Assignment 5 (MPI and OMP)

Daniel Wisner (daw268)

5/3/17

Implementation

The 2D Riemann sum algorithm was simple to parallelize using domain partitioning techniques. Each node in the MPI cluster integrates across a portion of the domain, since the integral over one subset of the domain is independent of the integral over another subset of the domain, these operations can be performed in parallel. The algorithm with an MPI cluster with N nodes, each with indices i, divide up the integral as follows:

$$\int_{-r}^{r}\int_{-r}^{r} f(x)dydx \quad = \quad \sum_{i=0}^{N-1} \int_{-r+2ri/N}^{-r+2r(i+1)/N} \int_{-r}^{r} f(x)dydx$$

At the end each node's computation, the result over the entire region is simply the sum of all the integrals. This is accomplished with one simple call to MPI_Reduce() with the sum reduction operator.

Additionally, since each MPI node has many cores, further parallelization was accomplished at the node level using OMP with a similar strategy of subdividing the domain into further subdomains. In this particular implementation, each thread of a node again split the the x domain by the number of threads per node:

```
# pragma omp parallel for schedule(guided) reduction(+: res)
for (uint32_t i=0; i < steps_x; i++)
{
    for (uint32_t j=0; j < steps_y; j++)
    {
        // always recalculate from index i and j to avoid precision errors
        double x = xstart + i*dx;
        double y = ystart + j*dx;

        res += f(x,y,args) + f(x+dx,y,args) + f(x,y+dy,args) + f(x+dx,y+dy,args);
    }
}
```

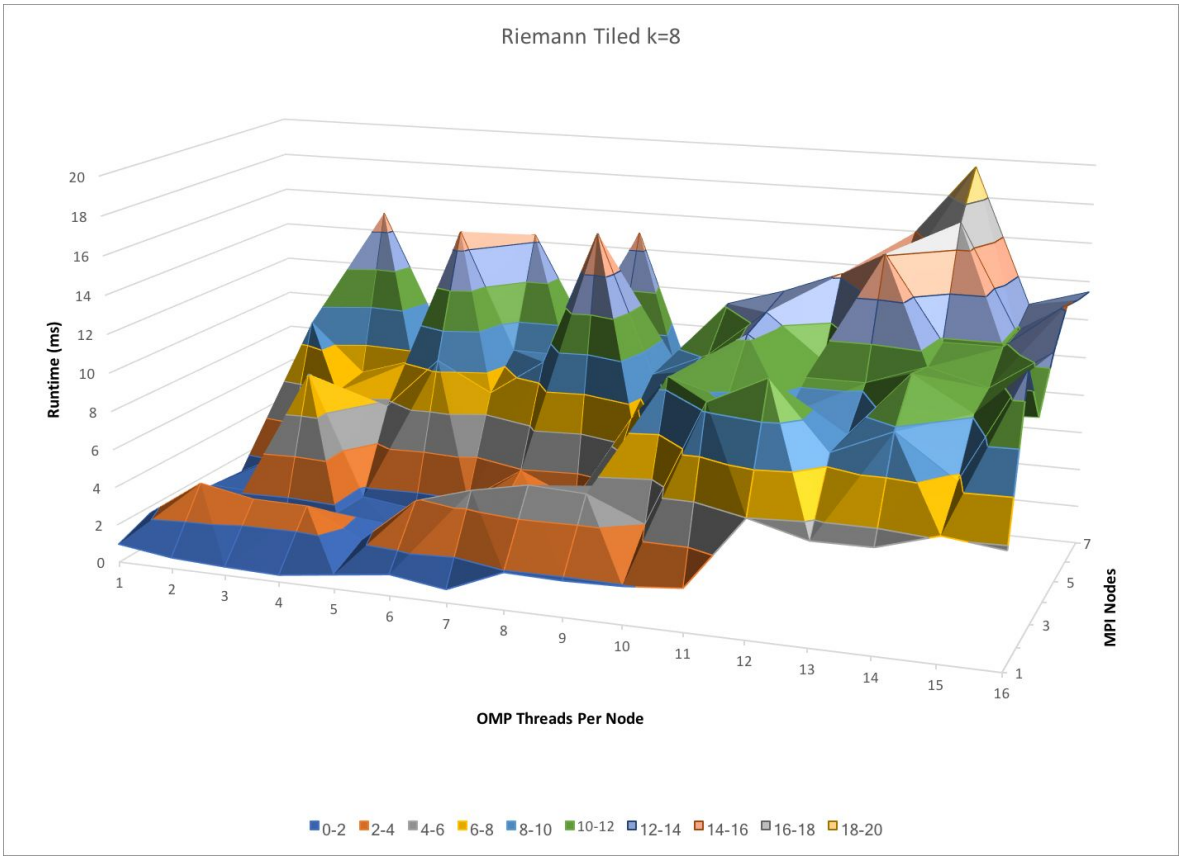
Here the for loop doing the integration using OMP has each thread split up the for loop that iterates through range of x. Each thread gets a private copy of res which is reduced at the end of the loop by summation. After testing, it appeared a guided schedule yielded the best results as a static schedule tended to result in threads finishing early and having to wait.

