CS515 Parallel Programming: Assignment #4

Due on June 1st, 2016 at 11:59pm

Jingke Li Spring 2016

Konstantin Macarenco

Implementation details

Implementation of "C" version was straight forward and didn't cause any problems, converting Jacobi to Gauss was simple - reuse the same matrix when preforming the computation.

```
for (i = 1; i < n-1; i++) {
    for (j = 1; j < n-1; j++) { ...
        old = x[i][j]
        x[i][j] = ...
```

Conversion from Gauss-Seidel normal order to Red-Black, order is implemented by using 4 loops:

```
for (i = 1; i < n-1; i = i + 2) { // Red - odd column odd row (row + column \% 2 := 0)
    for (j = 1; j < n-1; j = j + 2) { ...

for (i = 2; i < n-1; i = i + 2) { // Red  - even column even row (row + column \% 2 := 0)
    for (j = 2; j < n-1; j = j + 2) { ...

for (i = 1; i < n-1; i = i + 2) { // Black - odd column even row (row + column \% 2 := 1)
    for (j = 2; j < n-1; j = j + 2) { ...

for (i = 2; i < n-1; i = i + 2) { // Black - even column odd row (row + column \% 2 := 1)
    for (j = 1; j < n-1; j = j + 2) { ...
```

In Chapel Implementation looks quite similar, except for language specifics:

```
forall ij in innerDomain do { const innerDomain = BD[{1..n-2,1..n-2}];

forall ij in BDr1 do { ... // const BDr1 = BD[1..n-2 by 2, 1..n-2 by 2];
forall ij in BDr2 do { ... // const BDr2 = BD[2..n-2 by 2, 2..n-2 by 2];
forall ij in BDb1 do { ... // const BDb1 = BD[1..n-2 by 2, 2..n-2 by 2];
forall ij in BDb2 do { ... // const BDb2 = BD[2..n-2 by 2, 1..n-2 by 2];
```

It can be done in two loops, however this way's advantage - there is no need to calculate correct row offset.

