A Study of Successive Over-relaxation (SOR) Method Parallelization Over Modern HPC Languages

Sparsh Mittal

Future Technologies Group
Oak Ridge National Laboratory (ORNL)
Oak Rdge, TN, USA
Email: sparsh0mittal@gmail.com

Abstract:
Successive over-relaxation (SOR) is a computationally intensive, yet extremely important iterative solver for solving linear systems. Due to recent trends of exponential growth in amount of data generated and increasing problem sizes, serial platforms have proved to be insufficient in providing the required computational power. In this paper, we present parallel implementations of red-black SOR method using three modern programming languages namely Chapel, D and Go. We employ SOR method for solving 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 and performance results are compared with serial execution. The analysis of results provides important insights into working of SOR method.

Keywords: Successive over-relaxation (SOR), Chapel programming language, Go programming language (golang), D programming language (dlang), multithreading, high-performance computing (HPC), Parallelization.

Biographical notes: Sparsh Mittal received the B.Tech. degree in electronics and communications engineering from IIT, Roorkee, India and the Ph.D. degree in computer engineering from Iowa State University, Ames, IA, USA. He is currently working as a Post-Doctoral Research Associate at Oak Rige National Laboratory (ORNL), TN, USA. In his B.Tech. degree, he was the graduating topper of the batch and his major project was awarded institute silver medal. He was awarded scholarship and fellowship from IIT Roorkee and Iowa State University. His research interests include high-performance computing, multicore system architecture and graphics processing units (GPUs).

.

1 Introduction

Successive over-relaxation (SOR) is one of the most important method for solution of large linear systems ([1–3]). It has applications in CFD (computational fluid dynamics), mathematical programming ([4]), medical analysis ([5]) and machine learning ([6]) etc. The example of applications of SOR in CFD include study of steady heat conduction, turbulent flows, boundary layer flows or chemically reacting flows. For this reason, SOR method is important for both researchers and business policymakers.

Due to recent trends of exponential growth in amount of data generated ([7, 8]) and increasing problem sizes, serial platforms have proved to be insufficient in providing the required computational power. Hence, parallelization of computation intensive problems has become essential. Moreover, as the chip power-budget considerations restrict processor frequency-scaling, processor designers have focused on using tens of cores on a single chip to achieve high-performance and future processors are expected to have hundreds of cores. Thus, high-performance computing approach is expected to be even more useful for future systems.

In this paper, we present parallel implementations of SOR method using three concurrent programming languages, namely Chapel (from Cray Inc. ([9]), D (from Digital Mars ([10])) and Go (from Google ([11])). The SOR method is used for solving 2D steady-state heat conduction problem. We discuss the relavant programming constructs of these languages and compare them to gain insights into their features which are crucial for improving the program performance.

Accepted in Int. J. of High Performance Computing and Networking, 2014.

Experiments have been conducted with a square grid of dimension 4096×4096. Further, SOR method has been parallelized using 2, 4 and 8 threads using all the three languages and their performance has been compared with the serial execution. The analysis of results provides important insights into working of SOR method. The results also highlight the importance of using high-performance computing approach for obtaining solution of grid with large dimensions.

The rest of the paper is organized as follows. Section 2 presents a background on Chapel, D and Go languages and SOR method. Section 3 presents the algorithm, optimizations, implementation details and salient features of our approach. Section 4 discusses the performance results. Finally, section 5 concludes this paper.

2 Background and Related Work

In this section, we briefly review the SOR method and programming features of Chapel, D and Go.

2.1 Successive Over Relaxation (SOR) Method

For solution of partial differential equations, both direct and iterative solvers have been used. The direct solvers are susceptible to round-off errors; while iterative solvers provide the opportunity to achieve desired accuracy by trading-off the speed. Moreover, the iterative solvers are generally more memory-efficient than the direct solvers and hence are especially useful for solving large-sized problems. For this reason, we focus on iterative solvers in this paper. The examples of iterative solvers include Jacobi method, Gauss-Siedel (GS) method and SOR method.

The Jacobi method is a well-known iterative solver method which computes the value for iteration k based on values from iteration $k-1$. The Jacobi method is easily parallelized, however, its slow convergence rate prevents its use for any real-life application. GS method partially makes use of new values to reach to convergence faster. SOR is an extrapolation of the GS method which works by using weighted version of previous and computed GS iterate to accelerate its convergence.

$$X_k = \omega \overline{X_k} + (1-\omega) X_{k-1} \tag{1}$$

Here $\overline{X_k}$ is the k-th Gauss-Siedel iterate and $0 < \omega < 2$ is the extrapolation factor. By choosing a suitable value of ω, the convergence rate of SOR can be improved. GS method is a special case of SOR for $\omega = 1$. Since methods to improve the speed of convergence is outside the scope of this work, we use a single value of ω as 0.376.

The simulation is performed till the largest value of $|X_k(i,j) - X_{k-1}(i,j)|$ (i.e. difference between successive values of X_k) for any grid point (i,j) remains greater than ϵ, where ϵ shows the numerical tolerance. By choosing a suitable value of ϵ, a trade-off can be exercised between simulation speed and accuracy.

2.2 Chapel, D and Go Programming Languages

HPC is a promising approach for accelerating computational-intensive applications, such as multimedia processing ([12, 13]), atmospheric simulations ([14]), processor simulations ([15]), medical imaging ([16]), bioinformatics ([17]) etc. In recent years, HPC has been widely used by researchers ([18–21]). Several parallel programming languages have been used such as D, Go, Chapel, Java, OpenMP, and X10 ([22]) etc. These languages facilitate writing large-sized programs and provide programming constructs to express parallelism present in the program.

In this paper, we use Chapel, D and Go to accelerate SOR method. These languages have some similarities along with some unique features. These languages provide concurrent programming facility as part of the language itself. Neither of them require VM (virtual machine), rather they are statically compiled. Chapel aims at improving the performance, programmability, portability, and robustness of high-end processors, while also facilitating parallel programming on commodity clusters or desktop multicore systems. Both D and Go are system programming languages and provide automatic garbage-collection, and faster compilation speed than C/C++ ([23]). Chapel does not provide automatic garbage-collection.

These languages also have some important differences. Relative to each other, Go aims more for simplicity and faster development, D aims for providing more features and Chapel aims for higher performance. Go does not have classes and only uses structs and interfaces; Chapel has classes and records, while D has classes and structs. Further, unlike D and Chapel, Go does not provide operator over-loading. The differences in methods used for parallel programming are discussed in Section 3.

2.3 HPC Techniques for SOR Method

Recently, several researchers have used GPU (graphics processing unit) to accelerate SOR method by offloading the memory intensive computations to GPU ([24–26]). Compared to these, our implementation does not require a special-purpose hardware. In contrast to CPUs, GPUs have much smaller onboard memory and the bandwidth of the bus connecting CPU memory to GPU memory subsystem has limited bandwidth. Hence, for applications which require large working sets and number of iterations, the overhead of data-transfer between GPU and CPU presents a severe performance bottleneck ([27]). Moreover, use of GPUs also entails the overhead of porting the legacy code to graphics hardware, which may be economically infeasible in many scenarios.

3 Methodology

3.1 Parallelization of SOR Method

In literature several parallel version of SOR have been proposed such as red-black SOR, multi-color SOR ([28]), block-parallel SOR ([29]). In this paper, we use red-black SOR method.

Red-black SOR divides the grid into a chessboard of red and black cells, as shown in Figure 1. For a given row value i and column value j, a red cell is one for which $(i + j)$ is even and a black cell is one for which $(i + j)$ is odd. Clearly, all red cells have black cells as their neighbours and vice-versa.

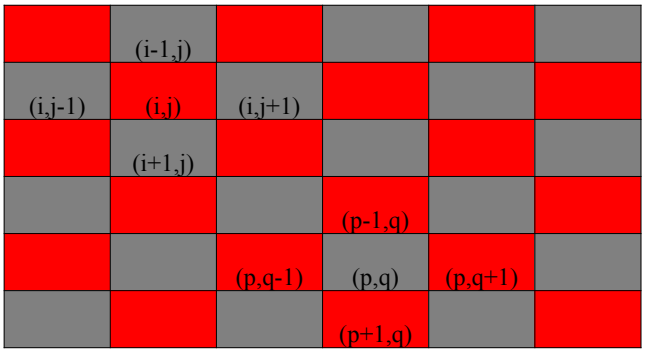

Figure 1 The checkerboard configuration for SOR. Note that all red cells have black cells as their four neighbors and vice versa.

The red-black group identification strategy along with use of the five-point finite-difference stencil leads to uncoupling of the solution of Eq. 1 at interior cells such that the value at the red cells depends only on the value at the black cells and vice versa. Thus, red-black SOR divides the iteration in two steps, namely red phase and black phase. In any iteration, first red cells can be updated and then for updating the black cells, the value just computed for red cells can be used. Clearly, such a strategy allows straightforward parallelization.