For the purpose of making reusable code that can easily be adopted to integrate other rectangular regions, the function performing the 2D integration on each node (riemann_rectangle()) takes a function pointer to the f(x,y) being integrated allowing it to integrate any rectangular region simply by changing the pointer to the function it is integrating: `double (*f)(double, double, void*);`. For this reason, no optimizations to the integration algorithm were made based on assumptions on the type of function being integrated (in this case a spherical surface). For example, using symmetry only a single quadrant would have to be integrated then multiplied by four. Additionally, for the same reason, the algorithm did not take advantage that the function was defined as zero outside of a radius r from the origin.

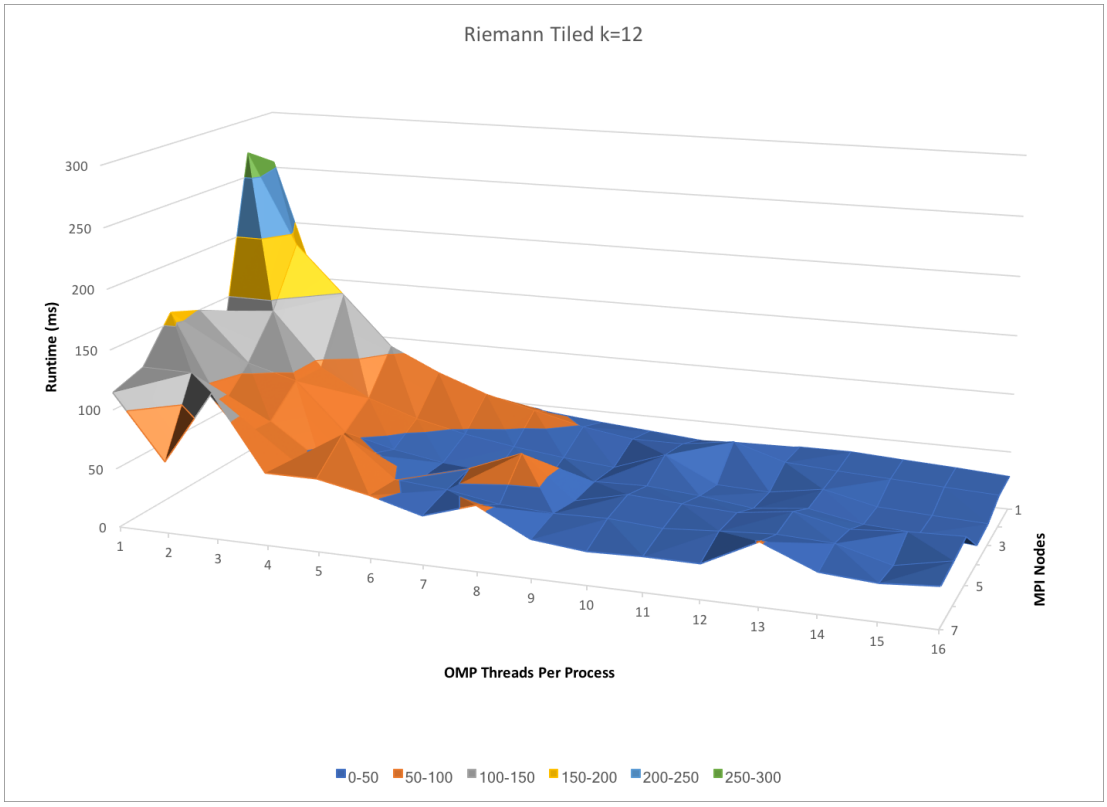
Testing/Results

The testing was performed on the MPI cluster, the bash script (test.sh) benchmarked the computation times with varying numbers of nodes, OMP threads per node, and step size (2^k steps in each direction). For these particular set of tests r was set to 1:

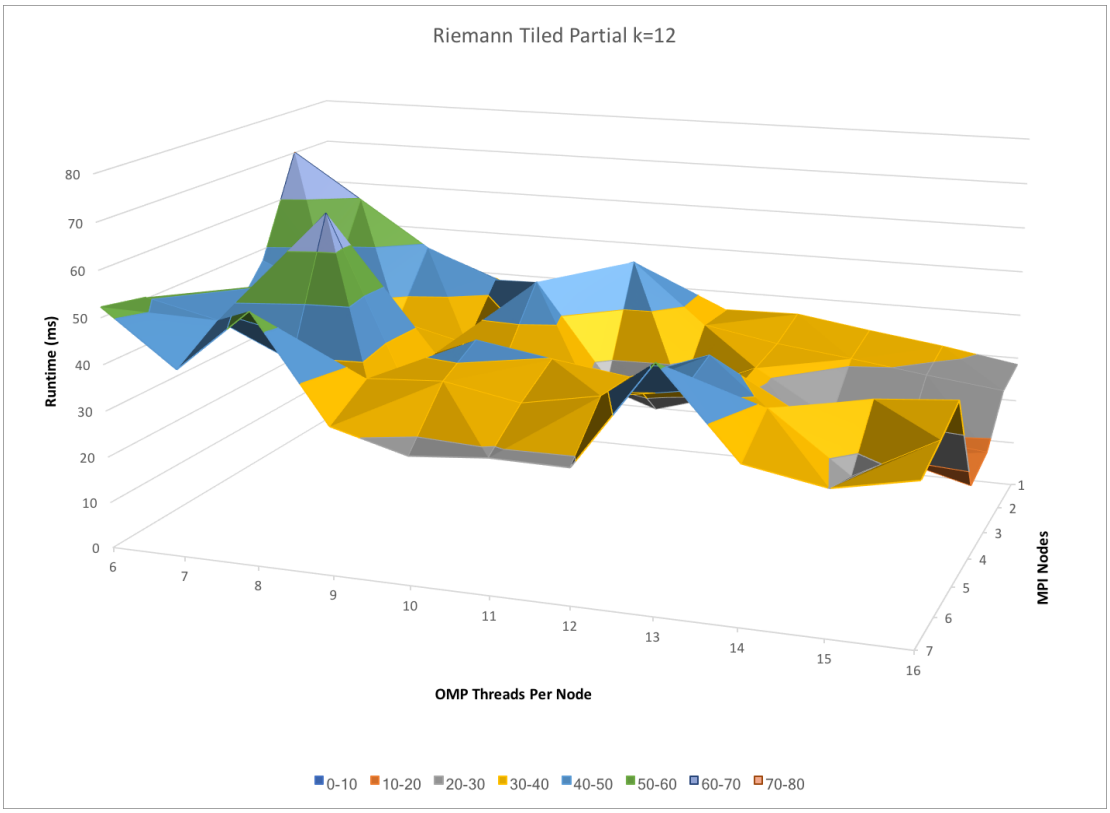
k=8



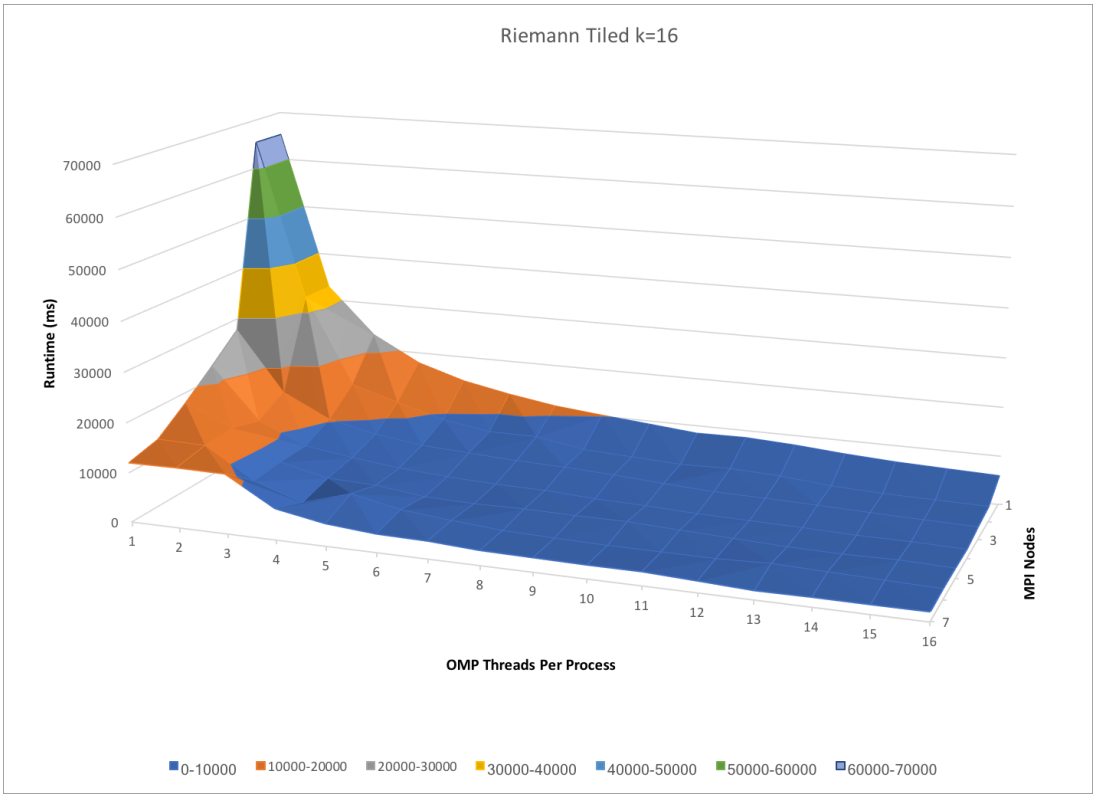
k=12



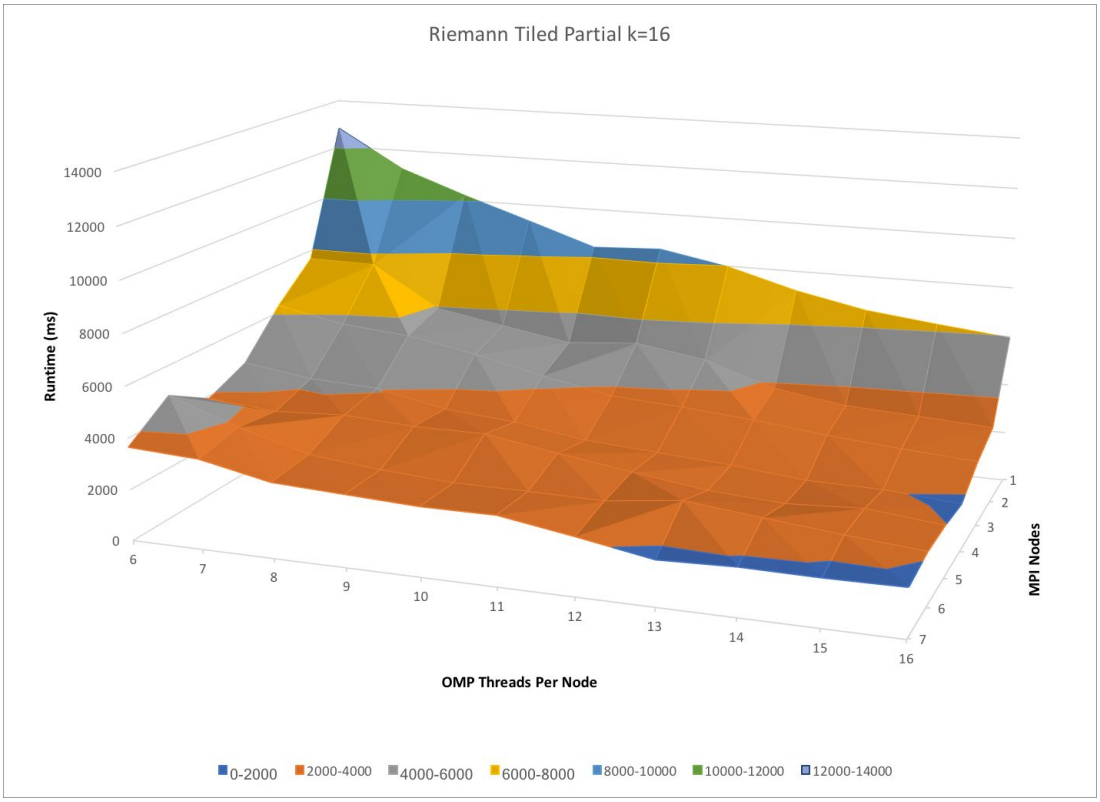
Same plot zoomed in for OMP Threads > 5



k=16



Same plot zoomed in for OMP Threads > 5



Conclusion

For all but the smallest k sizes, parallelizing provided a major speedup over the sequential version. Even for the k=8 test, multithreading with OMP provided a speedup over the serial version up till roughly 4 OMP threads.

For larger k's, it became clear that performance scaling with additional OMP threads per node was fantastic, as for all k's > 8, additional OMP threads always resulted in a speedup as long as the number of threads did not exceed the number of cores on a particular machine. Thus, based on the data collected, it is clear that in general for best performance the number of OMP threads per node should be set to the number of cores of that particular node. On a more surprising note, the benchmark was also run on a hyperthreaded quad core Intel desktop computer running as a single MPI