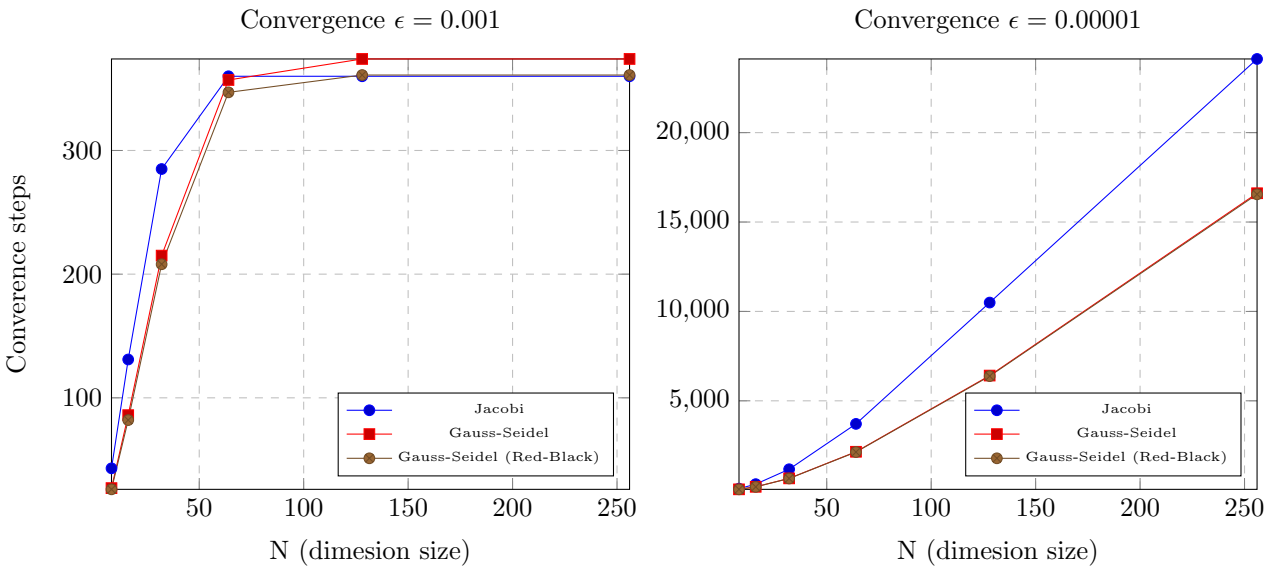
Performance

Chapel aims to make writing of parallel applications easy, and provides many built in mechanisms for this purpose. I enjoyed developing in Chapel, however the main problem with this language is performance (not sure if this is due to language problems or to my particular implementation of the Laplace algorithms). Sequential "C" is up to x9 times faster than similar shared memory algorithm in Chapel.

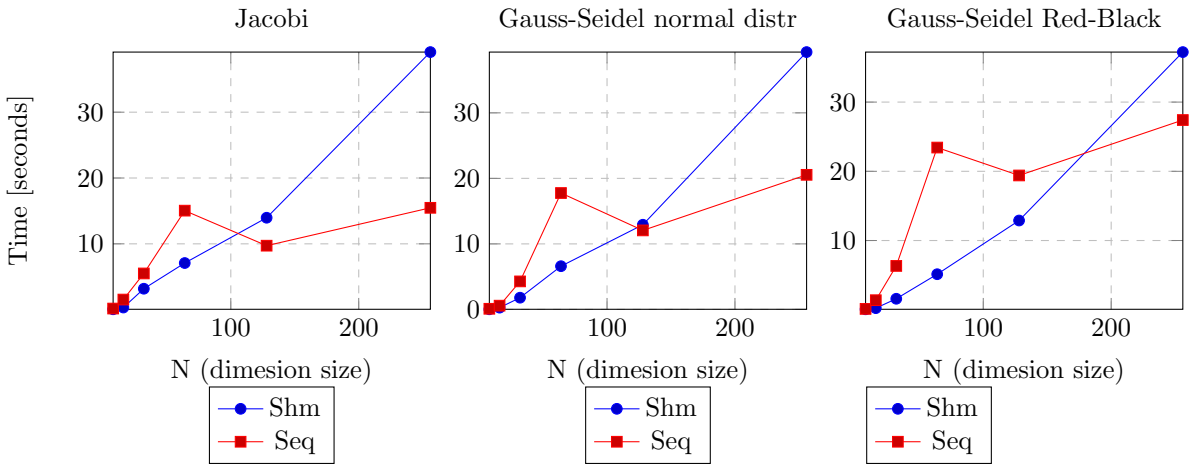
Distributed version written in Chapel required very little modifications to the original code - only correct mapping to the locales, however it was even slower than shared memory, with number locales doubled performance is slowed down by x2. Such a significant slowdown can be explained by high cost of communication between border locales, or by poor Implementation/Compilation of PGAS features.
In distributed version I verified locales affinity by creating functions with exactly the same semantics as Jacobi - Gauss methods, except instead of performing computation it records locale ID", and by adding print statement that outputs number of current locale and current matrix element locale. Printout can be enabled by setting verbose flag to > 0.

Convergence of all implementations is almost identical with insignificant deviations $\pm\%1-2$. Gauss-Seidel convergence rate (both versions), is around %40 better than Jacobi, this was expected since values propagate faster, when update happens "in place". Gauss-Seidel and Red-Black have about the same convergence rate, with Red-Black faster by some constant. The biggest downside of Red-Black, - need of at least two scans of the array, one for Red Cells and another for Black, which requires additional synchronization, however there is no potential concurrent writes as in Normal order. Shared memory version of Gauss-Seidel is usually faster, by a constant despite having a slower convergence rate.

With rough convergence tolerance, changes quickly propagate through the matrix, hence larger dimension have the same number of convergence steps, and vice versa:



Performance of shared memory vs sequential implementation (Chapel):



Performance of shared version vs distributed (CHAPEL) with number of locales = 2 (gets worse with higher number of locales - not shown):