton format [27]. This algorithm should have a performance advantage for very large problems and memory hierarchies with high latency and low associativity.

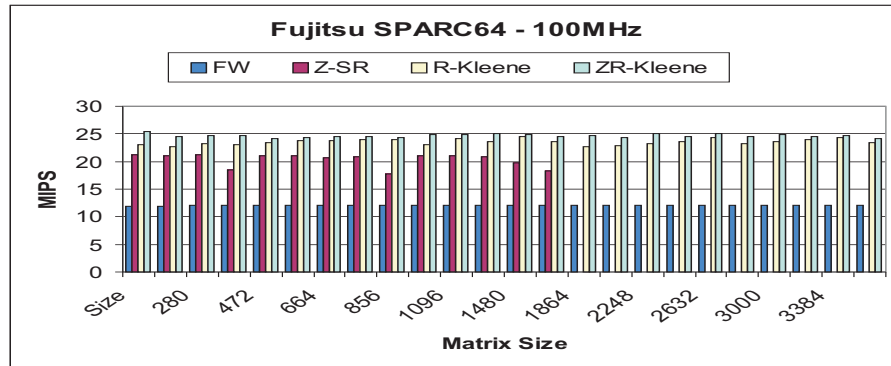
Our goal is to show the performance improvements obtained by register management only (R-Kleene), by memory layout optimization only (Z-SR) and by the synergy of both (ZR-Kleene). We measure performance as **millions of integer instructions per second** (MIPS) determined as  $n^3 / (\text{execution of the algorithm in seconds})$ . This format is consistent with the one used for classical linear algebra applications (e.g., MM). However, there is a major difference between APSP and MM. In MM, highly pipelined functional units execute the basic operation (*madd*) using a separate register file and the performance measure is MFLOPS. In contrast, in APSP algorithms, the basic operation is based on a conditional branch, that is, a **comparison-addition** (*compadd*). This constraint affects the final performance as a function of the values in the adjacency matrix, even if only by a constant factor.<sup>1</sup> In practice, some processors have available branch prediction units - e.g., R12K - allowing instruction speculation on either branch of a conditional jump. However, because

---

<sup>1</sup> But not as much as for algorithms on sparse adjacency matrices

the unpredictable nature of the adjacency matrix, branch predictor may be ineffective. Nevertheless, we assume that the reference peak performance of any algorithm is the number of cycles per second.

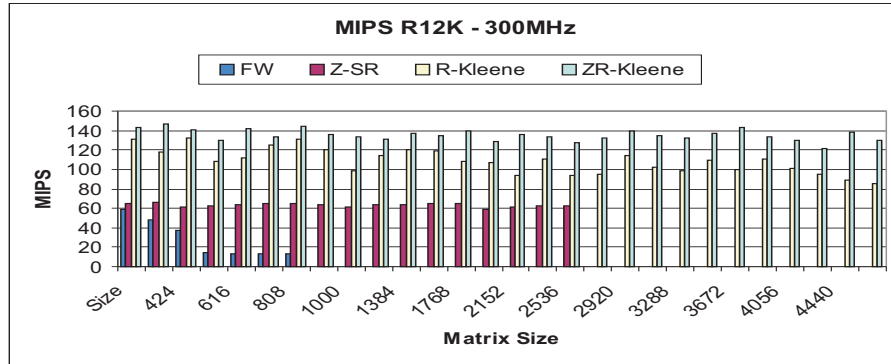
In the following, we discuss our results. In Figure 2, we present



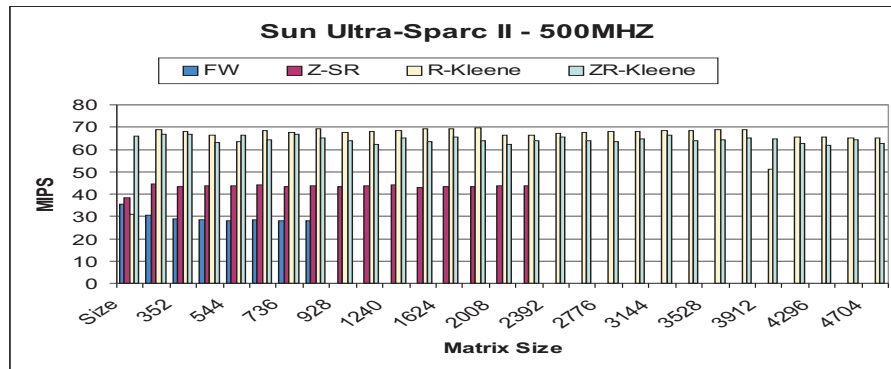
**Fig. 2.** Fujitsu HAL 100 - SPARC 100MHz: best performance 4 cycles per *compadd*.

the results for the Fujitsu HAL 100 system based on a SPARC64 processor (one level of split caches, 128KB 4-way data and instruction caches). The Z-SR algorithm performs 1 *compadd* every 5 cycles while R-Kleene and ZR-Kleene perform 1 *compadd* every 4 cycles; that is a 20% performance improvement. Notice that the memory layout has no significant effects on the overall performance of the algorithms. (We used the native compiler, *hcc*, to generate the executables.)

