CS 301 : High Performance Computing

Floyd-Warshall Parallelization

Assigned By
Prof. Bhaskar Chaudhury

Students
Purvil Mehta (201701073)
Bhargey Mehta (201701074)

Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar

May 2, 2020

Contents

1 Brief Introduction

The Floyd–Warshall algorithm (also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm) is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). This algorithm will find the shortest distance possible to any of the two nodes. We will later prove that maximum achievable speedup is p. Our aim is to optimize the cache usage in such a way that we achieve a speedup as close as possible to this upper bound.

1.1 Input

The input contains a graph of N nodes and their edge weights in the form of an adjacency matrix of size $N \times N$. This algorithm can handle negative edge weights but not negative cycles since if there does exist a negative cycle then we can circle to this cycle, go around it an infinite number of times and have a distance of $-\infty$ between all nodes. There doesn't remain a need of the algorithm in the first place.

More formally `adj_mat[i][j]` denotes the weight of the edge that originates in node i and terminates in node j. In case of an un-directed graph, `adj_mat[i][j] = adj_mat[j][i]`.

1.2 Output

The output is a matrix `dis` of size $N \times N$. This means that the optimum distance for going from node i to j is the value `dis[i][j]`. Again in case of an undirected graph, we will get `dis[i][j] = dis[j][i]`.

2 Algorithm

The algorithm works by considering a set of nodes which act as intermediate nodes between a given source and sink. This means that for a given source and sink, we have the optimum distance when the intermediate nodes are allowed to be from this set.

With each iteration of the algorithm, we expand this set. According to the definition of the kth iteration, we should obtain the optimum distance when the intermediate nodes allowed to be from the set $1,2,..,k$. The node k is now allowed to be an intermediate node so some paths, let's say from i to j would be updated but one condition that holds true is that $d(i,k)$ and $d(k,j)$ won't change in this iteration or that they are already optimum. This holds true because we are disallowing negative cycles.

- We initialize the solution matrix same as the input graph matrix as a first step. That is to say that there are no intermediate nodes. All the edges themselves are the optimum distance.
- Each iteration is the inclusion of the node k. For every pair (i,j) of the source and destination nodes respectively, there are two possible cases.
    1. k is not an intermediate node in shortest path from i to j. We keep the value of **dist[i][j]** as it is.
    2. k is an intermediate node in shortest path from i to j. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j].**

3 Scope of Parallelism

We will prove that `dis[i][k]` will not be updated during the kth iteration. For that statement to be executed,

$$\texttt{dis[i][k]} > \texttt{dis[i][k]} + \texttt{dis[k][k]}$$

$$\texttt{dis[k][k]} < 0$$

This can't be true since this implies a negative cycle and we have forbidden that. We can similarly prove that `dis[k][j]` will also not be updated.

Hence `dis[i][j]` always depends on two quantities which are not going to be updated. Hence there will be no data race, only parallel reads. Hence the one singular iteration of N^2 updates can safely be parallelised.

3.1 Serial Fraction of the code

1. The statement `dis[i][j]` = min(`dis[i][j]`, `dis[i][k]` + `dis[k][j]`) is a combination of 1 addition + 1 comparison + 1 assignment = 3 operations.
2. For each iteration there are a total of $5N^2$ operations of which 5 are to be done serially namely 1) selection of i, 1) selection of j, 3) addition of `dis[i][k]` and `dis[k][j]`, 4) comparison with `dis[i][j]` and 5) assignment to `dis[i][k]`.
3. For each iteration serial fraction $s_i = \frac{5}{5N^2} = \frac{1}{N^2}$ and let total operations $T_i = 5N^2$. In addition to these iterations, there are $2N$ comparison and addition operations for k which are completely serial.
4. Total serial code $S = 2N + N \times s_i T_i = 7N$.
5. total code $T = 2N + N \times (5N^2) = 5N^3 + 2N$.

Using the above facts, serial fraction of code

$$s = \frac{7N}{5N^3 + 2N} = \frac{7}{5N^2 + 2}$$

3.2 Upper Bound using Amdahl's Law

According to Amdahl's Law, maximum possible speedup is $S_{\max} = \frac{1}{s + \frac{1-s}{p}}$. But as $N \to \infty$, $s \to 0$, so $S_{\max} \to p$. Hence ideally we should get a speedup of p but the memory access pattern of the algorithm poses a challenge in achieving this speedup.

