

# CS 5220 – Project 3: Shortest Paths

Group 10: Ze Jin (zj58), Ben Shulman (bgs53), Michael Whittaker (mjw297)

November 18, 2015

## 1 Introduction

In this assignment we develop a set of optimized and parallelized programs that compute the shortest path between all pairs of vertices in a directed graph. In Section 2, we profile the release implementation which uses a repeated squares  $O(N^3 \log N)$  algorithm. In Section 3, we describe the repeated squares algorithm in depth, and we present two additional algorithms that we implemented: a blocked repeated squares algorithm and the Floyd-Warshall algorithm. In Section 4, we describe the three parallelization mechanisms we used: OpenMP, MPI, and a hybrid of the two. Finally in Section 5 and Section 6, we evaluate the performance of our implementations and conclude.

## 2 Profiling

The release implementation of all-pairs shortest paths is a naive implementation of a repeated squares algorithm that uses OpenMP for parallelization. We profiled the release code using VTune Amplifier and the vectorization reports produced by `icc`.

### 2.1 VTune Amplifier

The profiling results produced by VTune Amplifier are shown in Figure 1 and Figure 2. The profiles show that the release implementation spends most of its time inside the `square` function. The majority of the time in `square` is spent accessing the `l` array. This is in contrast to many applications that spend most of their time on computation. This is unsurprising as the only computation done in the release implementation is a single addition and comparison.

These results suggested that we optimize our implementations to have good memory access patterns to take full advantage of the cache as much as possible. This optimization is discussed in detail in Section 3.

Function	Module	CPU Time
-----	-----	-----
<code>square</code>	<code>omp.x</code>	41.807s
<code>__kmp_barrier</code>	<code>libiomp5.so</code>	13.993s
<code>__kmpc_reduce_nowait</code>	<code>libiomp5.so</code>	6.397s
<code>__kmp_fork_barrier</code>	<code>libiomp5.so</code>	2.962s
<code>__intel_sse3_rep_memcpy</code>	<code>omp.x</code>	0.040s
<code>fletcher16</code>	<code>omp.x</code>	0.030s
<code>gen_graph</code>	<code>omp.x</code>	0.020s
<code>__kmp_join_call</code>	<code>libiomp5.so</code>	0.010s
<code>genrand</code>	<code>omp.x</code>	0.010s

Figure 1: A snippet of the vectorization report produced by VTune Amplifier that shows the longest running functions.

Source Line	Source	CPU Time
41	int square(int n, // Number of nodes	
42	int* restrict l, // Partial distance at step s	
43	int* restrict lnew) // Partial distance at step s+1	
44	{	
45	int done = 1;	
46	#pragma omp parallel for shared(l, lnew) reduction(&& : done)	
47	for (int j = 0; j < n; ++j) {	
48	for (int i = 0; i < n; ++i) {	0.020s
49	int lij = lnew[j*n+i];	0.030s
50	for (int k = 0; k < n; ++k) {	8.072s
51	int lik = l[k*n+i];	25.646s
52	int lkj = l[j*n+k];	
53	if (lik + lkj < lij) {	3.644s
54	lij = lik+lkj;	
55	done = 0;	4.395s
56	}	
57	}	
58	lnew[j*n+i] = lij;	
59	}	
60	}	
61	return done;	
62	}	

Figure 2: A snippet of the vectorization report produced by VTune Amplifier that shows the longest running sections of the `square` function.

## 2.2 Vectorization Reports

A snippet of an `icc` vectorization report is shown in Figure 3. The report shows that most loops in the release implementation are vectorized. However, the vectorized loops have unaligned memory accesses because `l` and `lnew` are not allocated into aligned memory. This suggested that we optimized our code to allocate memory at aligned addresses.

```

LOOP BEGIN at path.c(108,5) inlined into path.c(258,5)
remark #15388: vectorization support: reference l has aligned access [ path.c(110,13) ]
remark #15388: vectorization support: reference l has aligned access [ path.c(110,13) ]
remark #15300: LOOP WAS VECTORIZED
remark #15448: unmasked aligned unit stride loads: 2
remark #15449: unmasked aligned unit stride stores: 1
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 14
remark #15477: vector loop cost: 1.250
remark #15478: estimated potential speedup: 9.830
remark #15479: lightweight vector operations: 10
remark #15488: --- end vector loop cost summary ---
LOOP END

```

Figure 3: `icc` vectorization report.