3.2 Using Chapel, D and Go For Parallelization

SOR method 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 all completed a phase before any thread continues. We now describe the parallelization approach and relavant programming constructs of Chapel, D and Go which enable us to achieve these functionalities.

3.2.1 Chapel Programming Language

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

3.2.2 D Programming Language

In D language, we have utilized the functionality of `std.parallelism` module. Each worker is started as a new `task`. Using `put` command, a `task` is queued to the `taskPool` for execution. The `taskPool` encapsulates a task queue and executes the task by efficiently mapping them onto the threads. For barrier synchronization, `yieldForce` function is used for each running task, and thus, the control waits till all the threads have finished execution.

To allow different threads to access the global data, `shared` qualifier is used to designate that a piece of data is shared in different threads. Otherwise, by default, the data is local to each thread and hence, multiple threads cannot safely access it. For designating that a piece of data is constant and hence safe for concurrent reading, `immutable` keyword is used.

3.2.3 Go Programming Language

In Go, we use *G*oroutines to achieve concurrent programming. A function which is called with `go` in its front is executed in its own goroutine. As an example, for the following code,

```
go function1()
function2()
```

both `function1` and `function2` run concurrently. Goroutines are lightweight and are multiplexed onto a set of threads. When a goroutine is blocked, the runtime automatically moves other goroutines on the same operating system thread to a different, runnable thread to allow maximum utilization of the resources.

The maximum number of processors to be used are specified using `GOMAXPROCS` function from `runtime` package. For implementing barrier synchronization, we used `WaitGroup` variable. Using `Add` function, the number of goroutines to wait for is specified and each such goroutine issues `Done` function to signal completion. When all goroutines complete, the barrier is released.

3.3 Optimizations Incorporated

To achieve high performance, we have applied several optimizations to both serial and parallel versions. We now discuss these optimizations briefly and then present the algorithm in the next section.

Code restructring to avoid branch misprediction penalty: Modern processors use long pipelines to allow instruction-level parallelism (ILP), however, this also increases the penalty of branch mispredictions. Branch prediction is required for predicting the outcome of conditional statements like `if`. To reduce branch misprediction penalty, we have

Code 1: Requires checking `if` condition several times

```
for (i= 0; i < DIM; i++)
    for(j= 0; j< DIM; j++)
{
    if ( (i+j)%2 ==0)
      doProcessing()
}
```

Code 2: Avoids checking `if` condition by restructring `for` loop

```
 for (i= 0; i < DIM; i+= 2)
    for(j= 0; j< DIM; j+= 2)
{
      doProcessing()
}
for (i= 1; i < DIM; i+= 2)
    for(j= 1; j< DIM; j+= 2)
{
      doProcessing()
}
```

Figure 2 Optimization: Avoiding `if` (branch statement) by restructring `for` loop for red phase. Similar idea applies to black phase also. Here `DIM` shows dimension of the grid.

structured the `for`-loops in a manner that `if`-condition checking is minimized, as shown in Figure 2. This also allows the compiler to apply optimization techniques such as loop-unrolling and vectorization, if applicable. Moreover, since branch conditions are removed, the runtime performance is significantly improved.

Minimizing serial execution bottleneck: In SOR method, checking for convergence requires finding the maximum absolute error for all the cells. This would require comparing the error value at all the cells to a single `maxChange` value (see Algorithm 1). This represents a critical section and to avoid data race condition, mutex functionality might be required. To avoid it, convergence check is done in a serial manner (see Algorithm 1). This increases the storage overhead slightly, but it also provides performance improvement.

Choice of granularity of convergence check: In SOR method, since generally convergence is reached after thousands of iterations, testing for convergence at the end of each iteration may lead to extra computations (e.g. comparisons). To reduce the overhead of convergence test, we perform it only once after every `K` iterations. The choice of `K` presents a trade-off since a large value of `K` may lead to performing extra iterations even after convergence is reached and a small value of `K` may lead to increase overhead of convergence checking. We have heuristically chosen `K` as 4000 to keep a balance between these two factors.

3.4 Parallel SOR Algorithm

Algorithm 1 shows the parallel SOR algorithm for the case of 2D steady-state heat conduction problem. Initially, the temperature at north boundary is assumed to be 1.0 unit, while at all other boundaries, it is assumed to be 0.

