CS5220 HW2 Report: Shallow Water Simulation

Team 5: Unmukt Gupta (ug36), Xinyi Wang (xw327), Ji Kim (jyk46)

1. Introduction

In this report, we explore parallelization strategies for the Floyd-Warshall algorithm for finding the shortest paths between all nodes in a graph. We start by profiling the naive implementation of the code in order to identify bottlenecks as well as build our intuition for developing a relatively accurate performance model. This information is used to guide the parallelization of the code for both the compute nodes (i.e., Intel Xeon E5-2620) and the accelerator boards (i.e., Intel Xeon Phi 5110P) on the Totient cluster. We further implement and evaluate several optimizations for tuning the parallel code.

2. Profiling the Floyd-Warshall Algorithm

2.1. Identifying Bottlenecks

From the VTune profiling reports (Figure 1 and Figure 2), we can see that a large proportion of the CPU time is spent in the function square, specifically in lines 52, 54 and 51. Upon further inspection of the code, we observe that in the innermost for-loop (line 52), we are accessing the array l at strides of size n to iterate over elements in row i. Since the matrix is stored in column-major order, accessing elements in a single row causes a stride of size n, resulting in poor spatial locality and inefficient use of the cache, thus contributing to the high CPU timing for that particular line as reported in Figure 2.

Function	Module	CPU Time
square	path.x	29.615s
square	path.x	15.876s
__kmp_fork_barrier	libiomp5.so	12.457s
__kmp_launch_thread	libiomp5.so	1.760s
__kmp_join_call	libiomp5.so	0.690s
fletcher16	path.x	0.020s

Figure 1: VTune profiling output: Hotspots report (for n = 2000, p = 0.05)

```
amplxe-cl -report hotspots -r r002hs -group-by source-line
```

From the original implementation of the code, we expected that the memcpy from lnew to l after each iteration could potentially be rather expensive – especially as n becomes large. Since the results of each iteration only depends on the matrix from the previous iteration (and not the values from the current iteration), and given that we already have two separate matrices, we could avoid this copying operation by swapping the matrices between iterations – reading from l and writing to lnew in one iteration, and reading from lnew and writing to l in the next. In practice however, this did not give us much speedup, due to the fact that the actual number of iterations that the algorithm takes to converge is rather low (for p = 0.05, and n = 2000, it typically converges within 2-4 iterations). This is also confirmed by the outputs from VTune profiling reports, where for n = 10000, the CPU time used by memcpy (0.270s) is rather insignificant as compared to the time spent in the functions square (1368.604s, after tuning with copy optimization).

Source File	Source Line	Module	CPU Time
path.c	52	path.x	33.688s
kmp_barrier.cpp	1,573	libiomp5.so	12.457s
path.c	54	path.x	8.613s
path.c	51	path.x	2.981s
kmp_runtime.c	5,635	libiomp5.so	1.760s
kmp_runtime.c	2,354	libiomp5.so	0.690s
path.c	56	path.x	0.080s
path.c	50	path.x	0.060s
path.c	49	path.x	0.049s
path.c	192	path.x	0.020s
path.c	90	path.x	0.010s

Figure 2: VTune profiling output: (by line in source)

```
42  int square(int n,               // Number of nodes
43             int* restrict l,     // Partial distance at step s
44             int* restrict lnew)  // Partial distance at step s+1
45  {
46      int done = 1;
47      #pragma omp parallel for shared(l, lnew) reduction(&& : done)
48      for (int j = 0; j < n; ++j) {
49          for (int i = 0; i < n; ++i) {
50              int lij = lnew[j*n+i];
51              for (int k = 0; k < n; ++k) {
52                  int lik = l[k*n+i];
53                  int lkj = l[j*n+k];
54                  if (lik + lkj < lij) {
55                      lij = lik+lkj;
56                      done = 0;
57                  }
58              }
59              lnew[j*n+i] = lij;
60          }
61      }
62      return done;
63  }
```