## 3 Algorithms

In this section, we describe the three algorithms we implemented to compute all-pairs shortest paths.

### 3.1 Repeated Squares

The release implementation uses a simple repeated squares (RS) dynamic programming algorithm. Let  $l_{ij}^s$  represent the length of the shortest path from vertex  $i$  to vertex  $j$  of at most length  $2^s$ . RS relies on the following recurrence:

$$l_{ij}^{s+1} = \min_k (l_{ik}^s + l_{kj}^s)$$

The base case  $l_{ij}^0$  is the weight of the edge from vertex  $i$  to vertex  $j$  or  $\infty$  if no such edge exists. This recurrence is nearly identical to the formula used to compute the square of a matrix  $A$ :

$$a_{ij}^2 = \sum_k a_{ik} a_{kj}$$

The algorithm initializes a distance matrix  $L^0$  and iteratively computes  $L^{s+1}$  by squaring  $L^s$ . The algorithm terminates once squaring  $L$  reaches a fixpoint; that is, once  $L^2 = L$ . Each squaring requires  $O(N^3)$  operations where  $N$  is the number of vertices in the graph and the side-length of  $L$ . A shortest path can be of at most length  $N$ , so the algorithm terminates after at most  $\log N$  iterations. Thus, the worst-case running time of RS is  $O(N^3 \log N)$ .

### 3.2 Blocked Repeated Squares

The release implementation of RS uses a naive matrix multiplication kernel to square  $L$ . We developed an optimized matrix multiplication kernel that implements a variety of optimizations. We use this kernel in an optimized repeated squares implementation that we call BLOCK. In this subsection, we describe the optimizations implemented by BLOCK.

**Blocking** A naive matrix multiplication kernel has very poor cache locality. BLOCK implements a blocked matrix multiplication to greatly improve cache locality. An optimal block size of 128 was found empirically.

**Copy Optimization** When BLOCK multiplies two sub-blocks, it first copies them into a smaller, aligned buffer. This copy optimization allows the compiler to more aggressively vectorize loops over the sub-blocks and it makes memory accesses to the sub-blocks much more regular.

**Compile-Time Loop Bounds** If the size of a matrix is not divisible by the size of a block, then not all sub-blocks have the same size, and the size of a sub-blocks is only known at runtime. When BLOCK multiplies two sub-blocks, it first checks to see if they are full-sized blocks, and if they are, it multiplies them using a kernel whose loop bounds are known at compile-time. By knowing loop-bounds at compile time, the compiler more aggressively optimizes this code path. If the sub-blocks are not full-sized blocks, then it multiplies them using loops with bounds known at runtime. Since most blocks are full-sized blocks, BLOCK often performs block multiplication with the fully optimized kernel.

**Vectorization** BLOCK organizes all loops and array accesses to be fully vectorized.

### 3.3 Floyd-Warshall

The Floyd-Warshall algorithm (FW) is a dynamic programming algorithm developed by Robert Floyd and Stephen Warshall in the 1960's. Consider a directed graph with  $N$  nodes ordered  $1, \dots, N$ . Let  $l_{i,j}^k$  be the length of the shortest path from vertex  $i$  to vertex  $j$  using only intermediate nodes from  $\{1, \dots, k\}$  or  $\infty$  if no such path exists. FW relies on the following recurrence:

$$l_{i,j}^{k+1} = \min(l_{i,j}^k, l_{i,k+1}^k + l_{k+1,j}^k)$$

The base case  $l_{i,j}^0$  is the weight of the edge from vertex  $i$  to vertex  $j$  or  $\infty$  if no such edge exists. FW initializes  $L^0$  and iteratively computes  $L^{k+1}$  from  $L^k$  using the above recurrence.  $L^{k+1}$  can be computed in  $O(N^2)$  time and the algorithm terminates when it computes  $L^N$ . Thus, the algorithm runs in  $O(N^3)$  time.

## 4 Parallelization

We parallelized the RS, BLOCK, and FW algorithm using three parallelization mechanisms: OpenMP, MPI, and a hybrid of the two. In this section, we describe in detail how each mechanism was used for parallelization.

### 4.1 OpenMP

Parallelizing each algorithm using OpenMP is simply a matter of annotating the outer loop of each algorithm's respective `square` functions with `#pragma omp parallel for`. The implementations of RS, BLOCK, and FW using OpenMP are in `rs-omp.c`, `block-omp.c`, and `fw-omp.c`.

