Pseudo-Applications – Computational Fluid Dynamics

The following benchmarks are three different implementations of scripts that solve the Navier-Stokes equations. Given that the objective of these three benchmarks are the same, they share many similarities in their implementations; naturally, they also share many of the same speedup limitations. As such, their speedup limitations of one of the pseudo-programs can be extended to explain that of the other pseduo-programs without loss of generality. It should be noted that the equations and objects used to solve the Navier-Stokes equations are not listed here due to size and quantity. To see these equations, they can be found on the NAS Parallel Benchmark Technical Report[1] The code that will be referenced can be found in our source code of the final project package submitted prior.

BT: Block Tri-Diagonal Solver, solving Block-Tridiagonal systems of 5x5 blocks in a sequential manner along each dimension

SP: Scalar Penta-Diagonal Solver, solving Scalar Penta-Diagonal bands of linear equations sequentially along each dimension

LU: Lower-Upper Gauss-Seidel Solver, uses symmetric successive over-relatxation (SSOR) method to solve a 7-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equation in 3D by splitting it into block Lower and Upper triangular systems

Below are two graphs: the speedup of each benchmark as a function of dataset size; the speedup of individual subroutines of the benchmarks of a Class C dataset:

<Insert Graphs Here>

There are several limiting factors to the speedup of this benchmark, mainly due to the rhs\* and \*solve subroutines (where \* = x,y,z). It should be noted that the nomenclature of the subroutines are derived from the NAS Parallel Benchmark Technical Report and thus are named in uniform manner:

1) There are high requirements for solving the Navier-Stokes equations. Since we are trying to solve the Navier-Stokes equations in 3D, a matrix cube with 100 entries on each side equates to having a matrix cube of 10^6 elements. Since double precision floating point numbers are 8B each, a matrix cube of length 100 occupies 8MB of space. Given that the shared L3 cache of the Linux Ubuntu is only 6MB large, it is impossible to store the entire matrix. As a the number of threads increase, the shared L3 cache will be a source of capacity misses in the cache; inevitably, the performance speedup will suffer. Note that the above explanation does not take into account the synchronization instructions that occur when two processors are reading/writing to the same memory location, nor does the dataset size take into account the auxillary matrices used in the intermediate calculations

2) The equations which defined the individual terms in a matrix were very long, often with more than 10 operands. While Fortran allows an variable to be defined as a long sequence of algebraic operations, the assembler will see only a sequence of instructions with at most two operands. This decomposition of a long Fortran variable assignment to a sequence of x86 instructions will inevitably lead to a true-data dependence. Given that the operands can be any algebraic operation (+,-,\*,/), they will inevitably lead to unpredictable true data dependence. Moreover, since there are many algebraic operations, there may also be structural hazards as the Linux Ubuntu processor has 4 ALU per core.

3) The code for the benchmarks took advantage of the column-major order array storage scheme native to Fortran, but did not always take advantage of loop-blocking. Moreover, there were instances when the memory accesss to the array elements were of non-unit stride. This increased the number of conflict misses that occurred in the cache.

4) In the rhs\* subroutines, there are true data dependencies between variable assignments

Conclusion

Amdhal's law dictates that the speedup between an optimized task and an unoptimized task is affected by the speedup of the we gain from improving the execution time of some part of a task. Nonetheless, we saw speedup from the benchmark tests to be sublinear to the increase in the number of cores used; at times, we saw a decrease in performance with an increase in the number of cores used. From each benchmark, we found different sources of bottlenecks, which range from data dependencies, to cache misses, to the computational overhead created in timing the benchmarks themselves. For applications which are not data-intensive, a speedup was observed when a benchmark ran with more cores. When, however, there is a large increase in the amount of memory that is required to run a benchmark of a certain class, we either see little increase or even a decrease in performance as the benchmark script is memory-limited.

Unsurprisingly, yet interesting to observe, different applications perform scale differently with a different number of cores. While increasing the number of cores will increase your computational capacity, it may strain the limited memory resources located on the processors. (This an important consideration for shared caches on any level.) Thus, it is also important to increase the size of the cache. This theme, however, is completely in line with a corollary to Amdhal's law, which suggests that there are diminishing returns to optimizing one aspect of the task very well. Since there is an upper bound in the speedup on can attain by optimizing only one part of a task, it is important to optimize all parts of the tasks to see the greatest overall speedup. In this case, we may be able to introduce speedup by having more cores, but we still need a larger cache to see speedup across all benchmarks.

References:

[1] http://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf