High Performance Computing Lab
Project Documentation

Gaurav Kukreja, Evangelos Drossos

January 20, 2014

Abstract

This report is a documentation for the final project in the Lab Course on High Performance Computing. The project was to optimize the Shock Wave Equations code, which is used for simulations of scenarios like Tsunami and Dam Break. This report discusses the various optimizations strategies used in the project. We discuss the motivation and the performance impact from our approach.

1 Instrumentation using Scalasca

Scalasca is a tool which helps to benchmark parallel softwares. We have instrumented the code using Scalasca, to get a better understanding about the Bottleneck issues and analyze the performance of the existing code. We have also used Scalasca to understand and represent the impact of our optimizations to the code.

...

2 Intel Cilk to Vectorize the FWaveVec Solver

Initially, we had decided to use Intel Compiler Intrinsics to improve the vectorization of the stencil operations performed in the FWave Solver for the SWE Equations. Later we decided to use Intel Cilk Array Extensions instead. Code written using Intel Cilk Array Extensions is more readable and easier to modify and extend. Additionally, the compiler should be able to extract optimized vectorization strategy from the modified code.

In our implementation we process chunks of a row of the matrix in one iteration. The operations on the elements are performed using Intel Cilk Array Extensions. By varying the number of elements that are processed in one iteration, we can choose the most optimal size, where the compiler uses most of the available vector registers and performs computations optimally. From our tests we found that a block size of 8 elements is the most optimal.

Though, we expected a vast improvement in performance, the actual improvement achieved was TODO percent. Our understanding is that the existing code was already being vectorized by the compiler, and our modifications have slightly improved the peformance.

3 OpenMP Optimization in Wave Propogation Block

The existing code using OpenMP for basic parallelization. We made a few modifications to make the code compile. We made minor modifications to the code to gain slight improvement in performance. We fused the loops inside *computeNumericalFluxes()* function, to gain another slight improvement. Our intention is to run OpenMP threads inside a single node, hence there is no need to perform Cache Optimizations as the L2 Cache is shared cores on one node.

4 MPI Optimizations

The motive behind using MPI is to achieve a Hybrid Implementation, which allows us to use a large number of nodes in the Linux Cluster. Our intention is to split the problem size between multiple nodes, which will compute in parallel using OpenMP on the 16 cores in a node. MPI provides control and flexibility to ensure that the communication between threads running on different cores can be optimized.

After fixing a few compilation errors with the existing MPI Code, we optimized the MPI code by overlapping the communication with computation. For each iteration, the MPI threads need to receive the values for the border edges calculated by the neighboring threads. These border values are exchanged using blocking MPI_Sendrecv primitive. We changed the implementation to use non-blocking MPI_Isend and MPI_Irecv primitives. Each thread issues request to send its borders to its neighbors and receive borders from its neighbors. Instead of waiting for the transfer to complete, it can proceed with the computation of the next iteration for the inner blocks which do not depend on the border values.

We created two versions of *computeNetUpdates()* function, *computeNetUpdates_innerBlock()* and *computeNetUpdates_borders()*. The *computeNetUpdates_innerBlock()* is called after send and receive requests are issued. While the computation proceeds, the transfer of data is completed. We check the status of the transfer requests, and then call the *computeNetUpdates_borders()* function to compute the values for the borders.

Again, only a slight improvement in performance is achieved. This is because the time spent in communication is comparitively much smaller than the time spent in computation.

5 Cache Blocking

We could not get time to invest in achieving Cache Blocking. However, based on our analysis, we believe Cache Blocking can provide only a slight improvement in performance. If we perform the updates for Horizontal and Vertical Edges in a loop blocking fashion, we will be able to reuse elements that are loaded into the L2 cache.

In other applications like Matrix Multiplication, cache blocking can significantly improve the performance because the block size is directly proportional to the Computational Intensity of the code. In this application, the Computational Intensity of the code is quite high, and Cache Blocking will only slightly improve it. Additionally, the impact of Cache Blocking will only be visible at really large problem sizes.

6 Conclusion

From this project, we have learned how various parallelization paradigms can be utilized in a hybrid implementation to achieve the best performance from the hardware.