Optimization of the The Floyd-Warshall Algorithm

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

MPI Implementation

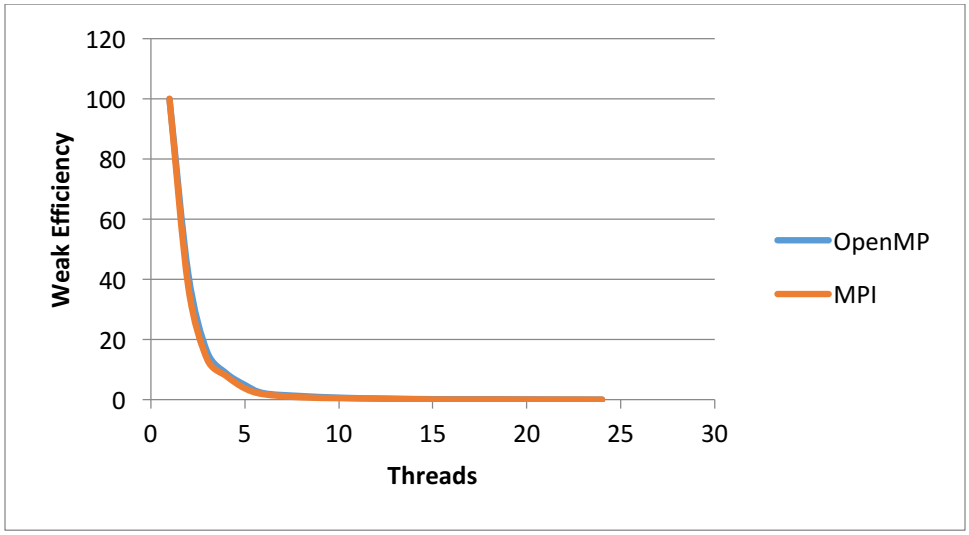
We implemented an MPI version of the Floyd-Warshall algorithm using a domain decomposition approach. The pairwise path distances are stored in a n by n matrix. Each processor is tasked with computing a portion of this matrix. The processors retain a copy of the full matrix to compute their portion. Every 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:

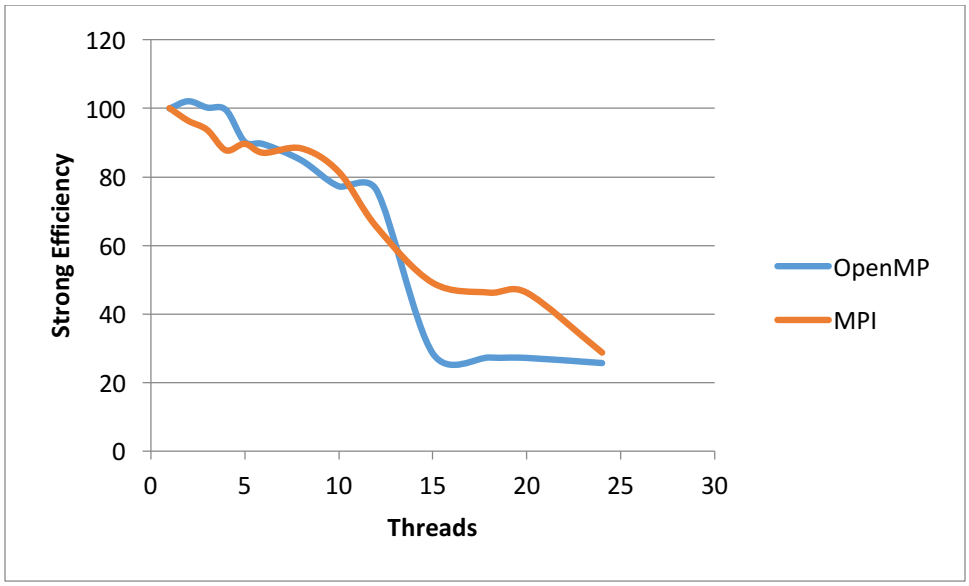
```
  for (int done2 = 0; !done2; ) {
    int count = 0;
    for (int j = iranky*ny; j < (iranky+1)*ny; ++j)
      for (int i = irankx*nx; i < (irankx+1)*nx; ++i){
        lnew[count]=l[j*n+i];
        count++;
      }
    done = square(n, l, lnew,nprocx,nprocy,irank);
    MPI_Allgather(lnew,nx*ny,MPI_INT,lnew2,nx*ny,MPI_INT,MPI_COMM_WORLD);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

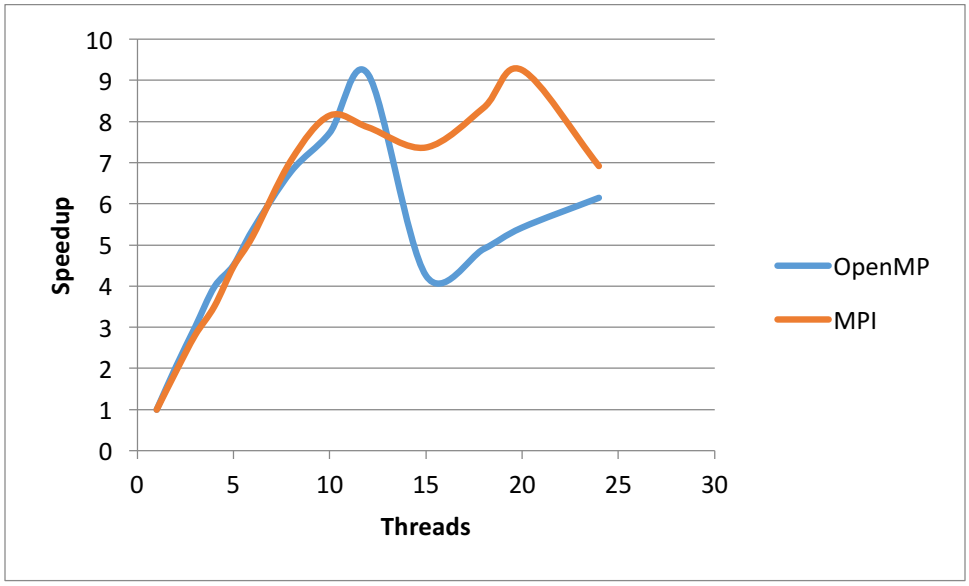
square has been modified to only operate on the processor's domain:

```
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 Plans

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.