### 4.2 MPI

Parallelizing the algorithms with MPI requires decomposing the problem into sub parts, similar to the idea of domain decomposition from the last project. We drew inspiration from previous work with distance-vector routing using Bellman–Ford in determining shortest distances between routers. In that algorithm, each router sends its distances to all other routers (each time there is a change) to its neighbors which then use that information to update their own distances. This continues until no routers have changed distances.

In our case, we do not necessarily have a single thread per node, as there may not be enough hardware threads available. Instead, each “router” is an MPI rank and is responsible for a set of nodes rather than a single node. Each MPI rank calculates the minimum distance to each of its nodes from all other nodes going through each node between 1 and  $N$ .

For RS and BLOCK, each rank also determines if any distances have changed. All MPI ranks then synchronize, to gather distances from one another and determine if any distances have changed. If none have changed, then the algorithm terminates and the “master” rank (rank 0) outputs checksum and timing information. To determine if any distances have changed we use `mpi_allreduce` on each rank's done variable. To synchronize distances across all ranks we use `mpi_allgather` which sends each rank's distances to all other ranks and collects them from every rank, including itself, into a single buffer. The implementation

of FW must synchronize for every iteration of  $k$ , and does not need to check if its finished as the algorithm will always finish in the same number of loops ( $N$ ).

The implementations of RS, BLOCK, and FW using MPI are in `rs-mpi.c`, `block-mpi.c`, and `fw-mpi.c`.

### 4.3 Hybrid

MPI and OpenMP interact seamlessly when put together, making it easy to combine our MPI implementation with the release OpenMP implementation. Given a fixed number of MPI ranks,  $r$ , and  $p$  available threads then each MPI rank will have access to  $\frac{p}{r}$  threads which can be used in OpenMP parallel sections of code. This means we do not take full advantage of all threads available to us if  $p$  is not divisible by  $r$ . We implemented a hybrid version of each algorithm (by combining our MPI implementation with the OpenMP portions of the original implementation) in `rs-hybrid.c`, `block-hybrid.c`, and `fw-hybrid.c`.

## 5 Evaluation

We evaluated our MPI and hybrid implementations of RS, BLOCK, and FW with strong scaling studies across a range of problem sizes. For RS and BLOCK, we used a baseline of RS-OMP, which has access to 24 threads. For FW, we used a baseline of FW-OMP, which has access to 24 threads. For RS and BLOCK, we also performed a weak scaling study with a constant amount of work per thread corresponding to a graph of size  $N = 500$ .

### 5.1 RS

#### 5.1.1 MPI

In Figure 4a, we can see that increasing the number of MPI ranks, which in turn decreases the number of OMP threads per rank, does increase speedup, but for any problem size larger than 960 does not get above 1. Further there is a drop when ranks goes beyond 12. This is because of increased MPI overhead across 2.

In weak scaling (Figure 4b) we that performance per thread drops as speedup falls. This is likely because of the increased overhead of synchronizing with 1 more thread as we add 1 more thread.

#### 5.1.2 Hybrid

In Figure 4c we can see that increasing the number of MPI ranks (which in turn decreases the number of OMP threads per rank) has mixed speedup for most  $n$ . When the number of MPI ranks is less than 10 we sometimes have speedup larger than 1, but as we go beyond 10 and 12 speedup drops off. This is because of increased MPI overhead across 2 chips and the hybrid codes inability to take advantage of all possible threads when there are more than 12 MPI ranks.

The hybrid implementation drops speedup more rapidly than the MPI implementation for weak scaling (Figure 4d). This is probably because inability to take advantage of the full 24 hardware threads when the number of ranks does not divide 24 perfectly.

## 5.2 Block

### 5.2.1 MPI

Our block implementation with MPI has much higher speedups than our repeated squares with MPI. It crosses 5x speedup for almost all problem sizes at some point (figure ??). In particular we see that smaller problems (960 being a notable outlier) with larger speedup. Speedup once again increases until 12 and then drops (due to multiple chips) and then increases once again.

### 5.2.2 Hybrid

Our block implementation with hybrid has much higher speedups than our repeated squares with hybrid, however the speedups are smaller than for MPI (figure ??). Speedup varies between sizes, particular with fewer than 12 ranks due to not always taking advantage of all 24 hardware threads. Once again, problem size 960 has much more speedup than other sizes.

