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

SPARSH MITTAL

OAK RIDGE NATIONAL LAB, USA.

ANNUAL CHAPEL IMPLEMENTERS AND USERS WORKSHOP
JUNE, 2015

Results used in this paper

- Sparsh Mittal, "**A Study of Successive Over-relaxation Method Parallelization Over Modern HPC Languages**", International Journal of High Performance Computing and Networking, vol. 7, number 4, pp. 292-298, 2014.
- Code available for download at: https://drive.google.com/folderview?id=0B3CSJpITzNscMVBpb3pfUFcwVzQ&usp=sharing
- Purpose: studying parallelization features of Chapel, D and Go, **not** to compare their performance

Presentation Plan

- Quick introduction of SOR
- Reason behind choice of SOR
- Optimization of SOR and the parallel algorithm
- SOR Parallelization in Chapel, D and Go
- Experiments and Results
- Salient Features of Chapel
- Comparison of Chapel with other languages
- Conclusion and future work

Successive Over-Relaxation Method

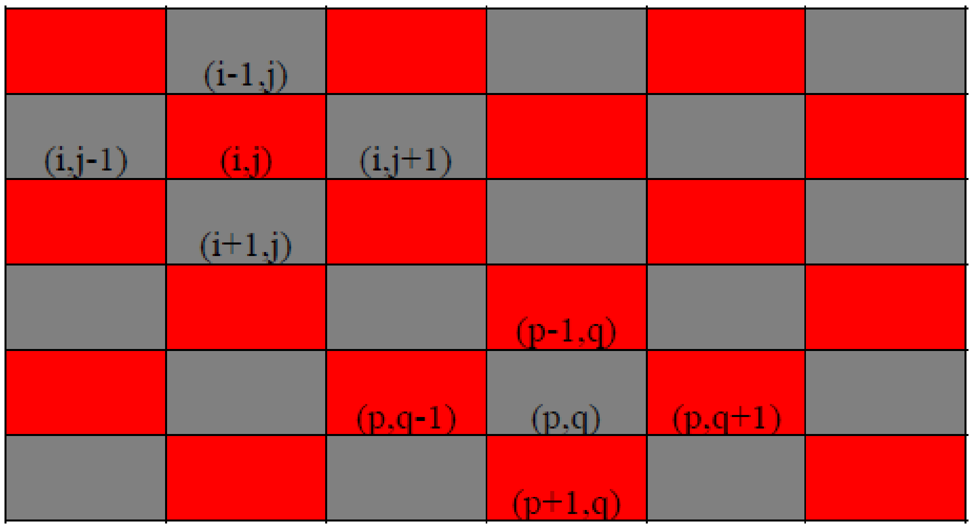
- An iterative method for solving partial differential equations
- More memory efficient than direct method
- Allows trading off accuracy with speed
- Converges faster than Jacobi method

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

- $\overline{X_k}$ is the k-th Gauss Siedel iterate
- $0 < \omega < 2$ is the extrapolation factor.

Red-black SOR

- Out of several possible parallel SOR versions, we choose red-black SOR
- Here all red cells have black cells as their four neighbors and vice versa



- This allows uncoupling of the solution at interior cells
- In an iteration, first update red cells, then while updating black cells, just use updated values of red cells
- This strategy allows straightforward parallelization

Why we chose Red-black SOR

- Parallel but not embarrassingly parallel
- Requires synchronization and convergence check
- Iterative in nature
- Reasonably small problem to allow focusing on key principles
- Useful for research and many real-life problems, e.g. computational fluid dynamics (CFD)

Optimizations for SOR

- Convergence check is done in serial manner
  - This avoids serial bottleneck which requires mutex functionality and incurs performance overhead

- Granularity of convergence check is kept high, since convergence is usually reached after many iterations
  - In our experiments, convergence is checked after 4000 iterations

Restructuring loop to avoid 'if' statements

Requires more if checks	Requires less if checks

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

Refer http://stackoverflow.com/questions/11227809/why-is-processing-a-sorted-array-faster-than-an-unsorted-array

Parallel SOR algorithm for 2D steady-state heat conduction problem

```
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,
shouldCheckConvergence (whether to check for
convergence in this iteration) = false and
maxChange= 0.0
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 < ϵ 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
```

