CS 5220: Project 3 Initial Report

Group 003: Robert Chiodi (rmc298), Stephen McDowell (sjm324)

1 Introduction

For the third assignment, we employ techniques we have learned in previous assignments as well as explore the capabilities of the Message Passing Interface (MPI) in the context of the Floyd-Warshall algorithm for computing all pairwise shortest paths. As described in the assignment description, this assignment is very similar to the matrix multiplication algorithm we have studied previously in terms of overall problem setup, and only differs in the actual computations being performed.

We begin by focusing our efforts on using MPI to allow computation across multiple nodes on the *totient* cluster, and then discuss optimizations of the algorithm to speedup the computation. We end with a discussion of offloading the computation to the Xeon Phi's, and a summary of our tactics.

It is worth mentioning at this point that the assignment asked us to implement a parallel MPI algorithm, and then choose between optimizing the given OpenMP implementation and our own MPI implementation. We choose to do both for two reasons:

1. The optimizations for the OpenMP implementation are very much the same as the MPI implementation in that the algorithm being executed is the same – the only difference being how threads are delegated work, and
2. Recent trends in high performance computing fall more along the lines of "first, use MPI to coordinate between multiple nodes, then use OpenMP on each node to further parallelize the work."

2 Optimizations

There are two general forms of optimizations that we employ:

1. Vectorization and/or compilation optimizations, and
2. Algorithmic optimizations and/or enhancements.

Each provide their own flavor of performance increase, but (1) is not as applicable for this assignment as it has been in the past since we do not have a large concern of optimizing the throughput of floating point operations. As such, after a few modifications, the majority of optimization efforts should be focused on (2).

2.1 Vectorization and Compilation Strategies

Though at a first glance there do not seem to be many opportunities for vectorization, we elect to give `icc` the hints it needs to optimize as much as possible. To enable portability, the following compilation hints are defined:

```
#ifdef __INTEL_COMPILER
    #define DEF_ALIGN(x) __declspec(align((x)))
    #define USE_ALIGN(var, align) __assume_aligned((var), (align));
#else // GCC
    #define DEF_ALIGN(x) __attribute__ ((aligned((x))))
    /* __builtin_assume_align is unreliabale... */
    #define USE_ALIGN(var, align) ((void)0)
#endif
```

These two must be used in lock-step. When declaring a variable of a specific alignment, we use `DEF_ALIGN` to declare that this variable has a specific alignment. An example usage would be

```
  DEF_ALIGN(BYTE_ALIGN)
  int * restrict lnew = (int *)_mm_malloc(num_bytes, BYTE_ALIGN);
```

noting that declaring alignment is not enough, it must be enforced through either aligned allocators or `_mm_malloc`. However, declaring alignment alone is not enough to enable the compiler to make the proper decisions when arranging loops and vectorizing. For example, when you pass a variable to a function, you need to indicate to the compiler that it has a given alignment, as in

```
int square(int n,                // Number of nodes
           int * restrict l,     // Partial distance at step s
           int * restrict lnew) { // Partial distance at step s+1

    USE_ALIGN(l,    BYTE_ALIGN);
    USE_ALIGN(lnew, BYTE_ALIGN);
    .
    .
    .
}
```

When used correctly, the `DEF` – `USE` strategy can be quite effective at optimizing, however for this program it does not appear to be too effective. Comparing to the base code, including this simple two-phase process gives the following increases:

# of Threads	Basic Time (s)	Optimized Time (s)	% Increase
1	538.585	538.384	0.04%
2	256.654	261.027	-1.70%
4	123.608	119.715	3.15%
8	62.932	62.926	0.01%
16	47.883	44.550	6.96%
24	41.951	44.717	-6.593%

Table 1: Comparison between base code and optimized code speeds for $n = 4000$ and probability of $p = 0.05$.

The only other optimization at this level that we tried was with respect to the branching in the square function:

```
    if (lik + lkj < lij) {
        lij = lik+lkj;
        done = 0;
    }
```

Basic parallel programming techniques dictate that branches are the death of all parallel algorithms – they induce thread divergence and prevent effective chaining of the parallel algorithm. Our remedy was to remove the branching using the following strategy:

```
    // https://graphics.stanford.edu/
    //~seander/bithacks.html#IntegerMinOrMax
    // If you know that INT_MIN <= x - y <= INT_MAX,
    // then you can use the following, which are faster
    // because (x - y) only needs to be evaluated once.
    int sum  = lik + lkj;
    int prev = lij;
    // min(sum, lij)
    lij = lij + ((sum - lij) & ((sum - lij) >>
     (sizeof(int) * CHAR_BIT - 1)));
    done = done && sum >= prev;
```

Though this removes the branching in the code, it actually performs significantly worse. The reason becomes clear after noticing two things. First, the introduction of significantly more instructions in this loop (as examined by using `objdump`):

```
216d: 04 7f                        add    $0x7f,%al
216f: 00 0d 0e 73 75 6d            add    %cl,0x6d75730e
2175: 2e 31 32                     xor    %esi,%cs:(%rdx
2178: 34 32                        xor    $0x32,%al
217a: 5f                           pop    %rdi
217b: 56                           push   %rsi
217c: 24 31                        and    $0x31,%al
217e: 65 00 0a                     add    %cl,%gs:(%rdx)
2181: 04 00                        add    $0x0,%al
2183: 04 00                        add    $0x0,%al
2185: 00 56 6e                     add    %dl,0x6e(%rsi)
2188: 82                           (bad)
2189: 88 00                        mov    %al,(%rax)
218b: 00 06                        add    %al,(%rsi)
218d: 04 7f                        add    $0x7f,%al
218f: 00 0e                        add    %cl,(%rsi)
2191: 0f 70 72 65 76               pshufw $0x76,0x65(%rd
2196: 2e 31 32                     xor    %esi,%cs:(%rdx
2199: 34 32                        xor    $0x32,%al
219b: 5f                           pop    %rdi
219c: 56                           push   %rsi
219d: 24 31                        and    $0x31,%al
219f: 66                           data16
21a0: 00 0b                        add    %cl,(%rbx)
21a2: 04 00                        add    $0x0,%al
21a4: 04 00                        add    $0x0,%al
21a6: 00 56 6e                     add    %dl,0x6e(%rsi)
```

```
21a9: 82                        (bad)
21aa: 88 00                     mov     %al,(%rax)
21ac: 00 06                     add     %al,(%rsi)
21ae: 04 7f                     add     $0x7f,%al
21b0: 01 4d 5f                  add     %ecx,0x5f(%rbp
21b3: 00 01                     add     %al,(%rcx)
21b5: 00 00                     add     %al,(%rax)
21b7: 00 02                     add     %al,(%rdx)
21b9: 4d 5f                     rex.WRB pop %r15
21bb: 01 02                     add     %eax,(%rdx)
```