## 5.3 FW

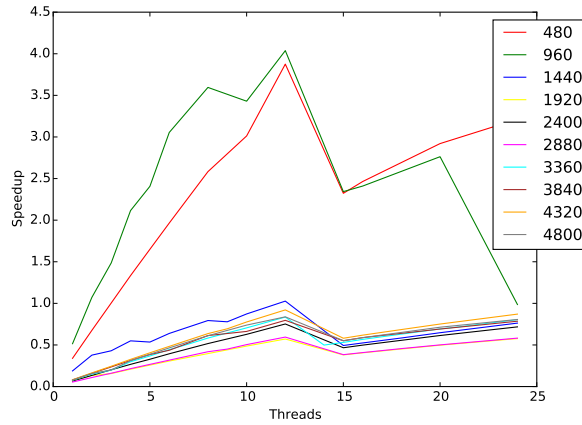
### 5.3.1 MPI

The performance of FW with MPI is typically poor, this is because of the increased number of synchronizations ( $O(N)$ ) as compared to  $O(\log(N))$ . We see this in figure ??, speedup slowly increases for all sizes, but in all cases except for the smallest size (480), never surpasses one; in fact, all but 480 and 960 never cross 0.5 speedup.

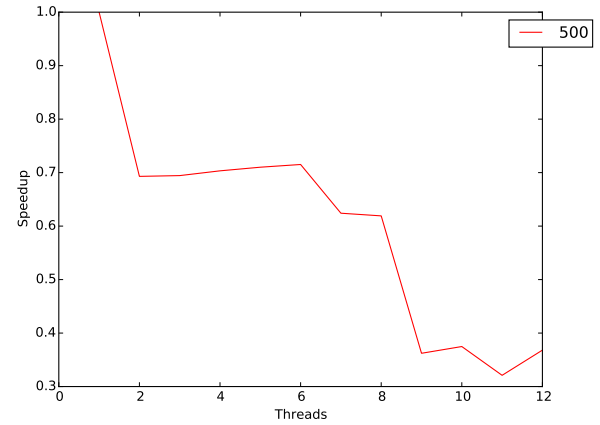
### 5.3.2 Hybrid

The performance of FW with hybrid actually tends to decrease as ranks increase, this is because of the increased number of synchronizations once again along with inability to use all 24 hardware threads. We see this in figure ??.

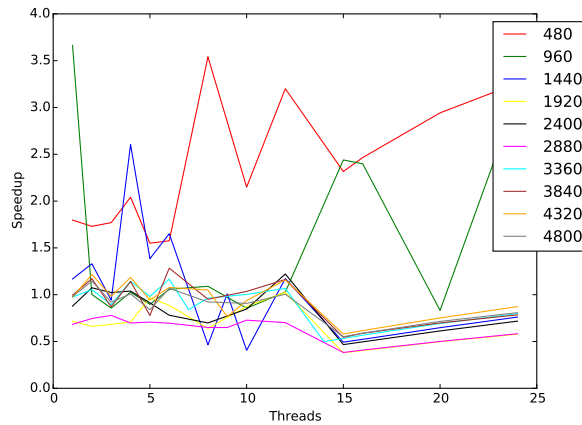
## 6 Conclusion



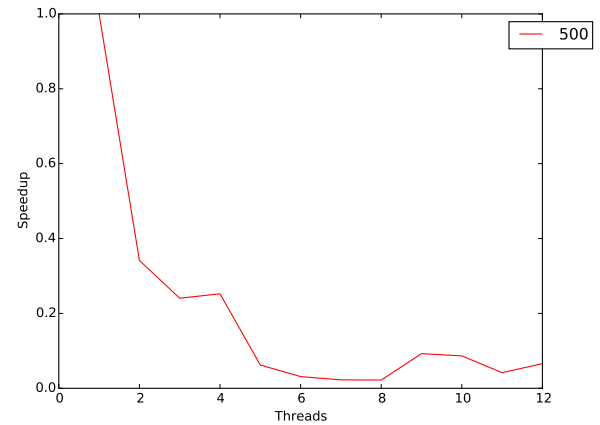
(a) RS-MPI strong scaling (RS-OMP baseline)