Initialization

Solve red cells & Synchronize

Solve black cells & Synchronize

Check for convergence

Parallelization of each SOR iteration in different languages

Chapel Language

- Solver is issued using **begin**
- Synchronization achieved using **sync**

```
sync {
for p in 1..nSlaves {
begin SolveRed(p);
   }
 }

sync {
for p in 1..nSlaves {
begin SolveBlack(p);
   }
 }
```

D Language

- We used functionality of **std.concurrency**
- Start new thread using **spawn**
- Thread id of the caller **thisTid**.
- **__gshared** to share a variable across all threads
- **Barrier** from **core.sync** for sync'ing multiple threads.

```
__gshared Barrier barr = null;
{
 barr = new Barrier(nSlaves+1);
  for (int cc=0; cc<nSlaves; cc++)
  {
 spawn(&SolveRed, thisTid,cc);
   }
barr.wait(); //sync
}
{
  barr = new Barrier(nSlaves+1);
  for (int cc=0; cc<nSlaves; cc++)
  {
spawn(&SolveBlack, thisTid,cc);
   }
barr.wait(); //sync
}
```

Go Language

- We used Goroutines for concurrent programming
- **WaitGroup** for barrier synchronization
- **Add** function to specify number of goroutines to wait for
- Each goroutine issues **Done** to function to signal completion.
- When all goroutines complete, the barrier is released.

```
var wg sync.WaitGroup

wg.Add(nSlaves)
for p := 0; p < nSlaves; p++
{
go SolveRed(p, isCheck)
}
wg.Wait()

wg.Add(nSlaves)
for p := 0; p < nSlaves; p++
{
go SolveBlack(p, isCheck)
}
wg.Wait()
```

```
sync {
 for p in 1..nSlaves {
 begin SolveRed(p);
   }
 }
```

```
sync {
for p in 1..nSlaves {
begin SolveBlack(p);
   }
 }
```

Chapel

```
{
 barr = new Barrier(nSlaves+1);
  for (int cc=0; cc<nSlaves; cc++)
  {
 spawn(&SolveRed, thisTid,cc);
   }
      //sync.
      barr.wait();
}
```

```
{
  barr = new Barrier(nSlaves+1);
  for (int cc=0; cc<nSlaves; cc++)
  {
spawn(&SolveBlack, thisTid,cc);
   }
      //sync.
      barr.wait();
}
```

D

```
var wg sync.WaitGroup

wg.Add(nSlaves)
for p := 0; p < nSlaves; p++
{
go SolveRed(p, isCheck)
}
wg.Wait()
```

```
wg.Add(nSlaves)
for p := 0; p < nSlaves; p++
{
go SolveBlack(p, isCheck)
}
wg.Wait()
```

Go

Experiments

- Compile Chapel code with **--fast** flag
- Compile D code with **-inline -O -release** flags.
- We could not find suitable flag for Go code

- Grid dimension 4096 X 4096
- MaxIterations 50,000, $\omega = 0.376$
- Convergence check after every 4000 (=K) iterations
- $\varepsilon = 0.00001$ (maximum diff b/w two iterations)
- Speedup = $T_{serial}/T_{parallel}$

Results

	Execution time (seconds)			Speedup w.r.t. their serial version		
	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	2834	3141	3052	2.67	2.74	3.46

Note: speedups are compared to serial language in the same language.

Some comments on results

- For small number of threads (e.g. 2) performance scales linearly
- With increasing threads, performance does not scale linearly due to
  - Thread synchronization for both red and black phase
  - Limited memory bandwidth and cache etc.