The other element that must be observed here is that x86 and later have instructions for conditional move / set, for which the branch above actually reduces to a small set of simple instructions. That is, although in code we write a branch using an if statement, it gets reduced to a consistent number of instructions, since the logic inside of the branch is simple enough for the compiler to reduce it to a branchless set of instructions anyway.

2.2 OpenMP Scaling Study

In order to analyze the efficiency of parallelizing this shortest paths algorithm with OpenMP, a strong and weak scaling study was performed. This will be used alongside a scaling study for our MPI implementation, presented later, to determine the best method for parallelization.

First, we performed a strong scaling study to see the potential speedup that can be gained by using more processors. For this, we define the strong scaling as

$$\text{Strong Scaling} = \frac{t_{\text{serial}}}{t_{\text{parallel}}}\ , \tag{1}$$

and a strong scaling efficiency as

$$\text{Strong Scaling Efficiency} = \frac{t_{\text{serial}}}{t_{\text{parallel}}}\frac{1}{p} \tag{2}$$

where p is the number if processors. To enable easy comparison, we will use the strong scaling efficiency, which is normalized by a linear (ideal) speedup. The speedup of OpenMP derived from our study can be seen in Figure 1. While it is strange that the strong scaling efficiency rises over one for several points, this is possibly due to heterogeneity in the cluster. In order to quantify the deviation in time with constant parameters, the program should be run several times, however we have not had the time available to conduct this study. The OpenMP parallelization does seem to show constant scaling efficiency decrease, with no discontinuity between 8 and 16 processors, where threads from two different Xeon chips begin to be used.
A weak scaling was also performed in order to see the significance of communication for the OpenMP parallelized program. The parameters for this study can be seen in Figure 2, with each run of the program being done four times for probabilities of 0.025, 0.05, 0.10, and 0.30. The results of this study can be seen in Figure 2 where weak scaling is calculated as

$$\text{Weak Scaling} = \frac{t_{\text{serial}}(n(p))}{t_{\text{parallel}}(n(p), p)} \tag{3}$$

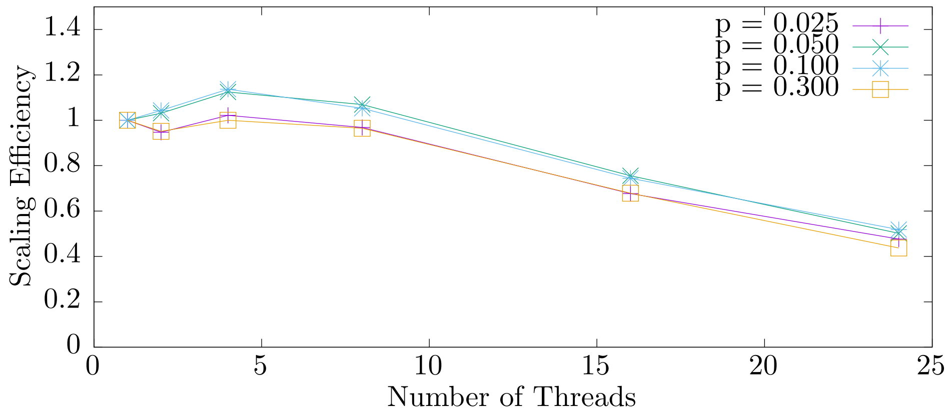

Figure 1: Strong scaling for optimized OpenMP parallelized path algorithm with `n` = 4000.

# of Threads	n
1	1000
2	1414
4	2000
8	2828
16	4000
24	4899

Table 2: Table of weak scaling parameters for number of threads and the square root of the problem size (n)

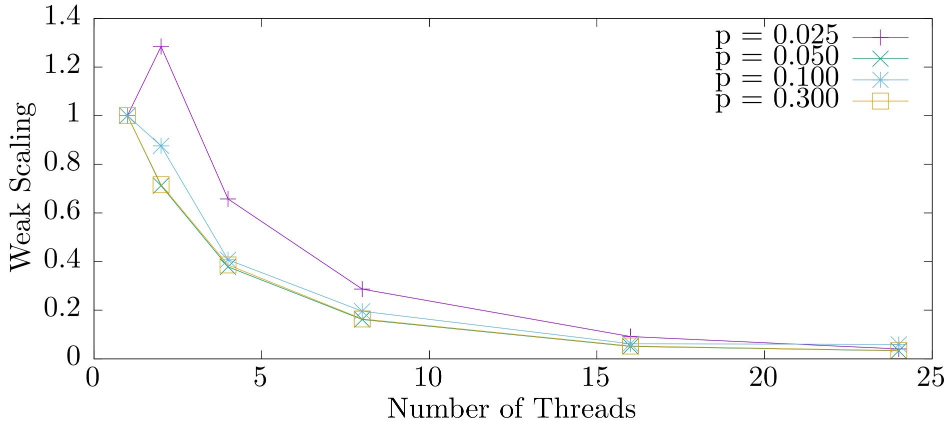

Figure 2: Weak scaling for optimized OpenMP parallelized path algorithm with program parameters given in Table 2.

From Figure 2, it appears that communication and parallel overhead is very costly in the OpenMP shortest paths algorithm. This is most likely due to the fact that the OpenMP team is launched each time inside the `square` function, which is called multiple times during each program execution. Once again, there is a point displaying super-linear weak scaling. This is due to the fact that there is 302 less nodes per processor, requiring less work per processor. Currently, we are unsure why this only occurs for one of the probabilities.

2.3 Algorithmic Optimizations

The most immediate and obvious form of algorithmic optimization for this assignment is to employ the `matmul-` style sub-blocking we used in the first assignment. This section will be detailed at a later time after we have incorporated this in our program. Since this technique is familiar at this point, we elected to focus our efforts on MPI and elsewhere for the initial report.

At this time we feel it only worth mentioning that arbitrarily sized sub-blocks are a desirable feature, and that when considering the dimensions of the sub-blocks to employ a simple enhancement can be to have rectangular blocks that are larger in the horizontal than in the vertical. This enables unit-stride access for a greater number of elements when doing the actual computation.

3 Message Passing Interface (MPI)

The method of MPI parallelization was chosen for its ease of implementation and decent performance. To start, the Floyd-Warshall pathing algorithm, originally parallelized using OpenMP, was changed in order to make use of MPI. First, the parallelization and delegation of work was moved outside of the square function and into the main function, in order to eliminate multiple parallel initialization costs. From this point, the pathing matrix is split into `npx` $\times$ `npy` subdomains, where `npx` and `npy` are the number of processors used to divide the matrix in the horizontal and vertical directions, respectively. `MPI_CART` commands are used to organize these threads in an optimal orientation and provide coordinates for the responsibilities of each processor, which are then used to determine the starting and ending indices of each thread's subdomain in relation to the global pathing matrix. At this point, each thread has its own copy of the entire pathing matrix and calls the `shortest_path` function.

The functions `infinitize` and `deinfinitize`, as well as the diagonal zeroing loop, were not parallelized due to their relatively little computational expense. In doing so, the associated parallel communication costs were avoided. The values of `l` are then copied into `lnew` using `MPI_ALLREDUCE` with the `MPI_MIN` operator. In this way, consistency between processors is ensured. It would also be possible to parallelize `infinitize` and `deinfinitize` with this method of communication, if the problem size became large enough to warrant the additional communication costs associated with it.

In the main loop of `shortest_path`, the `square` function is called by each thread, where each thread determines the shortest paths for its subdomain. Once each thread is finished inside square, `MPI_ALLREDUCE` is once again used with the `MPI_MIN` operator to determine if all processors returned a value of 1, indicating all shortest paths have been found. The array `lnew` is then communicated and stored in `l`, once again using `MPI_ALLREDUCE` with the `MPI_MIN` operator.

While this is not the most efficient MPI implementation, we believe it to be sufficient to show that shared memory OpenMP will be significantly faster than MPI for problem sizes that fit on one node. This is due to the nature of the pathing algorithm, where every point to the left, right, top, and bottom of the element of interest needs to be checked to determine if it provides a shorter path, making this more efficient with shared memory. If the problem sizes driven by an application scaled greatly so that they could no longer fit on one node, the MPI implementation could be rewritten to provide some marginal gains in performance and noticeable gains in memory efficiency, enabling the algorithm to be run on very large problem sizes.

3.1 MPI Scaling Study

In order to understand the performance of the MPI-parallelized shortest paths algorithm, strong and weak scaling studies were run. For the strong scaling, six simulations were run for a problem size of 4000×4000 for four separate probabilities: 0.025, 0.05, 0.1, 0.3. The results are plotted in Figure 3.

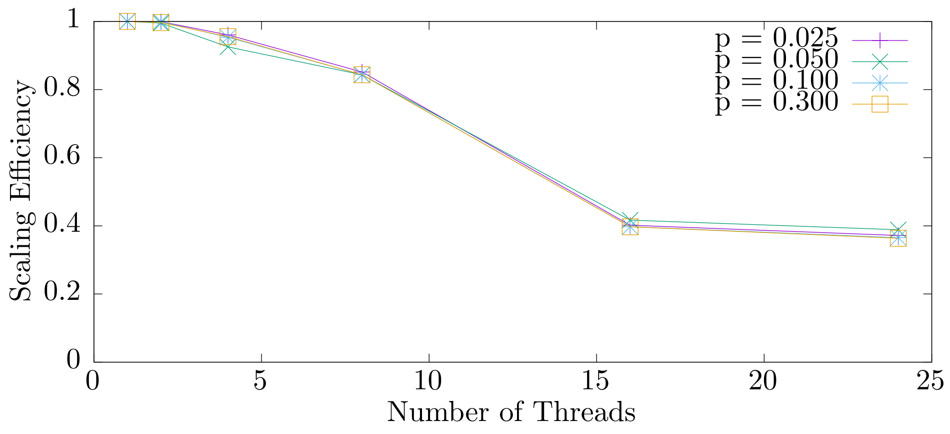

Figure 3: Strong scaling for MPI parallelized path algorithm with `n` = 4000.

A couple of comments can be made about the parallel efficiency shown in Figure 3. First, there is a stark decrease in scaling efficiency between 8 and 16 threads. This is most likely due to an increase in communication costs, as running the program with 8 threads only requires communication on the same Xeon chip, while 16 threads spans two Xeon chips. For all probabilities presented here, this actually leads to the program completing quicker when run with only 8 processors as opposed to 16. It is interesting to note that this discontinuity in scaling between 8 and 16 processors does not appear for OpenMP (see Figure 1), possibly due to optimized communications inside the OpenMP library itself. A second comment that can be made is that scaling efficiency appears to be independent of the probability parameter.