4 Parallelisation Strategies

```
for(k=0; k<N; k++)
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            dis[i][j] = min(dis[i][j], dis[i][k]+dis[k][j]);
```

Listing 1: Core Algorithm

4.1 Approach 1) Naive Strategy

Since we have proved that all the N^2 updates are independent, we can safely parallelise the entire N^2 updates. Serially for first iteration of i, we will suffer $\frac{N}{16}$ misses and none thereafter since the entire kth row will be in cache. The column experiences 1 miss for each row. This gives a total of $N \times (\frac{N}{16} + N) = \frac{17N^2}{16}$ misses.

Naively we will parallelise the second loop. The reasons for this decision are:

- We cannot parallelise the outer loop because of dependencies which can't be removed.
- If we parallelise the inner loop then there are N times greater (N^2 vs. N) synchronisation steps which results in an increase in an idle time.

4.1.1 Time Complexity

In approach 1, we observed that i and j loop are parallelised among P processor. Thus, time complexity will be $\theta\left(N\left(\frac{N^2}{P} + 1\right)\right)$.

4.1.2 Compute to Memory Access (CMA)

- Total Computation : $2N^3$ (For the addition between dis[i][k] and dis[k][j])
- Total Access for dis[i][j] = N^3 (i, j and k loop for N times)
- Total Memory Access : N^3
- CMA = $\frac{2N^3}{N^3} = 2$.

4.2 Approach 2) Using Auxiliary Arrays

We observed that we require kth column and kth row in each iteration. This means we will have cache misses only for y and x matrix which will be $\left(N + \frac{N}{16}\right)$ for a single value of k to give a total of $\left(\frac{17N^2}{16}\right)$ misses which is same as above. However this code performs better both serially and parallely inspite of doing extra $2N^2$ assignment operations.

```
for(k=0; k<N; k++)
    for(i=0; i<N; i++)
        x[i], y[i] = dis[k][i], dis[i][k];
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            dis[i][j] = min(dis[i][j], x[j]+y[i]);
```

Listing 2: Using Auxiliary Arrays

4.2.1 Time Complexity

There are N iterations of outer loop. For each such iteration, we do N assignment operations which are done parallely so they take $O(\frac{N}{p})$ time. In addition to these we do N^2 update operations which again are completely parallelisable so they take $O(\frac{N^2}{p})$ time. Hence total time complexity is $O(N*(\frac{N^2}{p}+\frac{N}{p}+1))$.

4.2.2 Compute to Memory Access (CMA)

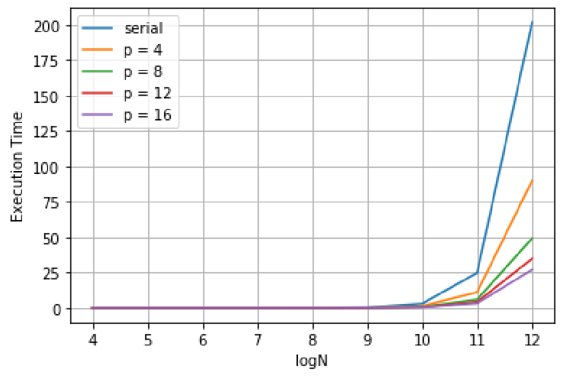
- Total Computation : $2N^3$ (For the addition between x[j] and y[i]).
- Total Access for dis[i][j] = N^3 (i, j and k loop for N times)
- Total Access for xp[j] and yp[i] = N^2
- Total Memory Access : $N^3 + 2N^2$
- CMA = $\frac{2N^3}{N^3+2N^2} \approx 2$

5 Results

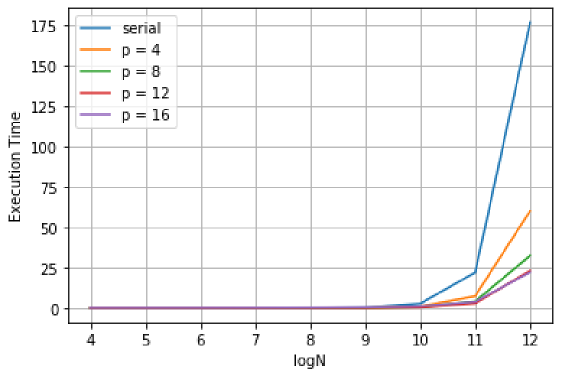
5.1 Execution Time

We observed that approach 1 takes too much amount of time in executing the serial code. This is in part due to cache misses but the major factor is the recalculation of indices that occur when we access `dis[i][k]` as internally this works by calculating $i*N+j$ as discussed when compute costs were considered in section 6.4.