The algorithm proceeds as follows. Initially, the grid is initialized. The main routine runs for a maximum of `MaxIterations` number of iterations. In each iteration, red and black phases spawn P workers (e.g. threads or goroutines) which need to be synchronized at the end of the phase. Each worker updates the cells of the given color allocated to it. Convergence check is performed only when `shouldCheckConvergence` is true, which happens after every `K` iterations. The serial version

Algorithm 1: Parallel SOR Algorithm (see Section 3.4)

Input: Initial temperature profile, P (number of workers) and ω.
Output: Final temperature profile and whether SOR converged
Constants Used: `MaxIterations` (max number of iterations), `K` (number of iterations after which convergence is checked) and ϵ (tolerance)
Variables Used: `gridData` and `gridDataOld`: 2D arrays, `hasConverged` = false, `maxChange`= 0.0 and `shouldCheckConvergence` (whether to check for convergence in this iteration) = false
Initialize the `gridData` with initial temperature profile
Algorithm for main routine
foreach *iteration* `iter` = *1 to* `MaxIterations` **do**
  if `iter` *is a multiple of* `K` **then**
    `shouldCheckConvergence` = true
    Copy entire `gridData` to `gridDataOld`
  else
    `shouldCheckConvergence` = false
  end
  Call updateGridRed with P workers in parallel
  Synchronize
  Call updateGridBlack with P workers in parallel
  Synchronize
  if `shouldCheckConvergence` **then**
    `maxChange` =0
    foreach *Cell* (i, j) *in the grid* **do**
      `maxChange` = Maximum($|gridData(i, j) - gridDataOld(i, j)|$, `maxChange`)
    end
    if `maxChange` $< \epsilon$ **then**
      hasConverged = true
      break
    end
  end
end
Print value of hasConverged. Return.
updateGridRed() for worker p_j
foreach *Cell of red color given to worker* p_j **do**
  Update `gridData` using Eq. 1
end
updateGridBlack() for worker p_j
foreach *Cell of black color given to worker* p_j **do**
  Update `gridData` using Eq. 1
end

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

of SOR algorithm follows along the same lines and has been omitted for the sake of brevity.

4 Experimental Platform and 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) and D code with `-inline -O -release` flags. For Go code, no suitable optimization flag could be found. The values of constants are taken as $\epsilon = 0.00001$, `MaxIterations` $= 50000$ and `K` $= 4000$. The dimension of the grid is 4096×4096. For each of the three languages, we wrote both serial and parallel programs. 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 presents these execution time values. It also presents the speedup values, where speedup S_P for P threads is defined as

$$S_P = \frac{T_1}{T_P} \tag{2}$$

Here T_1 and T_P refer to the execution time using 1 (serial) and P threads, respectively for the same language.

We now analyze these results. For the SOR program, out of the three languages tested, Chapel language gives the highest performance (Note that the above mentioned results should not be taken to conclude about the performance of these languages for all possible programs, since a different program and coding style might produce different results). For 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 large number of threads (e.g. 8), the performance does not scale linearly. This is due to the fact that SOR program involves multiple iterations and phases; and the synchronization required after each phase creates serialization bottleneck due to which multiple threads do not progress independently. If the number of iterations required for convergence is L, then synchronization is required for $2 \times L$ times. Moreover, with the increasing number of cores, 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.

5 Conclusion and Future Work

In this paper, we have highlighted the importance of using HPC approach [30] for accelerating solution of SOR method. We presented parallel implementations of SOR method for solving 2D heat transfer problem. We discussed the features of Chapel, D and Go, along with their functions which are important for achieving parallel implementation of SOR method. Experimental results have been conducted using 2, 4 and 8 threads and analysis of results provides important insights into SOR method. Our future work will focus on solving the SOR problem for 3D grids. We also plan to study parallelization of other computation intensive problems.

References

[1] D. Young, *Iterative solution of large linear systems.* Dover Publications, 2003.

[2] A. Hadjidimos, "Successive overrelaxation (SOR) and related methods," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1, pp. 177–199, 2000.

[3] L. M. Adams and H. F. Jordan, "Is SOR color-blind?," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 2, pp. 490–506, 1986.

[4] C. Cryer, "The solution of a quadratic programming problem using systematic overrelaxation," *SIAM Journal on Control*, vol. 9, no. 3, pp. 385–392, 1971.

[5] L. Ha, J. Krüger, S. Joshi, and C. Silva, "Multi-scale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs," in *GPU Computing Gems*, vol. 1, 2010.

