Sam Warring, 903943125
Steven La, 503929466
Zach North, 603885768

TEAM 18: CS 133 PROJECT REPORT

Our preferred library for writing parallel programs is OpenMP. With OpenMP, our implementations did little more than parallelize for loops to achieve impressive speedups, all while keeping the original code intact and readable. OpenMP also benefits from automatically performing much of the thread communication for the programmer. Even though OpenMP lacks the expressive power for task-parallelism (as opposed to a framework like CnC), the support for data-parallelism with for loops is enough to significantly improve performance for data-parallel algorithms.

Our second favorite library for parallel programs is OpenCL. Like OpenMP, the API is designed for massively data-parallel applications like ours. All threads execute the same instructions and differ only by their work indices. Global data allows a single matrix to be easily shared among threads. Unlike OpenMP, it OpenCL provides support for heterogeneous systems which means it has the potential to execute with tremendous speed on special purpose hardware like GPUs. It also means we had to spend more time setting up the OpenCL environment (getting devices, compiling kernels, etc.). Since kernels were executed on OpenCL devices, we had little support for debugging our code. Because of these reasons, we prefer OpenMP to OpenCL.

Our least favorite library for parallel programs is MPI. Compared to OpenMP, the MPI API allows for more detailed inter-process control, but this typically made the code difficult to debug. Furthermore, our MPI implementations contained only a few parallel regions, which means each process had to repeat much of the same work. Because of this, MPI may be more suited for distributed programs with more costly process-to-process communication. Since MPI creates workers as separate processes, scattering and gathering an array involved copying the data from one address space to another. Our applications required many scatters and gathers, so an OpenMP or OpenCL implementation with a shared memory model is more appropriate.

MRI RECONSTRUCTION PROJECT WRITE-UP

Sam Warring

My initial profiling of the MRI reconstruction code revealed the program spent most of its time in the dft and idft functions. Each of these functions worked the same way. A first set of nested for loops sets a 'tmp' matrix from the 'src' matrix, and a second set of nested for loops sets a 'dst' matrix from the 'tmp.' Every cell in 'tmp' depends on the all cells of the same row in 'src.' Every cell in 'dst' depends on all cells of the same column in 'tmp.'

The OpenMP implementation parallelized these two outermost loops for each function and this resulted in a significant speedup (14x on 16 cores). I parallelized the outermost loops to maximize the work done by each thread. The times can be found in /src/mri/omp/mri.omp.txt.

Since the MPI implementation spawns a separate process for each worker, communications involved transferring data from one address space to another. Compared to a shared-memory model like OpenMP, I had to explicitly partition the data and keep communication to a minimum. My plan was to have each worker produce a contiguous section of the 'dst' matrix then have them all gather together. Unfortunately, if each worker were to produce rows of the 'dst' matrix, it would require knowing the entire 'tmp' matrix. If each worker were to produce a columns of the 'dst' matrix, then each worker only required a portion of the 'tmp' matrix, but they could not be efficiently joined with a single gather operation. I solved this by having each worker produce columns of 'dst' and store them into the transpose of 'dst.' The transpose matrix could be easily gathered since each worker contributed a set of consecutive rows. When each worker received the gathered matrix, it performed a transpose locally to produce the true 'dst' matrix. The execution times for the MPI implementation are given in /src/mri/mpi/mri.mpi.txt.

My OpenCL implementation was very similar to my OpenMP implementation. I created one global buffer each for the 'src,' 'tmp,' and 'dst' matrices. A call to dft or idft wrote to the 'src' buffer, executed a kernel for each set of nested for loops, and read from the 'dst' buffer. Each kernel was executed with a 2-dimensional global work size of N-by-M where each work item produced a single cell in the resulting matrix. Using only these strategies, I achieved a significant speedup as seen in /src/mri/ocl/mri.ocl.txt.

DENOISE RECONSTRUCTION PROJECT WRITE-UP

by Zach North

Initial profiling indicated that the function spent almost all of its time in the main loop in the "denoise" function. This was expected, as it comprised the bulk of the project code. Parallelizing the calculation loop was my chief concern, as it handled the bulk of the requests and dealt with the most memory management, typically the bottleneck in a system like this.

My three implementations are explained in detail in their respective folders -- /src/den/mpi/den.mpi.txt, /src/den/ocl/den.ocl.txt, and /src/den/omp/den.omp.txt.
Note that my OpenCL implementation does not produce entirely correct results, but the other two are entirely correct, with varying amounts of speedup (I found it often depended heavily on the server load at the time.)

SEGMENTATION PROJECT WRITE-UP

by Steven La

All profiling showed that more than 99% of the time was spent in the seg() function, obviously. A lot of floating point operations were happening each iteration. A lot of the code could be done in parallel, since with image processing we could just split up an image and process the individual sections of the image separately. seg() first generates N*M values for phi and curv matrices. This is required for future computations. Each iteration of the loop computes for every N*M pixel in the image a value for curv, which is then used to compute values for each phi pixel. For all but the edge pixels, computing phi is only dependent on the value of curv on the same pixel location.

The OpenMP implementation was actually the best one. My idea was to put #pragma omp parallel for in as many places as possible without slowing down or messing up the output. This turned out to be really effective since the library takes care of all communication issues for me.

Details: */src/seg/omp/seg.omp.txt*

My MPI implementation was more involved, in that each thread manually had a sliver of the image to process and had to communicate with the other threads to compute new values for the edge pixels. This ended up being slower, especially for small images.

Details: */src/seg/mpi/seg.mpi.txt*

The OpenCL implementation operated on a pixel level. The buffers were shared amongst all threads. This allowed for easy speedup on large files.

Details: */src/seg/ocl/seg.ocl.txt*