The execution time for approach 2 is less due to better utilisation of cache and less computations as discussed in section 6.2 and 6.4.



(a) Approach 1 - Execution Time



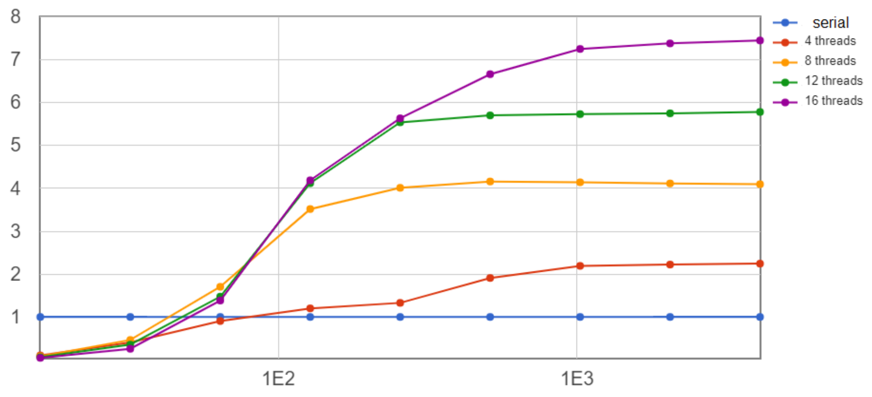
(b) Approach 2 - Execution Time

This trend is followed by all the threads. The execution time of approach 1 is always considerably greater than the same for execution time.

Problem Size	Serial Time for approach 1	Serial Time for approach 2	% Change
256	0.057	0.045	21.05
512	0.413	0.349	15.50
1024	3.137	2.761	12.00
2048	24.855	21.973	11.59
4096	201.371	176.583	12.30

5.2 Speedup with Naive Parallelisation

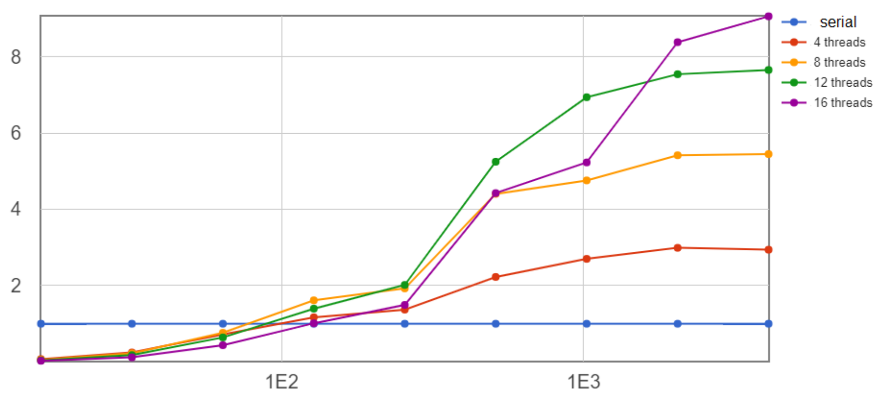
The speedup for approach 1 for 4 thread was observed as 2.224. The reason for this is that when cores work parallely, they encounter more than $\frac{N}{16}$ misses due to possible overlap between the data fetches. Hence along with the slow execution time, we get a slower speedup as well. This remains the trend for all parallelisations i.e. using 4 threads to 8 threads. This speedup can be increased if these cache misses are avoided.



5.3 Speedup using Auxiliary Arrays

In the second approach we are using two matrix which can be easily fit in L1 cache and **thus causes lesser cache miss**. That results in a significant speedup when we consider the serial code of the same approach.

This means that serially this approach performs better than the naive serial approach and since it also suffers less cache misses, the speedup is more than the naive parallelisation as discussed previously in sections 6.2 and 6.4.



