Extended Abstract: A Study of Red-Black SOR Parallelization Using Chapel, D and Go Languages

Sparsh Mittal
Oak Ridge National Lab, USA, mittals@ornl.gov

1. Introduction

Successive over-relaxation (SOR) is an important iterative solver for solving linear systems. We present parallel implementations of the red-black SOR method using three modern programming languages, namely Chapel, D and Go. We employ the SOR method for solving a 2D steady-state heat conduction problem. We discuss the optimizations incorporated and the features of these languages which are crucial for improving the program performance. Experiments have been performed using 2, 4, and 8 threads for a 4096×4096 square grid and performance results are compared with serial execution. The analysis of results provides important insights into the working of SOR method. These results have been published in a journal paper and both serial and parallel source-code in each of three languages have been made available to the public [1, 2]. We discuss the SOR algorithm and Chapel [3] specific details here and refer to [1] for other details.

2. Approach

The SOR method is used for solving 2D steady-state heat conduction problems. It involves iterative computations, where threads executing the same code in parallel must all complete one phase (viz. red or black) of the iteration before moving on to the next phase (or iteration). To ensure this, a synchronization barrier is used which enables worker threads to wait until all the threads have completed a phase before any thread continues.

Chapel specific features: In the Chapel language, we have utilized the task-parallel construct `begin` along with the synchronization construct `sync`. Using `begin`, the solver function is issued in an asynchronous manner, and using the `sync` statement, barrier synchronization is achieved.

Optimizations for SOR code: To optimize the SOR code, we have applied three optimizations. First, we have restructured the `for`-loops in a manner that `if`-condition checking is minimized. This reduces the branch misprediction penalty, which is especially large in modern processors with long pipelines. Second, since checking for convergence requires finding the maximum absolute error for all the cells, it becomes a critical section and hence requires mutex functionality. To avoid it, the convergence check is done in a serial manner. Third, since generally convergence is reached after thousands of iterations, testing for convergence at the end of each iteration may lead to extra computations and to avoid it, the convergence test is done only after every K (=4000 in our experiments) iterations.

3. Experimental Results

To gain highest performance in the final run, we compiled Chapel code with `-fast` flag (which also turns on the `-O3` flag for the compilation of the back-end C code produced by the Chapel compiler). The results are presented in Table 1. Note that it may be possible to optimize each code even further. We compare the performance scaling of these languages by comparing the execution time of parallel programs with the serial program written in the **same language**.

Table 1: Execution time and speedup values for different languages and different number of cores

	Execution time in seconds			Speedup relative to serial execution		
Threads	Chapel	D	Go	Chapel	D	Go
1 (Serial)	7538	8609	10551	-	-	-
2	3977	4099	5204	1.90	2.10	2.03
4	3139	3322	3834	2.40	2.59	2.75
8	2824	3141	3052	2.67	2.74	3.46

For a small number of threads (e.g. 2), the performance scales nearly linearly (a slight variation can be attributed to load on the host machine), however, for a large number of threads (e.g. 8), the performance does not scale linearly. This is due to the fact that the SOR program involves multiple iterations and phases; and the synchronization required after each phase creates a serialization bottleneck, which prevents multiple threads from progressing independently. Moreover, as the number of cores increases, although the processing power increases, the other resources such as cache, memory bandwidth etc. do not increase linearly and hence, the program performance does not scale linearly.

Impact of our work: The relevant paper and codes have been viewed/downloaded more than 2000 times and D/Go/Chapel community along with CFD (computational fluid dynamics) researchers have shown interest in them. As processors with many cores proliferate [4], we believe that our approach of using HPC languages such as Chapel for accelerating computation-intensive kernels will become even more useful. Our future work will focus on solving the SOR problem for 3D grids

References

[1] S. Mittal, "A Study of Successive Over-relaxation Method Parallelization Over Modern HPC Languages," *IJHPCN*, 2014. [Online]. Available: http://goo.gl/G5kLXt

[2] [Online]. Available: http://goo.gl/MkhkZS

[3] B. Chamberlain *et al.*, "Parallel programmability and the Chapel language," *IJHPCA*, 2007.

[4] V. Ahuja *et al.*, "Cache-aware affinitization on commodity multicores for high-speed network flows," in *ANCS*, 2012, pp. 39–48.