Figure 3: function square, showing line numbers –

3. Parallelizing the Floyd-Warshall Algorithm

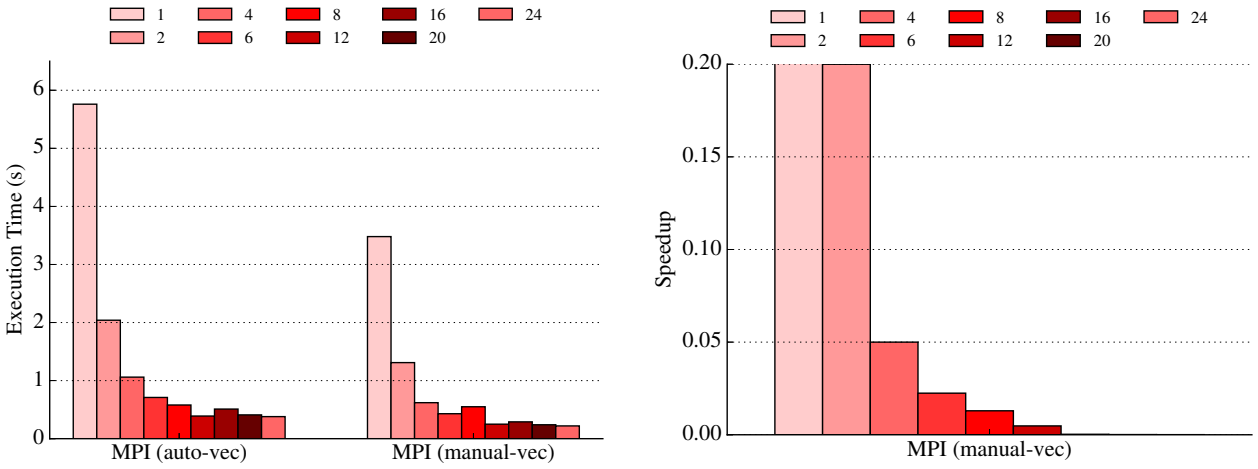
3.1. Parallelizing for the Compute Node

First an attempt was made to improve the data access pattern inside the square function. This was done by removing data dependencies from the k-loop, and making it the outermost loop. In doing so, we access data without striding through the adjacency matrix in the innermost (i) loop. For the purpose of distributing the adjacency matrix, we use a 1D domain decomposition method, wherein each process owns a non-overlapping, vertical "stripe" in the global grid. This assignment of chunks of continuous columns is done, using MPI_Scatterv, in order of process rank. In case the grid size isn't exactly divisible by the number of processes, we assign (for simplicity) the remainder of the columns to the last process. As we move along the columns in the outermost(k-loop), we do a modulus operation on the k value to calculate which process owns the column, and broadcast this column to every other process.

To strike a balance between communication and computation costs, we tried broadcasting the entire chunk of columns (instead of just one column at a time), that any given process owns, to every other

process. And it turns out that, atleast for a very specific case of 24 processes with a grid size of 2000, the performance decreases slightly.

We also see performance dips when the number of cores (equivalently, the number of stripes) is equal to integer powers of 2 (4, 8, and 16). This could perhaps be attributed to an increase in cache misses. We could perhaps do a copy optimisation operation that would help in leveling these dips out.



(a) **Strong Scaling Study**

(b) **Weak Scaling Study**

Figure 4: Performance Results of Parallel Implementation of Floyd-Warshall Algorithm Running on Compute Nodes – Performance of the parallel implementation of the Floyd-Warshall algorithm running on the Totient compute nodes compared against the provided serial implementation. For the strong scaling study, a dataset with 2000 nodes is used, and absolute execution time is shown for both the auto-vectorized and manually vectorized code. For the weak scaling study, the dataset is increased at the same rate as the number of ranks, and all speedups are normalized to the serial implementation using manually vectorized code.

3.2. Parallelizing for the Accelerator Device

There are some notable limitations with combining MPI with offloading computation to the accelerator. The most significant limitation is that we *cannot* invoke MPI routines inside of the offloaded kernel. Intel recommends three approaches for offloading code that uses MPI:

- Native Execution: All MPI ranks live on the accelerator, binary must be executed natively on the accelerator.
- Symmetric Execution: MPI ranks live on both the host and the accelerator, two separate binaries must be compiled.
- Host Execution: All MPI ranks live on the host, each rank can offload kernels separately.

For the purposes of this assignment, symmetric execution is not very practical, so the remaining options are native and host execution. For the sake of simplicity and due to issues running native MPI code on the Totient cluster, we focus on using host execution for offloading computation.

Because the parallelization for the compute nodes uses an outer loop that iterates across the k-domain, where each k-th iteration triggers a broadcast of some number of columns to all cores, the earliest point we can offload computation is right after this broadcast, effectively enclosing the j-domain and i-domain loops.

In our scheme, each MPI rank offloads these loops to the accelerator once they reach this point. Having each rank offload a separate kernel can cause suboptimal resource allocation inside the accelerator. Intel recommends either using a few dedicated ranks for the offloading, but this would limit our parallelism for the non-offloading ranks. Since the accelerators have longer vector lengths compared to the compute nodes (i.e., 512b vs. 256b), we ideally want the vectorized computation to happen on the accelerator as much as possible. Fortunately, in terms of memory transfer, the only significant regions of memory required by the offloaded kernel are the per-core local buffer and the column buffer that holds the broadcasted columns mentioned above.