In Figure 3, we present the results for the SGI O2 system based on a MIPS R12K 300MHz processor (with two-level cache: first level, 32KB 2-way distinct data and instruction cache; second level, 512K 2-way unified cache), where we can achieve the best relative performance. The Z-SR algorithm performs 1 *compadd* every 5 cycles, while R-Kleene and ZR-Kleene perform 1 *compadd* every 2 and 3 cycles. R-Kleene and ZR-Kleene achieve a two-fold speed up with respect to Z-SR. Notice that the memory layout has significant effects on the overall performance of the algorithms. (We used the native compiler, SGI compiler, to generate the executables.)

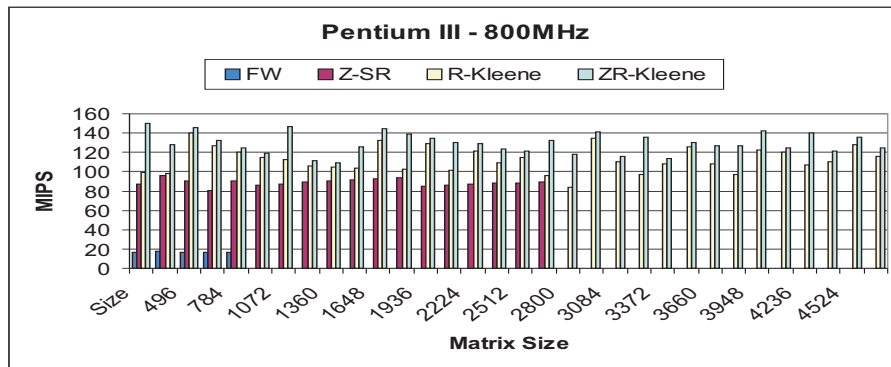


**Fig. 3.** SGI O2 - MIPS R12K 300MHz: best performance 2 cycles per *compadd*.



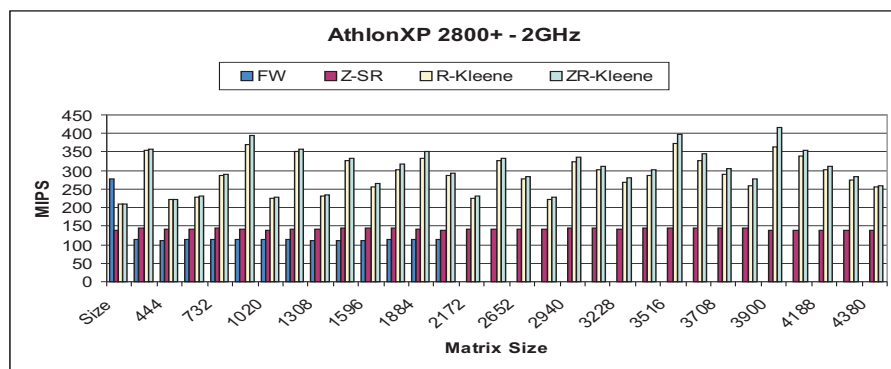
**Fig. 4.** Sun Blade 100 - UltraSparc II 500MHz: best performance 7 cycles per *compadd*.

In Figure 4, we present the results for the Sun microsystems Sun Blade system based on a UltraSparc IIe 500MHz processor (two-level cache: first level, 16KB direct mapped distinct data and instruction cache; second level, 512KB 2-way unified cache). The Z-SR algorithm performs 1 *compadd* every 11 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 7 cycles, which is a 30% performance improvement. Notice that the memory layout has no significant effects on the overall performance of the algorithms. (We used two compilers to generate the executables, *gcc/3.0.4* and *cc-forte-6*, we present the best performance.)



**Fig. 5.** FOSA 3240 - Pentium III 800MHz: best performance 6 cycles per *compadd*.

In Figure 5, we present the performance results for the FOSA 3240 system based on a Pentium III 800MHz (two-level cache: first level, 16KB direct mapped distinct data and instruction cache; second level, 256KB direct mapped unified cache). The Z-SR algorithm performs 1 *compadd* every 9 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 6 cycles, which is a 35% performance improvement. Notice that the memory layout has significant effects on the overall performance of the algorithms. (We used *gcc/3.1* compiler to generate the executables.)



**Fig. 6.** Asus - Athlon-XP 2800+ 2GHz: best performance 5 cycles per *compadd*.

In Figure 6, we present the performance results for ASUS system based on an Athlon-XP 2800 (two-level cache: first level, 64KB 2-way distinct data and instruction cache; second level, 256KB 16-way unified cache). The Z-SR algorithm performs 1 *compadd* every 13 cycles. In contrast, R-Kleene and ZR-Kleene perform 1 *compadd* every 6 cycles. Notice that the memory layout has no significant effects on the overall performance of the algorithms. (We used *gcc/3.3* compiler to generate the executables.)