Typically, weak scaling is used to view the communication costs associated with parallelizing a program since the algorithmic work should remain constant per processor during the study. This should allow us to see if communications become a dominant source of time in our program, requiring us to rethink our MPI parallelization strategy. The parameters for our weak scaling study can be seen in Table 2, where each parameter configuration was one again run for probabilities of 0.025, 0.05, 0.1, and 0.3. The results of the study are shown in Figure 4.

From the weak scaling, it can once again be seen that large communication costs are associated with using threads that span two Xeon chips by comparing the weak scaling at 8 threads and 16 threads. The poor performance indicates that if we desire to use the MPI parallelization as anything more than a message passer for OpenMP thread teams running on Xeon Phi's, significant effort should

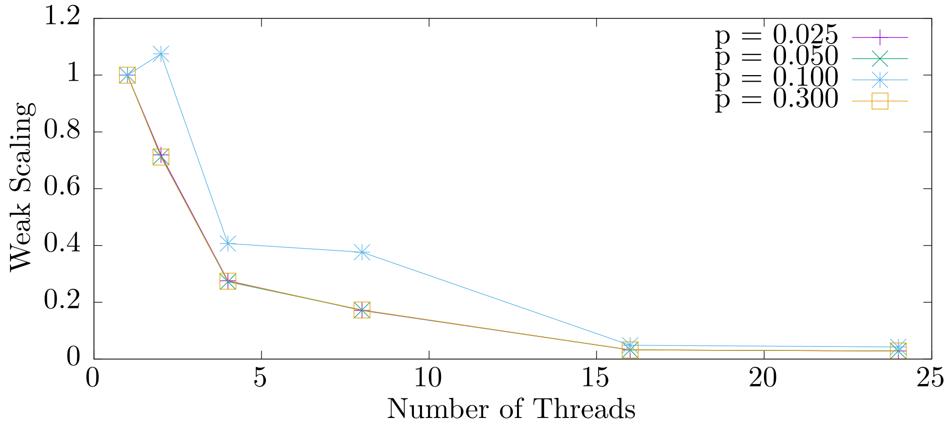

Figure 4: Weak scaling for MPI parallelized path algorithm with program parameters given in Table 2.

be spent on improving parallelization through reducing the communication frequency and the size of data communicated. The higher than one weak scaling for two threads with a probability of 0.1 is once again due to the fact that there is 302 less nodes in the system per processor. We are still unsure of the cause and do not yet have an explanation for why the super-linear scaling displayed in the MPI version is for a probability of 0.10, while for the OpenMP version it is super-linear for a probability of 0.025.

4 Post Initial Report Work

4.1 Blocking of the `square` function

In order to reduce cache misses and increase memory locality, the `square` function in the pathing algorithm was broken up into blocks in the same manner as used for the general matrix multiply assignment. In order to increase unit striding in memory, blocks moving in the horizontal direction across the matrix (`i`) are set to be longer than the blocks for vertical movement (`j`) and the search for shorter paths (`k`). Additionally, in an effort to take advantage of unit strides in memory, the loop indices were reordered to have horizontal movement (`i`) be the innermost loop, meaning only unit stride in memory is performed in the inner loop. The new `square` function appears as

```
// Major Blocks
for(int J = 0; J < n_height; ++J) {
  for(int K = 0; K < n_height; ++K) {
    for(int I = 0; I < n_width; ++I){
      // Calculate ending indices for the set of blocks
      int j_end   = ((J+1)*height_size < n ?
              height_size : (n-(J*height_size)));
      int k_end   = ((K+1)*height_size < n ?
               height_size : (n-(K*height_size)));
      int i_end   = ((I+1)*width_size  < n ?
               width_size  : (n-(I*width_size)));
      int j_init  = J*height_size*n;
```

```
          int kn_init = K*height_size*n;
          int k_init  = K*height_size;
          int i_init  = I*width_size;

          // Minor Blocks
          for (int j = 0; j < j_end; ++j) {
            int jn = j_init+j*n;

            for (int k = 0; k < k_end; ++k) {
              int kn  = kn_init+k*n;
              int lkj = l[jn+k_init+k];

              for (int i = 0; i < i_end; ++i) {
                int lij_ind = jn+i_init+i;
                int lij = lnew[lij_ind];
                int lik = l[kn+i_init+i];

                if (lik + lkj < lij) {
                  lij = lik+lkj;
                  lnew[lij_ind] = lij;
                  done = 0;
                }
```

Rearranging the loop structure in this way reduces the working set, allowing it to fit in the L2 cache, greatly reducing cache misses. Where possible, indices have been precomputed outside of the innermost loop in order to reduce the amount of floating point operations used in index calculation, such as `jn` and `kn`. Testing from Group 19's "matmul" report (Robert Chiodi's old group) was directly used here in determining the ideal block sizes. One major change, however, was each block size was doubled since in this case, integers are used, which are half the size of the doubles used in assignment one.

While naive implementation of general matrix multiply blocking structure may not be ideal for this pathing algorithm, almost an order of magnitude acceleration is achieved over a wide range of parallelization for a moderate problem size of $n = 4000$. This can be seen in Figure 5, where the blocked code remains an order of magnitude faster than the original optimized code presented in our preliminary report. This indicates that memory access issues were a major bottleneck in the original code, which were alleviated through reordering loop indices and performing the computation in block to force memory locality.