Thread 4/ N = 4096	Serial Execution Time	Parallel Execution Time	Speedup
Approach 1	201.625	89.885	2.24
Approach 2	176.45	60.011	2.95

5.4 Efficiency

According to Amdahl's law, the theoretical speedup can not be increased more than some extent. As the serial fraction of the code is very small, according to the law, the tighter bound for speedup of our approaches is the total number of the thread used. However, parallelisation overhead and cache misses restrict the speedup to the certain amount.Note that, for smaller input size, the cache misses is not going to happen because all arrays can be fit in cache itself. But read-write overhead in approach 2 will be higher putting efficiency down for smaller inputs. Whereas for large input, the cache misses and accessing time of div[i][j] in the approach 1 causes the delay in the execution time compare to approach 2. Thus from the definition of the efficiency

$$e = \frac{\text{Speedup}}{\text{Thread}}$$

the efficiency for approach 2 is higher than the efficiency of approach 1.

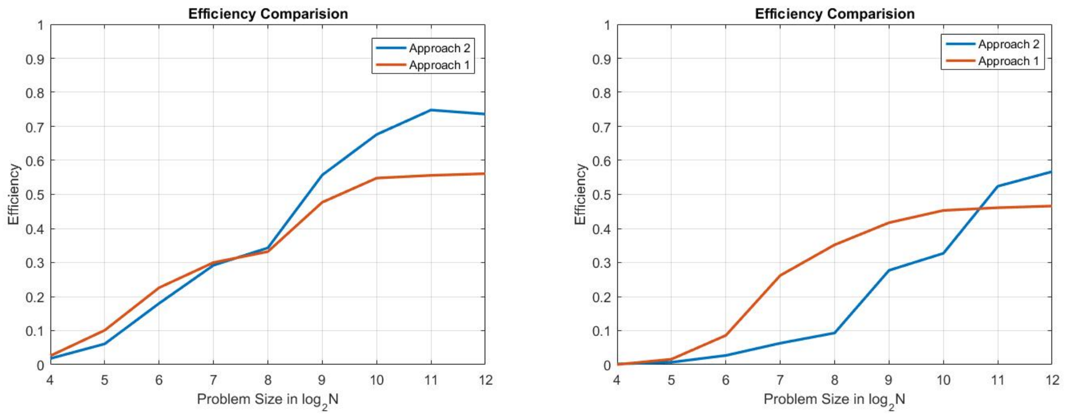


Figure 2: Efficiency Comparison a) 4 Threads b) 16 Threads

6 Analysis and Justifications

6.1 Memory Access Pattern

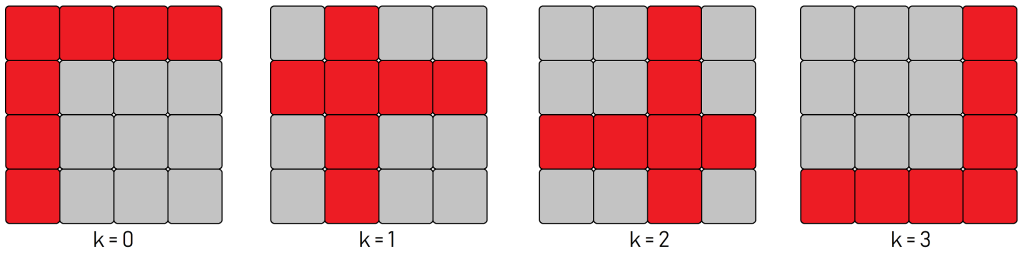


Figure 3: Memory Access Pattern for the Algorithm

The code access the memory in the following way. An update on element $dis[i][j]$ in the k$^{\text{th}}$ iteration requires the i^{th} value in the k$^{\text{th}}$ column and the j^{th} value in the k$^{\text{th}}$ row. The two auxiliary arrays i.e. the k$^{\text{th}}$ row and column move in the following manner during the algorithm. Our main focus during parallelisation would lie on optimising the access of these arrays since the algorithm requires synchronisation between each iteration and the updates are completely parallelisable for a given iteration and can't be avoided. The cache line is of size 64 bytes. Hence each miss fetches the next 16 integers as well in the memory. 2 arrays of size 2^{12} can be fit into L1 cache. (2^{12}elements × 4(sizeof(int)) × 2arrays = 32K). Hence for a problem size = 4096, both the column and the row can fit in the cache.