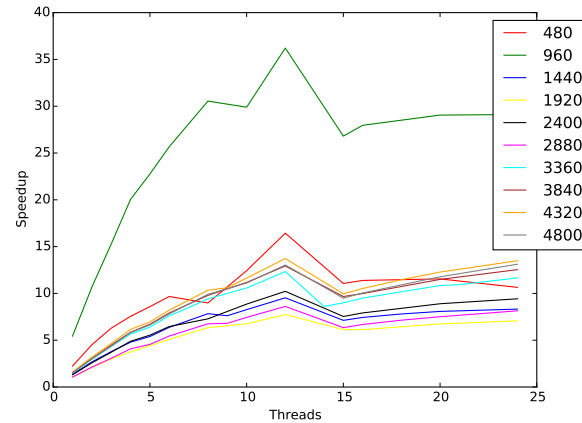
(b) RS-MPI weak scaling



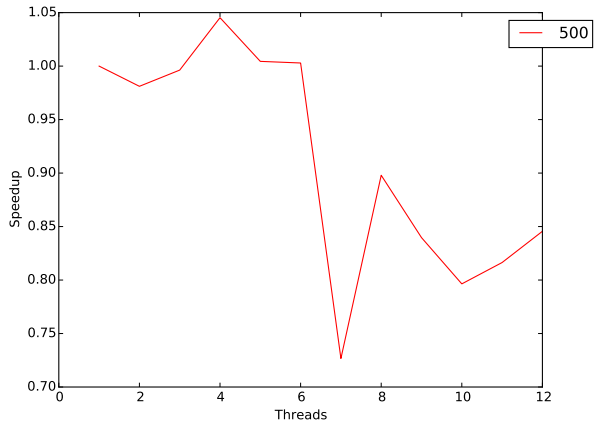
(c) RS-HYBRID strong scaling (RS-OMP baseline)



(d) RS-HYBRID weak scaling

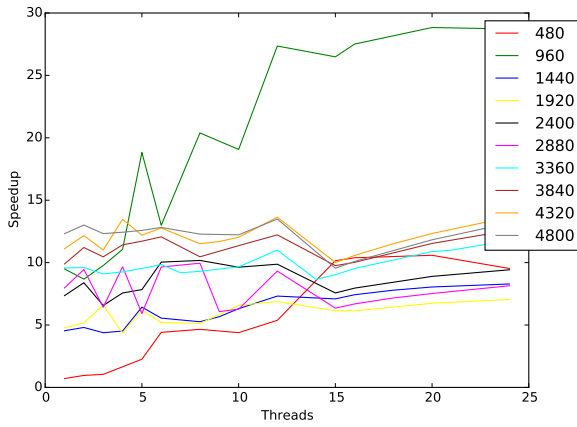


(e) BLOCK-MPI strong scaling (RS-OMP baseline)

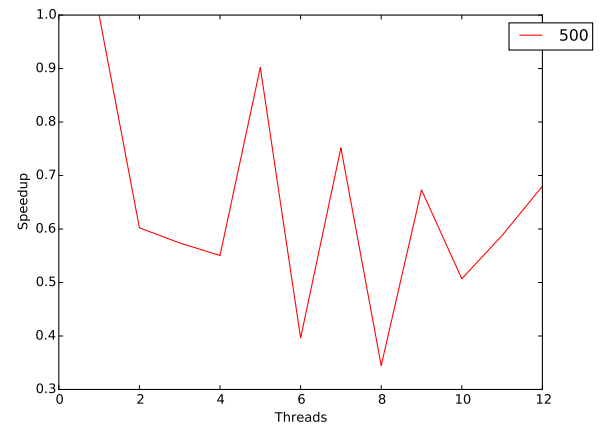


(f) BLOCK-MPI weak scaling

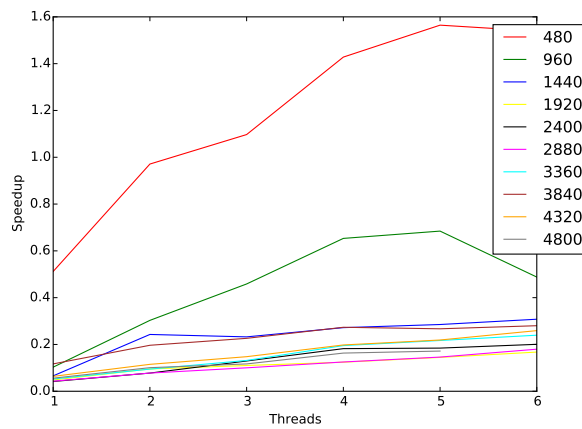
Figure 4: RS-MPI, RS-HYBRID, and BLOCK-MPI