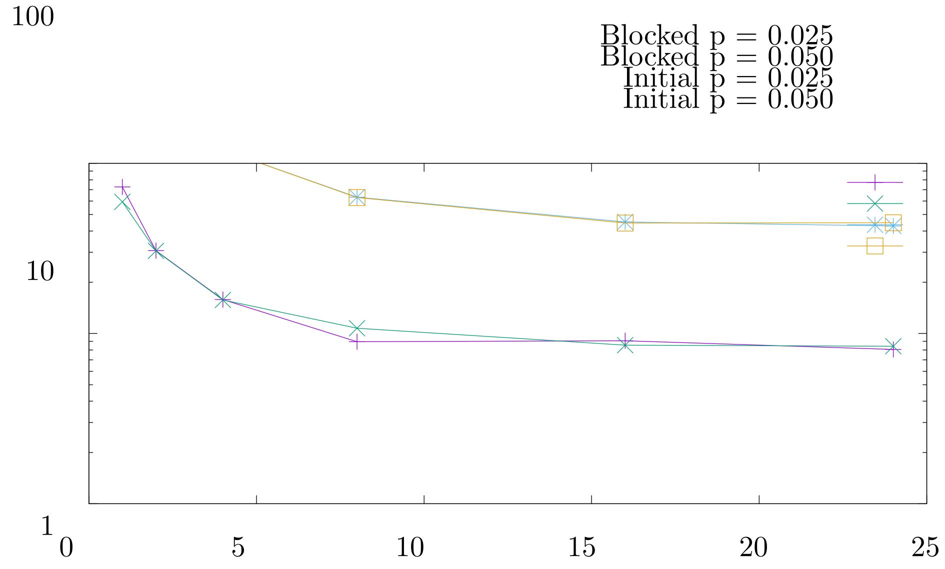

Figure 5: Comparison of runtimes for OpenMP with and without blocking. The problem size was `n` = 4000.

While the blocking provided significant performance gains over the previous, unblocked pathing algorithm, it was not expected that this would aid in the parallel efficiency of the OpenMP method. This was proven to be true when strong and weak scaling studies were performed, shown below in Figure 6 and 7, respectively. Once again, the strong scaling was performed using 4000 points on each side and probabilities of 0.025, 0.050, 0.10, and 0.30. The weak scaling was performed using the configurations given in Table 2.

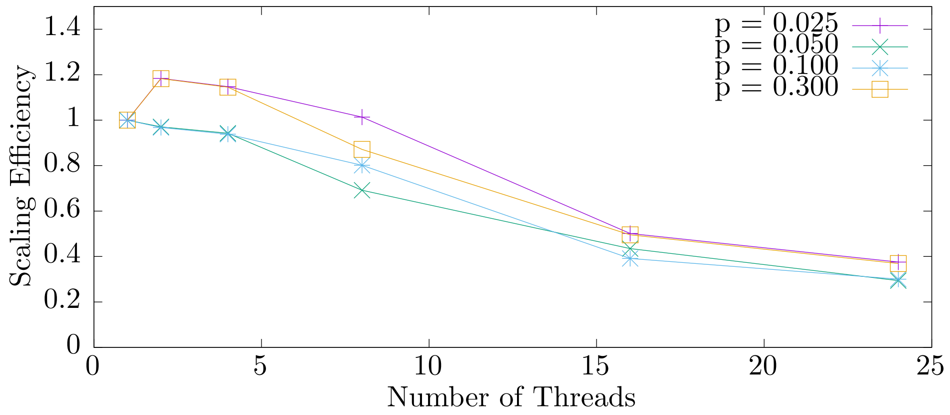

Figure 6: Strong scaling for OpenMP parallelized path algorithm with blocked submatrices. The problem size was `n` = 4000.

By comparing these figures to the original OpenMP scaling figures (1 and 2), it is obvious that blocking had no effect on the parallel efficiency. Once again, we are not sure why strong scaling efficiency displays values greater than one, however it is possible that this is due to node inhomo-

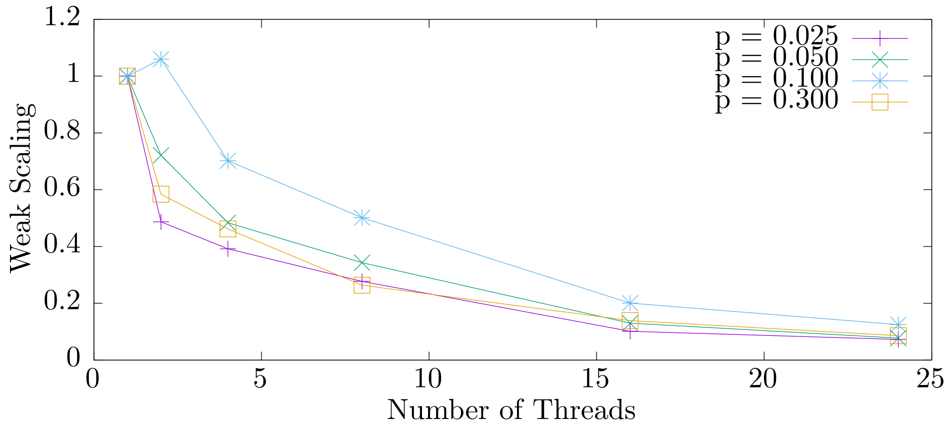

Figure 7: Weak scaling for OpenMP parallelized path algorithm with blocked submatrices. Program parameters are given in Table 2.

geneity on totient, since each job was not submitted to a specific node. For the weak scaling case, this greater than one scaling is due to the limited communication costs when using two threads relative to one, coupled by the fact that the problem is not exactly scaled, meaning with two threads, each thread has 302 less nodes compared to the serial case, as discussed before.

