Optimization of the The Floyd-Warshall Algorithm

Ravi Patel (rgp62), Saurabh Netravalkar (sn575), Patrick Cao (pxc2)

MPI Implementation

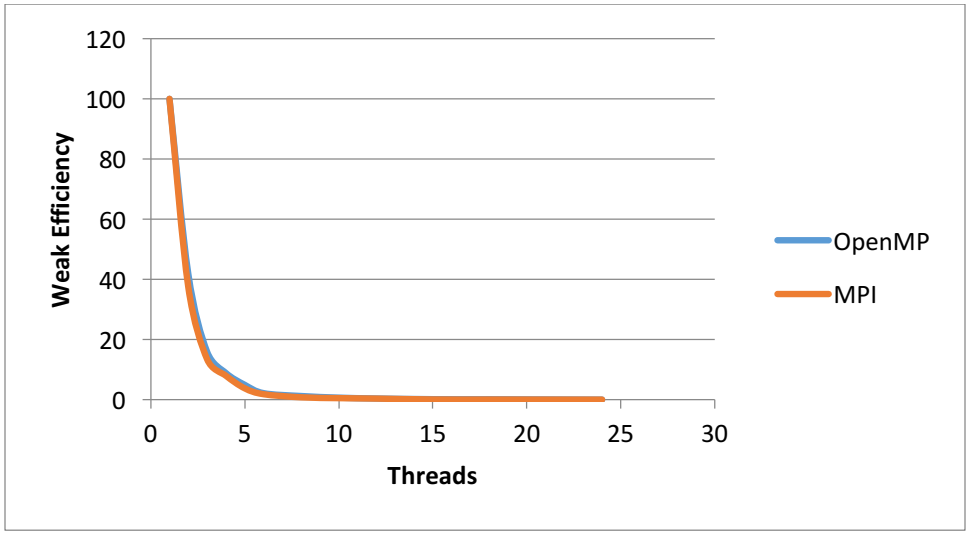
We implemented an MPI version of the Floyd-Warshall algorithm using a domain decomposition approach. The pairwise path distances are stored in a n by n matrix. Each processor is tasked with computing a portion of this matrix. The processors retain a copy of the full matrix to compute their portion. Every iteration the processors copy their part into a buffer and perform the minimum operation on it using data from the full matrix. The processors share their part of the work with each other using an Allgather operation and copy back this data into their own copy of the full matrix for the next iteration. This is repeated until every processor reports no more operations are necessary. Below is the relevant section of code:

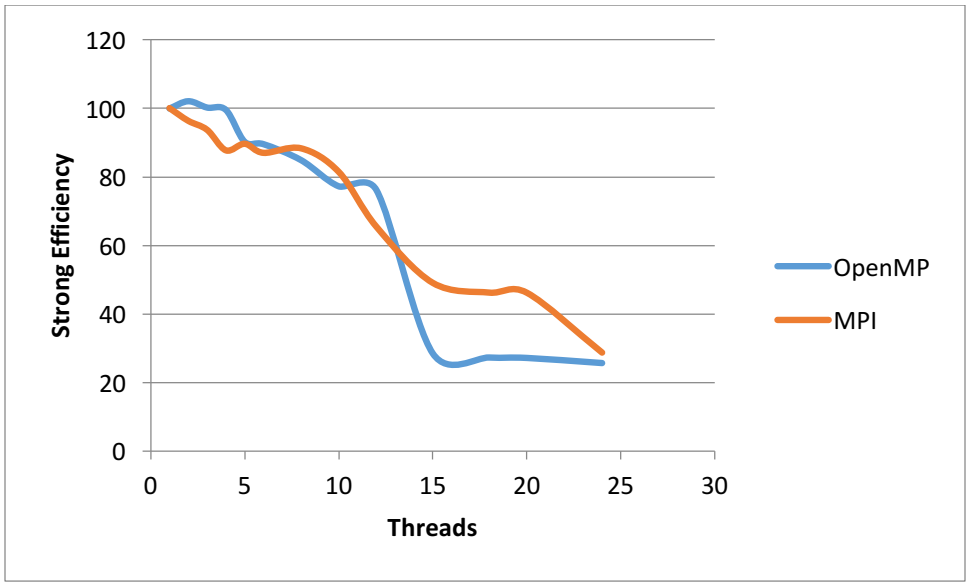
```
for (int done2 = 0; !done2; ) {
   int count = 0;
   for (int j = iranky*ny; j < (iranky+1)*ny; ++j)
      for (int i = irankx*nx; i < (irankx+1)*nx; ++i){
         lnew[count]=l[j*n+i];
         count++;
      }
   done = square(n, l, lnew,nprocx,nprocy,irank);
   MPI_Allgather(lnew,nx*ny,MPI_INT,lnew2,nx*ny,MPI_INT,MPI_COMM_WORLD);
   MPI_Allreduce(&done,&done2,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
   count = 0;
   for (int proci = 0; proci < nprocx*nprocy; proci++){
      rankx = proci % nprocx;
      ranky = proci / nprocx;
      for (int j = ranky*ny; j < (ranky+1)*ny; ++j)
         for (int i = rankx*nx; i < (rankx+1)*nx; ++i){
            l[j*n+i]=lnew2[count];
            count++;
         }
   }
}
```