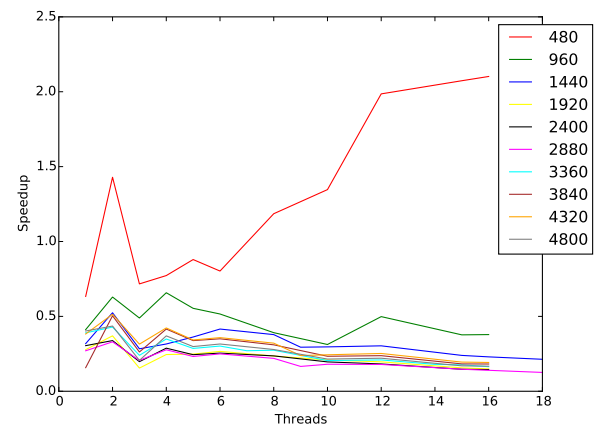
(a) BLOCK-HYBRID strong scaling (RS-OMP baseline)



(b) BLOCK-HYBRID weak scaling



(c) FW-MPI strong scaling (FW-OMP baseline)



(d) FW-HYBRID strong scaling (FW-OMP baseline)



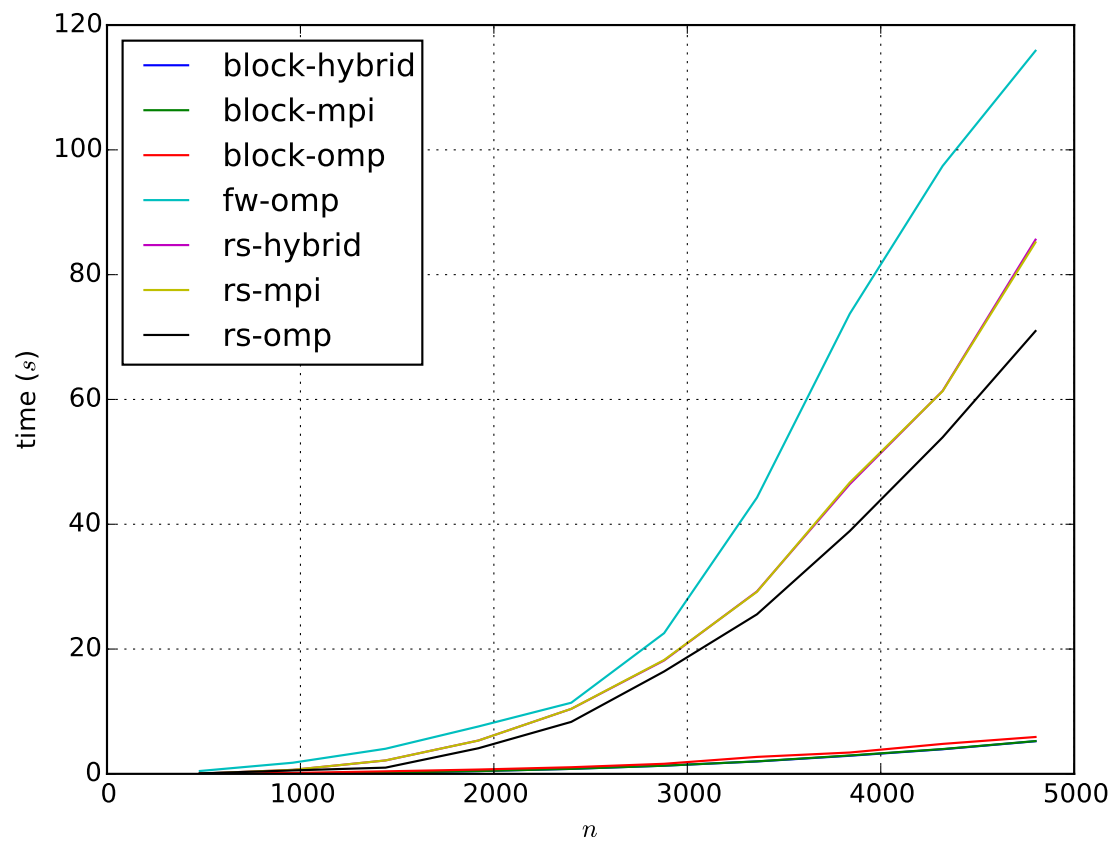


Figure 5: Performance comparison of all algorithms excluding FW-MPI and FW-HYBRID.