4.2 Offloading to the Phi's

4.2.1 Compilation Assistance

Since the Xeon Phi has a larger set of registers, it's preferred data alignment is 64 bytes. As such, the files that offload work to the Phi (`path-blocked-device.c` and `path-blocked-naive.c`) define the following compilation hints:

```
#define BYTE_ALIGN  64
#define width_size  512
#define height_size 128
```

Originally this was in a conditional `#ifdef __MIC__` region, but after careful thought and analysis we conclude that since no real work was intended for the Nodes in these files, the conditional definitions had little meaning, for the following reasons

1. When the code is compiled for the Node, the byte alignment will be 32.
2. When the code is compiled for the Device, the byte alignment will be 64.

This is because, where `#pragma offload target(mic)` and `__declspec(target(mic))` sections are concerned, the code is compiled in two passes by `icc` – once for the Node and once for the Device. Although the Node and Phi have different preferred alignments, the conditional definition violates what we assume to be true in the Phi code. You cannot allocate data on the Node with a 32 byte alignment and assume it is 64 byte aligned after things get transferred to the Phi.

Similar to before, to enable portability to environments where `icc` is not present, we create the following definitions

```
#ifdef __INTEL_COMPILER
    #define DEF_ALIGN(x) __declspec(align((x)))
    #define USE_ALIGN(var, align) __assume_aligned((var), (align));
    #define TARGET_MIC __declspec(target(mic))
#else // GCC
    #define DEF_ALIGN(x) __attribute__ ((aligned((x))))
    /* __builtin_assume_align is unreliabale... */
    #define USE_ALIGN(var, align) ((void)0)
    #define TARGET_MIC /* n/a */
#endif
```

Where the new addition is `TARGET_MIC`. This allows us to declare functions targeted for the Phi in the same way for both `icc` and `gcc`:

```
TARGET_MIC
void solve(int n,                    // Number of nodes
           int * restrict orig_l,    // Partial distance at step s
           ...
```

The last bit of information the compiler needs is the `include` and `define` statements at the top, which can be surrounded with the alternative approach for declaring Phi regions:

```
#pragma offload_attribute(push, target(mic))
#include <...>
...
#define ... ...
...
#pragma offload_attribute(pop)
```

Not all of the `include` statements necessarily need to be included, but this is simpler in that it enables more computation to be offloaded to the Phi in the future.

4.2.2 Offloading – A Naive First Approach

With the same blocking code presented before, our first approach to offloading was to send the `square` function we identified as being the bottleneck of the algorithm straight to the Phi:

```
#ifdef __INTEL_COMPILER
#pragma offload target(mic:0)                                    \
    in(n_threads)                                                \
    in(n)                                                        \
    in(n_width)                                                  \
    in(n_height)                                                 \
    inout(l    : length(n*n) alloc_if(first_iter) free_if(0)) \
    inout(lnew : length(n*n) alloc_if(first_iter) free_if(0))
#endif
    done = square(n, l, lnew, n_width, n_height, n_threads);
```

The `INTEL_COMPILER` condition was needed for `gcc` compilation, likely because of the line continuations. All this does is specify the parameters to `square`, noting that for both `l` and `lnew` we need those results copied back to the host in this naive approach.

The only optimization made here is recognizing that since this method is called potentially many times, we want to avoid allocating on the Phi each time. On the first iteration we specify that the memory needs to be allocated, and for all other iterations we can keep the pointers that were allocated in the first run. Correspondingly, though, it is important to free this memory outside of the `for(int done=0; !done;)` loop:

```
#ifdef __INTEL_COMPILER
    // free the phi memory used in the loop
    #pragma offload_transfer target(mic:0)   \
                    nocopy(l : free_if(1))   \
                    nocopy(lnew : free_if(1))
#endif
```

At this point we observe a key flaw in this approach. There is no need to continue copying over `l` and `lnew` every time that `square` is called. One option that would be simple to implement is to change the length specifiers

```
inout(l    : length(first_iter*n*n) ...
inout(lnew : length(first_iter*n*n) ...
```

This can be done because `first_iter` will be `1` on the first iteration, and `0` for all other iterations. The only extra piece of data that would be needed is a flag (say as a parameter to `square`) to indicate whether this was an even or an odd iteration. This will enable the usage of `l` and `lnew` on the Phi since they have already been allocated, but indicates to the dispatcher that no data needs to be copied. Outside of the loop, though, we would need to call `square` again to ask for the data to be transferred back, as well as free the memory. This could be achieved using another flag e.g. `int teardown`, and just check inside square if this bit is set. If it is, we would have two options:

i) Copy `orig_lnew` back into `orig_l` if necessary and return (avoiding the need to copy both back), or

ii) Simply return immediately, let both be copied back, and let the Node decide if `lnew` needs to be copied back into `l` as is currently done.

We present now a strong and weak scaling study demonstrating the results of this approach:

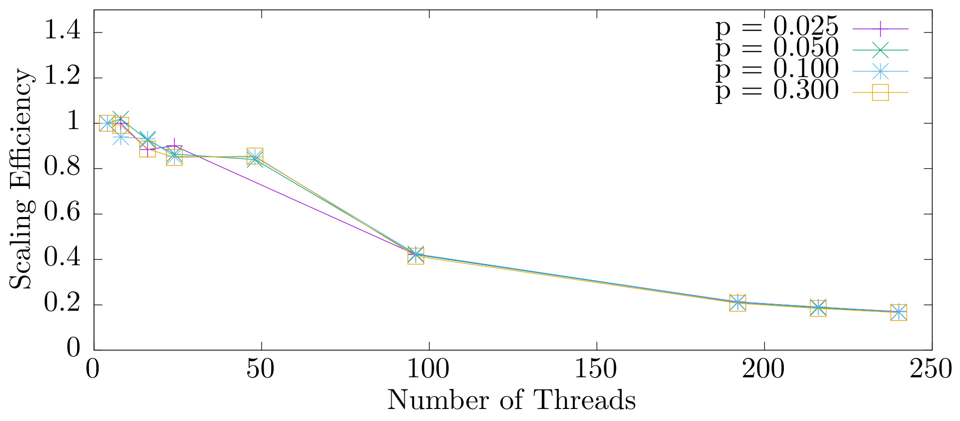

Figure 8: Strong scaling for OpenMP parallelized path algorithm with blocked submatrices computed on the Xeon Phi coprocessors. The problem size was `n` = 11111. We start at `threads`=4 because this is equivalent to using one core on the Phi, as well as the probability that the task will complete with using one thread on the Phi is very small given the roughly two times slower clock rates.