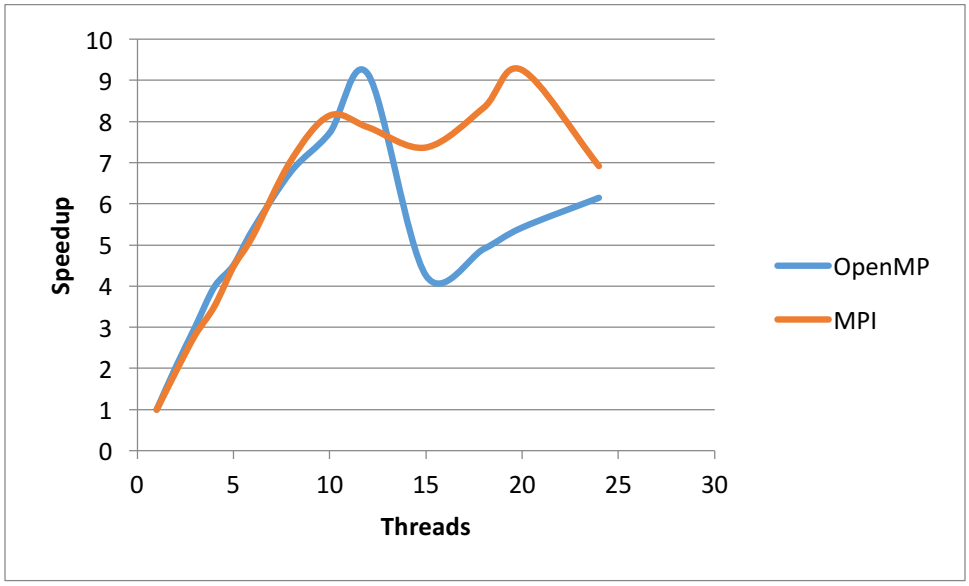
square has been modified to only operate on the processor's domain:

```
for (int j = iranky*ny; j < (iranky+1)*ny; ++j) {
    for (int i = irankx*nx; i < (irankx+1)*nx; ++i) {
  ...
```

Below are the weak scaling efficiency, strong scaling efficiency, and speed up charts.







We found that sectioning the matrix into columns for each processor was most efficient as it leads to the best vectorization. However, further optimizations, such as blocking, will be implement in addition. Weak and strong scaling analysis was performed to compare this MPI implementation to the original OpenMP implementation. From weak scaling, the OpenMP version performed consistantly better than the MPI version, marginally at low thread counts, but significantly at higher thread counts. Strong scaling and the speedup graph shows that the MPI code performs about the same as the OpenMP code for low thread counts but better for high thread counts, although this advantage disappears at the highest 24 thread count. The Allgather communication cost is extremely high so it ideal to communicate enough information that communication occurs less often and communication latency is limited. However, communication must occur often enough so that the processors do not have to share so much information with each other.

Serial Tuning of the Original OpenMP code

The memory access patterns of the $O(n^3 \log n)$ Floyd-Warshall Algorithm are exactly the same as the matrix multiplication kernel, with the only difference of the summation operation being replaced by minimum computations. Hence, we approached serial tuning in a similar way to that of matrix multiply. The optimizations we performed are as follows:

1. **Redundant Loop Elimination**: We merged the for loop in the infinitize function with the loop in shortest_paths function which sets self looping paths to length zero.