6.2 Case 1: $2N \leq C$ and $C << N^2$

6.2.1 Approach 1

1. We require the entire k^th row which would be available in L1 cache after one iteration of i incurring $\frac{N}{16}$ misses.
   (a) Serially: $\frac{N}{16}$ misses
   (b) Parallely: In addition to the above misses, we will also experience misses when some processors require the same element of the k^th row but both find it absent in the cache. In the worst case if the processors are working on the same index of i everytime, then we can see that this will happen $p\frac{N}{16}$ times.
2. We also require the entire k^th column.
   (a) Serially: We will get N misses, 1 for each row (since after the first miss, it will be there in L1 cache)
   (b) Parallely: Because one row is assigned to only one core, we will get no more than N misses for the column.
3. We will also incur misses for accessing `dis[i][j]`. We can see that such misses are bound to occur no matter what we do.

Thus we can conclude that the naive approach requires the same amount of cache to store the required data but also experience more misses as compared to the serial code when implemented parallely.

6.2.2 Approach 2

1. We again need to copy the entire k^th row again to the auxiliary arrays.
   (a) Serially: We experience the same number of cache misses as in the naive approach i.e. $\frac{N}{16}$
   (b) Parallely: This time the row itself is divided amongst the processors. So each miss is unique and won't be repeated by any processor like in the naive approach.
2. We need the entire k^th column in this approach too. The behaviour is same as the previous approach.
3. We will again experience misses for accessing `dis[i][j]`.

After analysing both the parallel approaches, we can see that using auxiliary arrays, we are experiencing less misses as compared to the naive approach. Thus in the worst case the naive approach experiences $p\frac{N}{16}$ more misses and in the best case, both the approaches experience the same number of misses. However both are pathological situations and are very unlikely to happen during runtime so approach 2 performs better.

6.3 Case 2: $C << 2N$

In this case will experience cache misses since the k^th column/row or auxiliary arrays wouldn't fit in the cache.

6.3.1 Approach 1

1. We will now get a miss for every row when accessing `dis[k][j]` as opposed to case 1 when we used to get a miss only once for a value of k.
   (a) Serially: For one iteration of i, we get $\frac{N}{16}$ misses. So for the entire algorithm, we will get $\frac{N^3}{16}$ misses.
   (b) Parallely: It is possible for cores to access the same memory location resulting in $p\frac{N^3}{16}$ misses in the worst case.
2. While accessing the k^th column we have:
   (a) Serially: While accessing `dis[i][k]`, we get 1 miss per row. So there would be N^2 misses.
   (b) Parallely: Again all the misses are unique like in case 1 i.e. we get N^2 misses.
3. Accessing `dis[i][j]` results in a total of $\frac{N^3}{16}$ misses.

6.3.2 Approach 2

1. While loading the auxiliary arrays, it is clear that we will experience $\frac{N}{16}+N$ misses for each k or $\frac{N^2}{16}+N^2$ for the entire algorithm.
2. The analysis done for accessing the k$^{\text{th}}$ row is duplicated but this time for the auxiliary array
3. We will now experience lower cache misses when accessing `dis[i][k]` because now it is replaced by a contiguous auxiliary array.
   (a) Serially: Reduced to $\frac{N^2}{16}$
   (b) Parallely: Reduced to $\frac{N^2}{16}$
4. Accessing `dis[i][j]` results in a total of $\frac{N^3}{16}$ misses.

Concluding we have,

- Naive Parallel: $p\frac{N^3}{16}+N^2+\frac{N^3}{16}$ misses
- Approach 2: $\frac{N^2}{16}+N^2+p\frac{N^3}{16}+\frac{N^2}{16}+\frac{N^3}{16}$ misses

Analysing both the parallel approaches we see that, approach 2 has an extra $\frac{N^2}{8}$ misses. However approach 2 has a compute advantage over the naive approach which is explained in the next section.

6.4 Compute Costs

The memory allocated for the `dis` array is linear so when we access `dis[i][j]`, the offset calculation takes about 10 cycles for multiplication since we need to calculate offset $= i \times N + j$. There are 2 such operations (`dis[i][k]` and `dis[k][j]`) in each update which mean $2N^3$ such operations.

There are 2 possible ways to remove this computation.

1. Store the values contiguously as we did in approach 2. Here we see that the code remains parallelisable but we incur an extra $\frac{N^2}{8}$ cache misses.
2. Use a contiguous array but calculate the offset at each iteration by adding k to the previous offset with initial offset being equal to i. This adds an extra computation of 1 cycle but that is still better than 1 multiplication instance. However note that this introduces a dependency in the algorithm and hence can't be parallelised anymore.