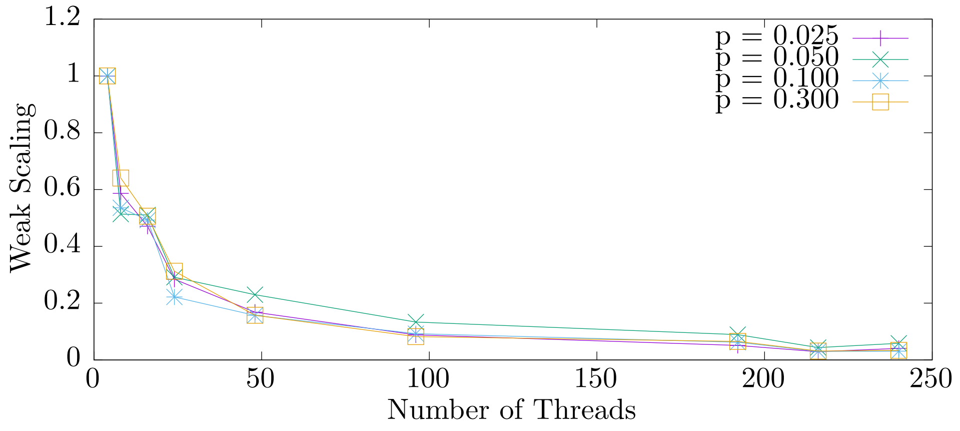

Figure 9: A weak scaling study for the OpenMP parallelized path algorithm with blocked submatrices computed on the Xeon Phi coprocessors. See Table 3 for the parameters.

4.2.3 Offloading to the Phi – A New Perspective

At this point, we decided that a more optimal implementation would be to avoid the Node driving the Phi computation altogether, and just use a single `offload target(mic)` statement that will not terminate until the results are in. The major difference being to basically take the `for(int done=0; !done;)` loop and put it in `square`. At this point the purpose of the method has changed enough that it warrants a renaming (to `solve`), with additional parameters.

# of Threads	n
4	2000
8	2828
16	4000
24	4899
48	6928
96	9797
192	13856
216	14696
240	15492

Table 3: Table of weak scaling parameters for number of threads and the square root of the problem size (n). We start at `threads`=4 because this is equivalent to using one core on the Phi, as well as the probability that the task will complete with using one thread on the Phi is very small given the roughly two times slower clock rates.

```
TARGET_MIC
void solve(int n,                     // Number of nodes
           int * restrict orig_l,     // Partial distance at step s
           int * restrict orig_lnew,  // Partial distance at step s+1
           int n_width,               // Width (x direction) of block
           int n_height,              // Height (y direction) of block
           int n_threads,             // how many threads to use
           int *copy_back) {          // copy back?
```

Where the new parameter of interest here is `int *copy_back`. This is just a single int that serves as a flag to the Node of whether the results of the computation ended on an even or an odd iteration, and therefore whether or not `lnew` needs to be copied back into `l` after the call to `solve` terminates inside of `shortest_paths`.

To properly explain `copy_back`, we will first explain the `solve` algorithm. The method essentially amounts to the following changes:

```
// since this is double buffering, we may have written the final results
// into the alternate orig_lnew, and therefore need to copy them
// back at the end
*copy_back = 0;

// buffered variables
int *l, *lnew;
size_t step = 0;

// while(!done)
for(/* step = 0 */; /* true */ ; ++step) {
    // setup double buffers
    int even = step % 2;
    if(even) {
        l    = orig_l;
        lnew = orig_lnew;
    }
    else {
        l    = orig_lnew;
        lnew = orig_l;
    }

    // Major Blocks
    int done = 1;
    #pragma omp parallel for
    ... same blocking code ...
    }// end Major Blocks (and omp parallel for)
    if(done) break;
}

*copy_back = step % 2 == 0;
```

In theory, it would be much simpler to just have `copy_back` be the return value of the `solve` function, but for whatever reason this would not compile. Passing it as a pointer was a cheap hack to work around this issue.

So all of this turns into the statement that if `solve` had an even number of iterations, this means that the results were written into `lnew` – and need to be copied back, and if there were an odd number of iterations then the results were written into `l` – and do not need to be copied back.

Then in the `shortest_paths` method, we can remove the `for(int done=0; !done;)` loop and instead just call one time

```
// the phi code is double buffered; may need to copy back after
int copy_back = -1;
int *cb = &copy_back;

#ifdef __INTEL_COMPILER
#pragma offload target(mic:0)                                    \
        in(n_threads)                                            \
        in(n)                                                    \
        in(n_width)                                              \
        in(n_height)                                             \
        inout(cb   : length(1)   alloc_if(1) free_if(1)) \
        inout(l    : length(n*n) alloc_if(1) free_if(1)) \
        inout(lnew : length(n*n) alloc_if(1) free_if(1))
#endif
solve(n, l, lnew, n_width, n_height, n_threads, cb);

if(copy_back)
    memcpy(l, lnew, n*n * sizeof(int));
```