Some salient features of Chapel

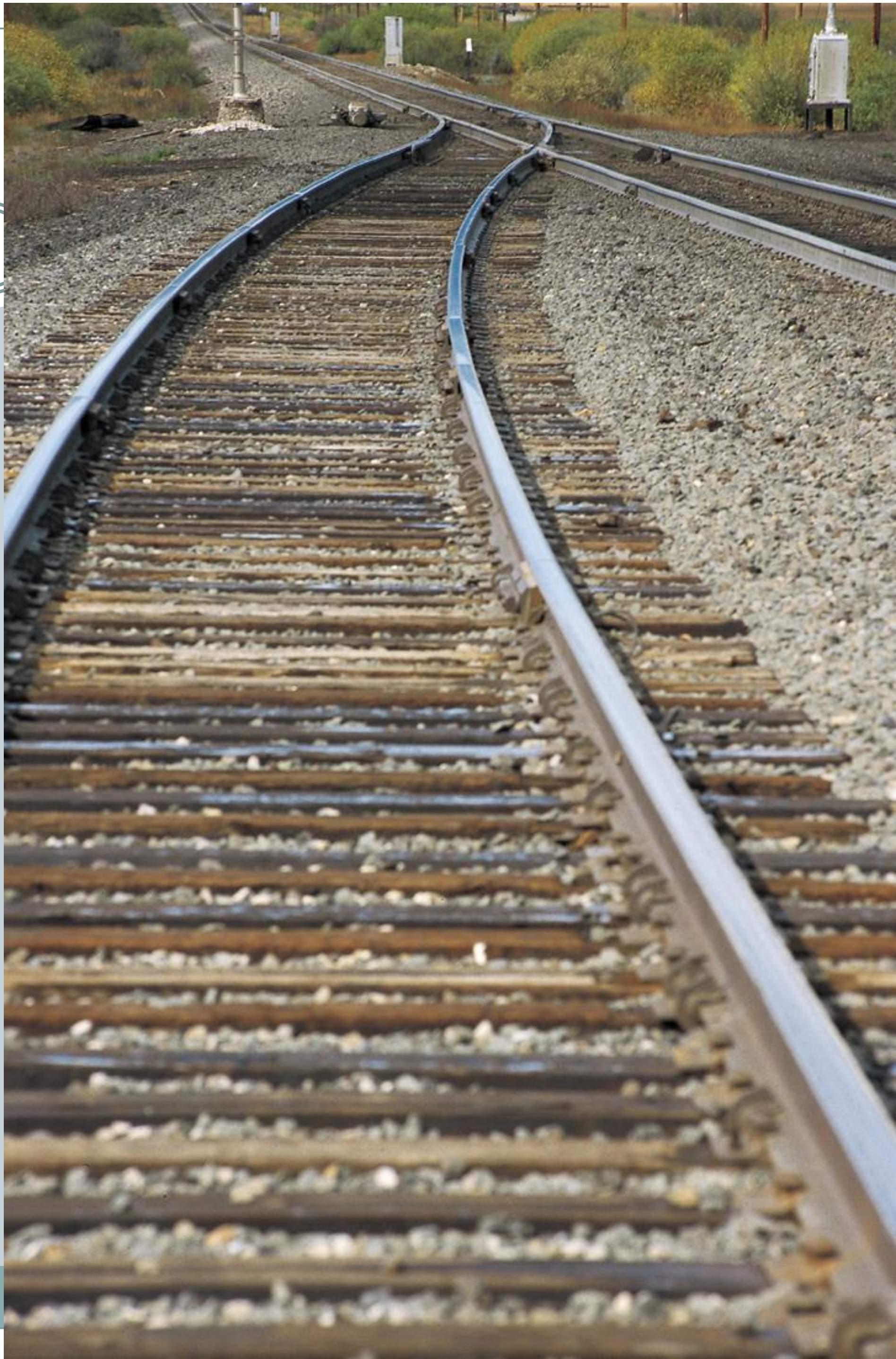
- Provides features for concurrent programming as part of language itself, and not library or pseudo-comment directives
- Can target inter-node, intra-node and instruction-level parallelism
- Supports both data and task parallelism.
- Interoperability with C/C++
- Provides several object-orient programming features
- Supports arbitrarily nested parallelism and composition of parallel tasks

Comparison of Chapel with other languages

- D/Go provide auto garbage collection, Chapel doesn't
- D/Go/Chapel execute natively, unlike Java => speed
- OpenMP has limited support for synchronization operations inside parallel loops. Unlike OpenMP, Chapel is a language itself and allows supporting higher-level data abstractions
- D allows exception handling, Chapel/Go do not
- No inheritance or classes or function/operator overloading in Go
- Go function can return multiple values as such.

Conclusion and Future Work

- We parallelized SOR in Chapel, D and Go.
- Future Work
  - Solving SOR for 3D grid
  - Study of other languages
  - Experiments with larger number of threads
  - Further optimizing each program



Questions and comments are welcome!

Sparsh Mittal

mittals@ornl.gov