Before: infinitize function	**Before: shortest_paths function**

for (int i = 0; i < n*n; ++i)
  if (l[i] == 0)
  l[i] = n+1;
	for (int i = 0; i < n*n; i += n+1)
	  l[i] = 0;

After: **infinitize function**
for (int i = 0; i < n*n; ++i)
  if (l[i] == 0 && i % (n + 1) != 0)	

2. Elimination of Expensive memcpy Operations by Swapping Pointer Roles

The original code uses two matrices 'l' and 'lnew' while computing the shortest paths at every iteration. Matrix 'l' is used to perform the computations and results are written out into 'lnew'. At the end of each iteration, the entire new matrix 'lnew' is copied into 'l'. We eliminated this by using 'l' and 'lnew' swapping roles of 'l' and 'lnew' in every iteration, among being the input matrix and being the matrix where results are written out respectively.

Note: After the end of all iterations, the function expects to produce results in matrix 'l'. Hence, if we terminate in an oddth iteration, we need to perform one memcpy of 'lnew' back into 'l'.

Code Snippet

```
int* restrict lnew = (int*) calloc(n*n, sizeof(int));
  int flag = 1;
  for (int done = 0; !done; ) {
    done = flag ? square(n, l, lnew) : square(n, lnew, l);
    flag = !flag;
  }
  if(!flag) {
    memcpy(l, lnew, n*n * sizeof(int));
  }
```

3. Copy Optimization

In the square function, we created an extra matrix 'ltrans', a transposed version of 'l' in order to gain cache hits in the innermost 'k' loop of the i,j,k loop ordering.

Code Snippet:

Copy Optimization for better cache hits in the OpenMP Parallel For

```
int* restrict ltrans = malloc(n*n*sizeof(int));
for (int j = 0; j < n; ++j) {
  for (int i = 0; i < n; ++i) {
    ltrans[i*n + j] = l[j*n + i];
  }
}
```

Innermost Loop of the OpenMP Parallel For

```
for (int k = 0; k < n; ++k) {
        int lik = ltrans[i*n+k];
        int lkj = l[j*n+k];
        if (lik + lkj < lij) {
          lij = lik+lkj;
          done = 0;
        }
      }
```

Profiling with Amplxe Hotspots

Function	Original	Loop Elimination	Memcpy Elimination	Copy Optimization
square	40.431	41.269	**31.663**	**6.608**

Observations and Further Steps

The Copy Optimization works wonders and gives almost a 7x speedup from the original. The next goal will be to implement blocking by integrating it with the respective OpenMP/MPI based blocked access with **block level copy optimizations** for each processor.

Blocking with Block Level Copy Optimizations using Square Tiles

1. Tiling

We mapped the problem to match the Matrix Multiplication problem to enable effective copy optimizations and vectorization. Thus, we created another copy of the 'l' matrix named 'lcopy', such that the Floyd Warshall Algorithm uses 'l', 'lcopy' and 'lnew' analogously to matrices A,B and C in Matrix-Matrix Multiplication of A and B to give C.
We implemented the single level blocked kernel (such that the blocks fit in L2 cache).

Code Snippet

Blocking Into Tiles

```
const int n_blocks = n / BLOCK_SIZE + ( n % BLOCK_SIZE ? 1 : 0 );
int bi, bj, bk;
int done = 1;

for ( bi = 0; bi < n_blocks; ++bi ) {
   const int i = bi * BLOCK_SIZE;
   for ( bj = 0; bj < n_blocks; ++bj ) {
      const int j = bj * BLOCK_SIZE;
      for ( bk = 0; bk < n_blocks; ++bk ) {
         const int k = bk * BLOCK_SIZE;
         if(do_block( n, l, lcopy, lnew, i, j, k ) == 0) {
            done = 0;
         }
      }
   }
}
```

2. Copy Optimization using local Buffers

Local buffers tremendously boost performance by fitting into L2 cache.

Code Snippet

Lnew_buf is a Buffer for the Output Block lnew

```
for( j = 0; j < N; ++j ) {
      for( i = 0; i < M; ++i ) {
         lnew_buf[ i + M * j ] = lnew[ i + lda * j ];
      }
}
```

3. Transposing one copy of 'l' for Vectorization

Transposing 'l' while storing it into 'l_buf' enables effective vectorization of the inner loop of the Floyd Warshall Algorithm

Code Snippet

Copy Block of l into l_buf in row-major form to aid vectorization of innermost loop	**Enables Vectorization of Inner loop by improving locality of reference**