At this point we must discuss the prior versions of this code. Originally we had decided it would be preferrable to copy less data, and do everything in-house on the Phi

```
DEF_ALIGN(BYTE_ALIGN)
int * restrict orig_lnew = (int * restrict)_mm_malloc(n*n * sizeof(int),
                                                          BYTE_ALIGN);
USE_ALIGN(orig_lnew, BYTE_ALIGN);

// initial conditions
memcpy(orig_lnew, orig_l, n*n * sizeof(int));

// keep track of who is done and who is not (manual reduction)
int *even_done = (int *)alloca(n_threads*sizeof(int));
int *odd_done  = (int *)alloca(n_threads*sizeof(int));

int copy_back = 0;

// avoid re-spawning threads for every iteration
#pragma omp parallel           \
        num_threads(n_threads) \
        shared(orig_l, orig_lnew, even_done, odd_done)
{
    // buffered variables
    int *l, *lnew, *done;

    // while(!done)
    for(size_t step = 0; /* true */ ; ++step) {

        // setup double buffers
        int even = step % 2;
        if(even) {
            done = even_done;
            l    = orig_l;
            lnew = orig_lnew;
        }
        else {
            done = odd_done;
            l    = orig_lnew;
            lnew = orig_l;
        }

        // assume the best
        int idx = omp_get_thread_num();
        done[idx] = 1;

        // Major Blocks
        #pragma omp for
        ... same blocking code ...
      }// end omp for

        #pragma omp barrier
        // if any thread is not finished, then all threads continue
        int finished = 1;
        for(int i = 0; i < n_threads; ++i)
            finished = finished && done[i];

        if(finished) break;

    }

    // not strictly necessary
    #pragma omp single nowait
    copy_back = lnew == orig_lnew;
}// end omp parallel

if(copy_back)
    memcpy(orig_l, orig_lnew, n*n * sizeof(int));

_mm_free(orig_lnew);
```

In theory, this implementation should be greatly improved because we avoid the need to spawn a new thread pool every time. Threads can independently perform their tasks, and sync up at the end with a manual reduction to determine if they need to continue. In practice, though, this implementation was almost twice as slow.

After the preliminary results, we decided that the likely cause was the creation of `orig_lnew` on the device. So we put this initialization back into the host code and did the same trick with `copy_back` mentioned previously. This gave an increase, but not anything substantial.

The next thought was that the manual reduction was the problem, and indeed after reverting back to the thread spawn for every iteration we saw an almost 200% increase in speed in comparison to the manual reduction.

Somewhat shockingly, though, the implementation that lets the Phi just keep spinning until it is done without the need of the Node to drive it performs almost twice as slow as the naive implementation presented above. As such, we decide not to include any strong or weak scaling studies of the above code. We are uncertain as to the cause of this, and are rather perplexed by the slowness of the above implementation.

4.3 Phi vs Node Comparison

As a final comparison, we present the results of the naive Phi computation juxtaposed with the Node computation in Figure 10. As you can plainly see, our efforts to offload computation to the Phi have been rather ineffective. We do not feel that this is due to lack of effort, or even improper offloading methodology. Rather, it is clear that the level of vectorization supplied by the compiler alone is not sufficient to take full advantage of the Phi's extra-wide vectors. Given that it has a clock rate of roughly half the Node, in conjunction with our plotted Phi results being wasteful in the amount of memory being transferred back and forth on each iteration, we feel that the plotted results can be explained by two simple factors:

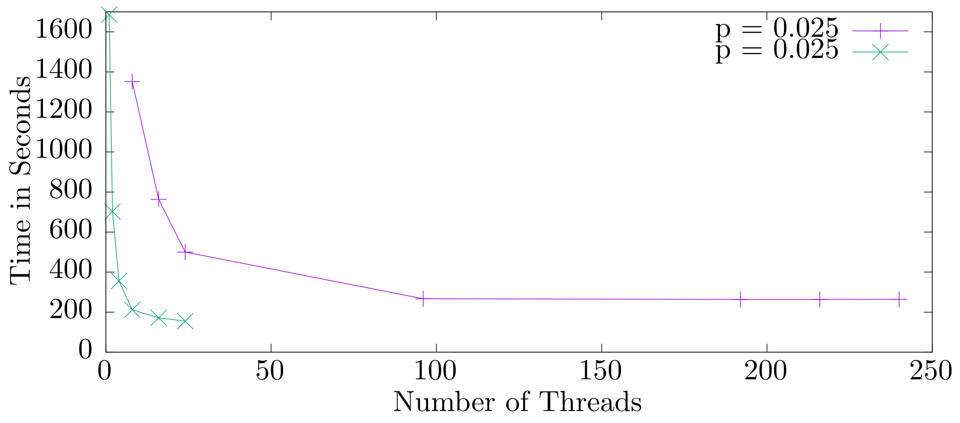

Figure 10: A comparison of the strong scaling for OpenMP parallelized path algorithm with blocked submatrices on the Xeon Phi Coprocessors and the Xeon Nodes. The problem size was `n` = 11111, and the probability was $p = 0.025$.

1. Lack of proper vectorization, and
2. Improper / unnecessary data transfer (e.g. why the line is almost flat – our task is memory bound)

The only other explanation we see as reasonable is that our `matmul-` style blocking incorporated in the Phi code likely does not wield the amount of threads available to us. That is, were there more time, we would also experiment with doing smaller blocks / sub-blocks to see how the performance changes.

4.4 Conclusion

Over the course of this project, we have implemented an MPI parallelization scheme, as well as tuned the initial OpenMP code. After seeing the superior performance of OpenMP due to its use of shared memory, we focused our efforts on tuning the OpenMP version of the code. This was mainly done through the implementation of blocking in the `square` function, which increased cache locality and sped the code up by an order of magnitude for the moderate case of $n = 4000$. We then focused on offloading the computation to the Xeon Phi co-processors, however were not able to effectively use them due to large amount of memory transfers, which have a significant amount of time associated with them. While our improvements did not have a significant impact on strong or weak scaling, parallelization alongside tuning has led to significant speed up of the shortest path algorithm and a substantial increase in performance.