The fact that each rank offloads computation *every* k-th iteration means that we want to maximize the compute density of the offloaded kernel. In other words, we want to amortize the overhead of the offload over the largest amount of work that we can assign to the accelerator at a time. One way we can do this is by broadcasting chunks of multiple columns every k-th iteration instead of just one column. For example, this decreases the order of MPI messages for the column broadcast from $O(n^2)$ to $O(num\_cores^2)$. Although this reduces inter-core communication, all ranks still need to synchronize at every k-th iteration. The offloading should also benefit from reducing the number of synchronization points. Ideally, we want all ranks to synchronize once per super-step (i.e., computing outputs for its assigned block). We can do this by increasing the per-core local buffer to hold the entire grid instead of just its assigned block. Each rank would only write to its assigned block but use the elements from other blocks as read-only, similar to how we used ghost cells in the previous assignment.

Currently, our parallelization for the accelerator runs at orders of magnitude slower than even the serial implementation of the algorithm (i.e., 3 minutes for a 2000 node graph). However, this was not using the potential optimizations described above, so we believe that there is still a much room for improvement.

One problem we ran into when implementing the parallelization for the accelerator was an offload error complaining that it could not find the offload entry. Apparently the source of this error was the use of forced inlining of the `square()` function. Only by disabling inlining for this function, the host was able to find the section of the binary corresponding to the offload kernel.

4. Tuning the Floyd-Warshall Algorithm

4.1. Copy Optimization

We implemented copy optimization to address the cache locality issue raised by the VTune profiler. At the beginning of each iteration of the function square, we create an additional transposed copy of the matrix, so that we can use this matrix when accessing the array in row-major order. We can see significant performance gains from doing so, especially as the number of nodes in the graph increases as shown in Figure 5.

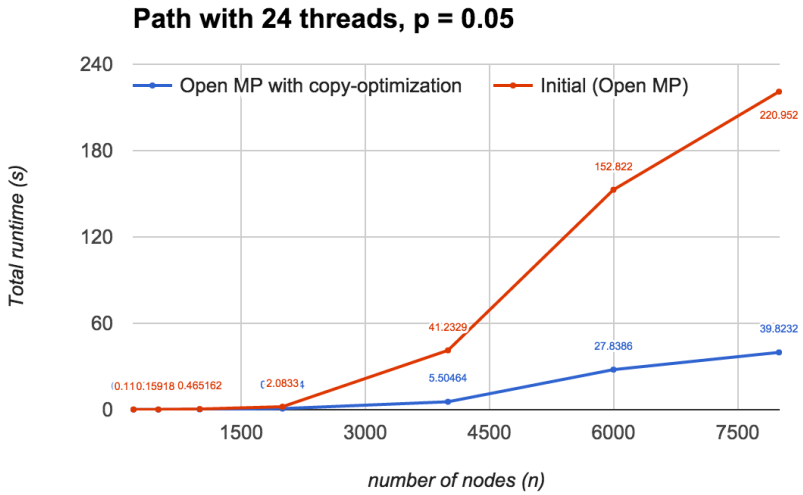


Figure 5: Performance of Copy Optimization – Execution time of the parallel implementation of the algorithm running with 24 threads across a wide range of dataset sizes with and without copy optimization.

From this experiment, we can see that when n is small (< 1000), the overheads from copying outweighs the benefits of cache locality, but as the number of nodes increases past 1000, we see significant improvement gains.

Source File	Source Line	Module	CPU Time
path.c	71	path.x	5.319s
kmp_barrier.cpp	1,573	libiomp5.so	3.039s
kmp_barrier.cpp	1,204	libiomp5.so	2.559s
path.c	73	path.x	1.812s
path.c	66	path.x	1.736s
kmp_csupport.c	2,381	libiomp5.so	0.640s

Figure 6: VTune profiling output of Open MP with copy/transpose optimization

The performance benefits of reading from a transposed copy of the matrix is also shown in the profiling output of the tuned code in Figure 6, as seen from the reduced CPU time.

4.2. Manual Vectorization

In order to best utilize the resources on both the compute nodes and the accelerator device, it is critical to ensure that the code we execute is properly vectorized. Although modern compilers are capable of auto-vectorizing code, the results can be inconsistent and often requires the user to massage the code in such a way that the compiler can recognize opportunities for vectorization. As such, manually vectorizing code using intrinsics is commonly used when optimizing performance critical kernels, such as the `square()` function in the Floyd-Warshall algorithm for this assignment.