In addition to the above observations, let us see when compute costs advantage overcome the memory access disadvantage. Assuming that a L3 cache miss means a penalty of 1000 cycles and a multiplication operation takes 5 cycles, we find that the turning point of N is,

$$(2N^3)\times 5 > (\frac{N^2}{8})\times 1000$$

$$N > \lceil\frac{100}{8}\rceil \implies N > 13$$

Thus we conclude that the time saved in compute costs of offsets have overcome the cache miss penalties far earlier and even in those cases approach 2 remains the better one since the cache contains the auxiliary array before the core algorithm begins.

6.5 Scheduling Policies

In the schedule(static, chunk-size) scheduling policy, OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order. In the schedule(dynamic, chunk-size) scheduling type, OpenMP divides the iterations into chunks of size chunk-size. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available. Here two terms are important to understand.

- **Scheduling Overhead:** Since dynamic scheduling divide the work load among thread according to their availability, every time a thread is idle it has to be assigned to a chunk by the OS. This scheduling will cause a delay in the run time called as "Scheduling Overhead".
- **Load Balance:** Load balance means "Equal work to all threads". Dynamic scheduling automatically assigns the work to that thread which is available causing no delay in the run time. Whereas static scheduling algorithm assign the work depending upon the chunk size.

Since we observed that there are no condition checks inside the parallel region and all thread have to do a fix amount of work on each iteration. Thus the work load can be evenly distributed among threads at compile time and we can get rid of dynamic scheduling overhead.

Thread 4/ N = 4096	Serial Execution Time	Parallel Execution Time	Speedup
Dynamic	189.479	74.609	2.54
Static	176.45	60.011	2.95

7 Conclusion

After all analysis we can say that using auxiliary arrays results in an improvement in the serial algorithm by reducing computation incurred in offset calculation while maintaining the same if not less cache miss rate. Removing these computation costs in the naive approach requires introduction of dependencies which renders the algorithm unparallelisable. Using auxiliary arrays has it's own compute costs and overheads but doing so allows us to parallelise the problem and still get a considerable improvement over the naive approach.

8 References

- Basics of the serial Floyd-Warshall Algorithm:
  - https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- Contiguous Memory Allocation Syntax:
  - https://stackoverflow.com/questions/13105056/allocate-contiguous-memory
- Scheduling Policies:
  - http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
  - https://people.sc.fsu.edu/~jburkardt/c_src/schedule_openmp/schedule_openmp.html

9 Appendix

Compute Node Specs

Architecture	x86_64
Cores	16
Threads per Core	1
Sockets	2
CPUs per Socket	8
Model	Intel Xeon E5-2640 v3 @ 2.60GHz
L1 Cache	32K
L2 Cache	256K
L3 Cache	20480K

Naive Parallelisation Static Scheduling

N	serial	$p=4$	$p=8$	$p=12$	$p=16$
64	0.001	0.001	0.001	0.001	0.001
128	0.008	0.007	0.002	0.002	0.002
256	0.056	0.043	0.014	0.010	0.010
512	0.411	0.216	0.099	0.072	0.062
1024	3.129	1.429	0.755	0.546	0.042
2048	24.824	11.184	6.033	4.317	3.363
4096	201.625	89.885	49.228	34.867	27.059

Using Auxiliary Arrays and Static Scheduling

N	serial	$p=4$	$p=8$	$p=12$	$p=16$
64	0.001	0.001	0.001	0.001	0.039
128	0.006	0.005	0.004	0.004	0.086
256	0.044	0.033	0.023	0.022	0.186
512	0.348	0.157	0.079	0.066	0.394
1024	2.754	1.019	0.579	0.397	1.122
2048	21.937	7.336	4.047	2.907	3.855
4096	176.45	60.011	32.376	23.039	22.194

Using Auxiliary Arrays and Dynamic Scheduling

N	serial	$p=4$	$p=8$	$p=12$	$p=16$
64	0.001	0.002	0.002	0.002	0.002
128	0.007	0.008	0.006	0.007	0.007
256	0.051	0.039	0.025	0.025	0.027
512	0.372	0.186	0.119	0.101	0.093
1024	2.951	1.243	0.761	0.601	0.525
2048	23.565	9.339	5.393	4.017	3.297
4096	189.479	74.604	41.358	30.096	23.888