## 5 Conclusion and Future Work

We presented R-Kleene, a novel recursive D&C algorithm for the solution of APSP; we also presented a quantitative measure for its performance across five systems. We conclude that an efficient register allocation is an important feature of any APSP algorithms; we also notice that non-standard layouts are beneficial for very low associative caches but otherwise row-major layouts are quite adequate.

We have started the design and implementation of a preliminary parallel R-Kleene algorithm for a two-processor system achieving speed-ups ranging from 1.41 to 1.74. In the future, we intend to import ATLAS (also multithreaded) routines to improve performance further and exploit parallelism at processor and thread level.

## References

1. Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press (1990)
2. Ullman, J., Yannakakis, M.: The input/output complexity of transitive closure. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data. Volume 19. (1990)
3. V.Strassen: Gaussian elimination is not optimal. *Numerische Mathematik* **14** (1969) 354–356
4. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Proceedings of the 19-th annual ACM conference on Theory of computing. (1987) 1–6
5. Zwick, U.: All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM* **49** (2002) 289–317
6. Floyd, R.: Algorithm 97: Shortest path. *Communications of the ACM* **5** (1962)
7. Warshall, S.: A theorem on boolean matrices. *Journal of the ACM* **9** (1962)
8. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society (1998) 1–27

9. Blumofe, R., Frigo, M., Joerg, C., Leiserson, C., Randall, K.: Dag-consistent distributed shared memory. In: IPPS '96: Proceedings of the 10th International Parallel Processing Symposium, IEEE Computer Society (1996) 132–141
10. Chatterjee, S., Lebeck, A., Patnala, P., Thottethodi, M.: Recursive array layout and fast parallel matrix multiplication. In: Proc. 11-th ACM SIGPLAN. (1999)
11. Zwick, U.: Exact and approximate distances in graphs - a survey. In: Proceedings of the 9th Annual European Symposium on Algorithms, Springer-Verlag (2001) 33–48
12. Dijkstra, E.: A note on two problems in connection with graphs. *Numerische Math* (1959) 269–271
13. Park, J., Penner, M., Prasanna, V.: Optimizing graph algorithms for improved cache performance. In: Proceedings of the International Parallel and Distributed Processing Symposium. (2002)
14. Penner, M., Prasanna, V.: Cache-friendly implementations of transitive closure. In: Proceedings of International Conference on Parallel Architectures and Compilation Techniques. (2001)
15. Sibeyn, S.: External matrix multiplication and all-pairs shortest path. *Information Processing Letters* **91** (2004) 99–106
16. Cherkassky, B.V., Goldberg, A.V., Radzik, T.: Shortest paths algorithms: theory and experimental evaluation. In: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (1994) 516–525
17. Demetrescu, C., Italiano, G.F.: A new approach to dynamic all pairs shortest paths. In: Proceedings of the thirty-fifth ACM symposium on Theory of computing, ACM Press (2003) 159–166
18. Demetrescu, C., Emiliozzi, S., Italiano, G.F.: Experimental analysis of dynamic all pairs shortest path algorithms. In: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics (2004) 369–378
19. Frens, J., Wise, D.: Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In: Proc. 1997 ACM Symp. on Principles and Practice of Parallel Programming. Volume 32. (1997) 206–216
20. Gustavson, F.: Recursion leads to automatic variable blocking for dense linear algebra algorithms. *Journal of Research and Development* **41** (1997)
21. Gustavson, F., Henriksson, A., Jonsson, I., Ling, P., Kagstrom, B.: Recursive blocked data formats and blas's for dense linear algebra algorithms. In: ed.: PARA'98 Proceedings. Lecture Notes in Computing Science, No. 1541. (1998) 195–206
22. Hong, J., Kung, T.: I/o complexity, the red-blue pebble game. In: Proceedings of the 13th Ann. ACM Symposium on Theory of Computing. (1981) 326–333
23. Jalby, E.G.W., Teman, O.: To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In: Proceedings of Supercomputing. (1993) 410–419
24. Rothberg, M.L.E., Wolfe, M.: The cache performance and optimizations of blocked algorithms. In: Proceedings of the fourth international conference on architectural support for programming languages and operating system. (1991) 63–74
25. Dayde, M., Duff, I.: A blocked implementation of level 3 blas for risc processors. Technical Report TR\_PA\_96\_062, CERFACS (1996) [http://www.cerfacs.fr/algor/reports/TR\\_PA\\_96\\_06.ps.gz](http://www.cerfacs.fr/algor/reports/TR_PA_96_06.ps.gz).

26. Bilardi, G., D'Alberto, P., Nicolau, A.: Fractal matrix multiplication: a case study on portability of cache performance. In: Workshop on Algorithm Engineering 2001, Aarhus, Denmark (2001)
27. Chatterjee, S., Jain, V., Lebeck, A., Mundhra, S.: Nonlinear array layouts for hierarchical memory systems. In: Proc. of ACM international Conference on Supercomputing, Rhodes, Greece (1999)