node, and speedups due to OMP continued to occur up till 8 OMP threads. This is surprising, given the machine only had 4 hardware cores, and the speedups must of been coming from Intel's hyperthreading, which rarely seems to provide any speedup in parallel applications.

The question of the optimal number of MPI nodes in the cluster for best performance had a much less obvious answer that determining the optimal number of OMP threads and depended strongly on k (number of steps in integration). For small k's the communication overhead completely degraded performance resulting in substantial slowdowns as seen in the k=8 case. For small k's best performance will be achieved with a single node. For more medium valued k's, the answer to this question was the most unclear. For the k=12 case, additional nodes offered moderate to substantial speedups up until the the number of nodes exceeded roughly 4. After this point, performance with additional nodes seemed to level off or get worse. For large k's, adding additional nodes to the MPI cluster in almost all cases offered a speedup over fewer nodes. In the k=16 case, unsurprisingly the best performance occurred when all 7 available nodes in the cluster were utilized.

Building and Running the Benchmark

Compiling: mpicc -std=gnu99 -O3 -Wall -fopenmp daw268_hw5.c -lm -o daw268_hw5

Running: mpirun -mca plm_rsh_no_tree_spawn 1 -np **<N>** -use-hwthread-cpus -hostfile hostfile --map-by node:PE=**<X>** daw268_hw5 **<X> <r> <k_start> <k_end>**

N = number of nodes (processes)
X = number of omp threads per process
r = is the radius of the hemisphere
k_start = Starting number of steps (2^{k_start}) while integrating
k_end = Ending number of steps (2^{k_end}) while integrating

The results of the benchmark will be placed in the data folder as a .csv file.

The benchmark can also be run by running the “test.sh” bash script which will run the benchmark with a range of nodes, omp threads per node, and k’s. Again the results of each test will be saved in the data folder as .csv files.

Macros

- **NUM_RUNS** - The number of averages for each run