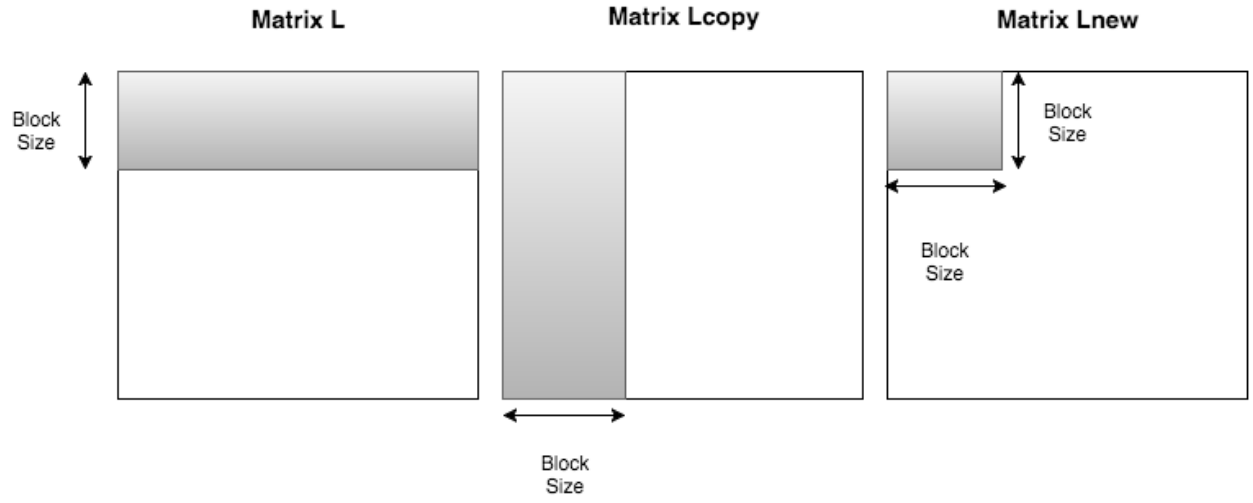
for(k = 0; k < K; ++k) {
  for(i = 0; i < M; ++i) {
  l_buf[K * i + k] = l[i + lda * k];
  }
}
	#pragma vector aligned
	  for (k = 0; k < K; ++k) {
	  int lij = l_buf[K * i + k] + lcopy_buf[k + K * j];
	  if(lij < lnew_buf[i + M * j]) {
	  lnew_buf[i + M * j] = lij;
	  done = 0;
	  }
	}

Parallelization of Serial Tuned Blocked Code in OpenMP

1. Challenges with Parallel Matrix-Matrix Multiplication based square tiling approach

Since, partial results of the same block of the output matrix could be computed in parallel by multiple threads, there are write conflicts which mandate that either access to the output matrix be atomic or that each processor have its own copy of a block which would then be reduced via a 'min' operation as required by the Floyd Warshall Algorithm.

2. Our Solution: Every Processor Computes completely a Square Tile of 'lnew' by Striping relevant Rows of 'l' and Columns of 'lcopy'



Each Processor Uses a row of L and a Colum of Lcopy to Compute a Square BLock of Lnew

Final Timing Results on Xeon Boards

Timing in Seconds

Matrix Dimension N	**1000**	**2000**	**3000**	**4000**	**5000**
Naïve OpenMP	0.455697	3.45738	11.9905	42.7985	81.9797
Serial Tuned Blocked and Vectorized	0.681521	5.3975	18.4377	43.5968	84.7128
Unblocked, Vectorized OpenMP Code	0.268743	**0.910615**	3.53801	11.6873	**18.4562**
Final Tuned, Blocked and Vectorized OpenMP code (Block Size 32)	0.251862	0.961094	3.57934	11.2161	18.677
Final Tuned, Blocked and Vectorized OpenMP code (Block Size 96)	**0.23143**	0.959934	**3.49549**	**10.4431**	25.0266
Max. Speedup w.r.t. Naïve OpenMP Code	**1.9690**	**3.7968**	**3.4303**	**4.0983**	**4.4419**

Observations

1. Well Tuned Serial code runs comparably fast as naively parallelized OpenMP code.

2. Applying Copy Optimizations, Vectorization and other above mentioned smart optimizations like Pointer Swapping with OpenMP gives a drastic speedup ranging from 2 to 4.5 from input square matrix sizes of 1000 to 5000 respectively!

3. Unblocked as well as Blocked-Striped Parallel OpenMP Implementations run in similar or slightly faster time depending on Block Size because of dependencies of Strips of ‘l’ and ‘lcopy’ on the input size ‘n’ which when becomes large enough, strip sizes of BLOCK_SIZE x n no longer fit in L2 cache and hence run in a time comparable to the unblocked code.