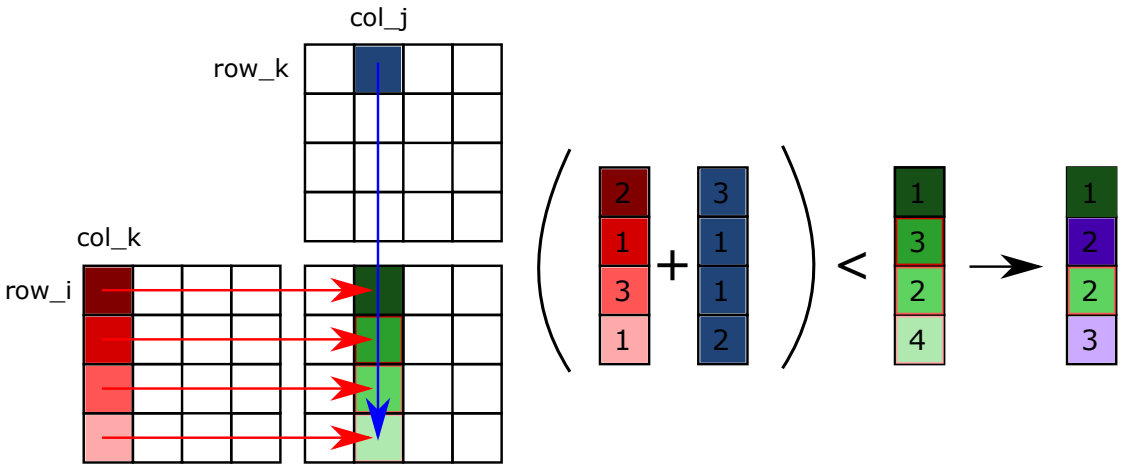


Figure 7: Overview of Manual Vectorization Strategy – All matrices shown on the left side represent the same shortest path matrix. The elements of the k-th column are added to the (k,j)-th element and compared against the current shortest path value in the (i,j)-th element. An example vectorized computation with a vector length of 4 is shown on the right, where the final result to be stored only contains the elements with the shortest paths between the sum vector and the current vector.

The vectorization strategy used here is similar to that used in matrix multiplication; the difference is that we are operating on a single matrix and we need to do a comparison instead of a multiplication. The primary insight here is that all elements in the k-th column need to be added the same element in the k-th row of the j-th column, where the (i,j)-th element is the output being calculated (i.e., the shortest path between node i and j). Therefore, we vectorize the computation for calculating a vector length worth of outputs in the j-th column as shown in Figure 7.

Using manual vectorization, we can vector load the elements in both the j-th and k-th columns, and broadcast the (k,j)-th element to a vector register. Although vector loads/stores to non-vector-aligned addresses are allowed, such unaligned or masked operations are much less efficient than the aligned variants. This becomes relevant when the number of nodes (i.e., the dimension of the matrix) is not evenly divisible by the vector length. In this case, even if we force the alignment of per-core local buffers, none of the columns after the first are guaranteed to be aligned. We address this by over-allocating the local buffers with extra padding elements so that each column is always a multiple of the vector length. The `pack_padded_data()` and `unpack_padded_data()` functions are used to copy columns between the unaligned local buffer to the aligned local buffer so that each column starts at a vector-aligned address. The overhead of this copy is captured in the timing loop.

The most challenging aspect of vectorization in this algorithm is actually the comparison. We use a vector greater-than comparison that can be used to check if the (i,k) to (k,j) path (i.e., the sum vector) is less than the current shortest path in (i,j) (i.e., the current vector). This returns a vector with an enabled mask (0xffffffff) for elements that had a true comparison, or a disabled mask (0x00000000) for elements that

had a false comparison. If any of the elements in the mask vector are 1s, we know that computation is not yet done and we need to clear the done variable. We can do this with a vector testc operation that returns true only if the mask vector is all 0s. In order to determine the output elements, we first AND the mask vector with the sum vector to zero-out the elements that were not shorter than the current shortest path. Conversely, we AND the NOT of the mask vector with the current vector to zero-out the elements that are not the shortest path anymore. By adding these masked vectors, we obtain a vector of the newest shortest paths that we can vector store to the j-th column.

If the matrix dimensions are not evenly divisible by the vector length, the mask vector must also be further masked to zero-out the elements that correspond to junk beyond the padding elements when computing the last iteration for a given column.

Experiments show that using manual vectorization achieves a 75% speedup compared to using only auto-vectorization on the parallel implementation of the Floyd-Warshall algorithm running on the compute nodes.