[6] O. Mangasarian and D. Musicant, "Successive overrelaxation for support vector machines," *Neural Networks, IEEE Transactions on*, vol. 10, no. 5, pp. 1032–1037, 1999.

[7] S. Mittal and A. Mittal, "Versatile question answering systems: seeing in synthesis," *International Journal of Intelligent Information and Database Systems*, vol. 5, no. 2, pp. 119–142, 2011.

[8] S. Mittal, S. Gupta, and A. Mittal, "BioinQA: Metadata based Multidocument QA system for addressing the issues in Biomedical domain," *Int. J. of Data Mining, Modelling and Management (IJDMMM)*, vol. 5, no. 1, pp. 37–56, 2013.

[9] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[10] A. Alexandrescu, *The D Programming Language.* Addison-Wesley Professional, 2010.

[11] Google, "The Go Programming Language ." `http://golang.org/`, 2012.

[12] S. Gupta, S. Mittal, S. Dasgupta, and A. Mittal, "MIMO Systems For Ensuring Multimedia QoS Over Scarce Resource Wireless Networks," *ACM International Conference On Advance Computing, India*, 2008.

[13] A. Pande *et al.*, "BayWave: BAYesian WAVElet-based image estimation," *International Journal of Signal and Imaging Systems Engineering*, vol. 2, no. 4, pp. 155–162, 2009.

[14] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable CPU-GPU algorithm for global atmospheric simulations," in *18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 1–12, 2013.

[15] S. Mittal and Z. Zhang, "Integrating sampling approach with full system simulation : Bringing together the best of both," in *IEEE International Conference On Electro/Information Technology (EIT)*, (USA), 2012.

[16] Y. Liu, A. Fedorov, R. Kikinis, and N. Chrisochoides, "Real-time non-rigid registration of medical images on a cooperative parallel architecture," in *IEEE International Conference on Bioinformatics and Biomedicine*, pp. 401–404, 2009.

[17] Y. Zhang, S. Misra, D. Honbo, A. Agrawal, W.-k. Liao, and A. Choudhary, "Efficient pairwise statistical significance estimation for local sequence alignment using GPU," in *International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pp. 226–231, IEEE, 2011.

[18] S. Gupta *et al.*, "Guaranteed QoS with MIMO Systems for Scalable Low Motion Video Streaming Over Scarce Resource Wireless Channels," in *International Conference On Information Processing*, 2008.

[19] S. Mittal *et al.*, "FPGA: An efficient and promising platform for real-time image processing applications," in *National Conference On Research and Development In Hardware Systems (CSI-RDHS)*, 2008.

[20] S. Mittal, S. Gupta, and S. Dasgupta, "System Generator: The State-Of-Art FPGA Design Tool For DSP Applications," in *Third International Innovative Conference On Embedded Systems, Mobile Communication And Computing (ICEMC2 2008)*, Global Education Center, India, 2008.

[21] M. Raju *et al.*, "High performance computing of three-dimensional finite element codes on a 64-bit machine," *Journal of Applied Fluid Mechanics*, vol. 5, no. 2, pp. 123–132, 2012.

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *ACM SIGPLAN Notices*, vol. 40, pp. 519–538, ACM, 2005.

[23] R. Hundt, "Loop Recognition in C++/Java/Go/Scala," *Proc. Scala Days*, 2011.

[24] P. Di, Q. Wan, X. Zhang, H. Wu, and J. Xue, "Toward harnessing doacross parallelism for multi-GPGPUs," in *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 40–50, IEEE, 2010.

[25] L. Itu, C. Suciu, F. Moldoveanu, and A. Postelnicu, "GPU accelerated simulation of elliptic partial differential equations," in *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, vol. 1, pp. 238–242, IEEE, 2011.

[26] E. Konstantinidis and Y. Cotronis, "Graphics processing unit acceleration of the red/black SOR method," *Concurrency and Computation: Practice and Experience*, 2012.

[27] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 4, IEEE Press, 2008.

[28] L. Adams and J. Ortega, "A multi-color SOR method for parallel computation," in *Proceedings of the International Conference on Parallel Processing*, vol. 3, p. 53, Citeseer, 1982.

[29] D. Xie, "A new block parallel sor method and its analysis," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1513–1533, 2006.

[30] S. Mittal, A. Pande, L. Wang, and P. Kumar, "Design Exploration and Implementation of Simplex Algorithm over Reconfigurable Computing Platforms," in *IEEE International Conference on Digital Convergence*, pp. 204–209, 2011.