First Assignment

Abhijit Chowdhary
High Performance Computing
Prof. Stadler

February 11, 2019

Before we begin, I am running the code Intel Core i5-3320M CPU @ 2.60GHz. This is a processor with 2 cores and 4 threads (on a thinkpad X230).

1 Matrix-Matrix Multiplication

Here is the reported Glop/s and GB/s for various compiler flags.

(a) This is with $-O0$.

```
Dimension       Time    Gflop/s       GB/s
       20   0.005186   0.451190   0.617012
       40   0.024997   0.758485   1.024115
       60   0.073145   0.878534   1.181222
       80   0.177361   0.860618   1.154708
      100   0.352622   0.846516   1.134360
      120   0.616006   0.838044   1.122067
      140   1.004585   0.816516   1.092591
      160   1.492329   0.820838   1.097882
      180   1.994492   0.874779   1.169621
      200   2.748728   0.870948   1.164175
      220   3.797066   0.839369   1.121708
      240   5.017759   0.824782   1.102006
      260   6.099069   0.862863   1.152700
      280   8.050332   0.816592   1.090738
      300   9.613858   0.841130   1.123378
      320  12.707382   0.772389   1.031463
      340  13.980333   0.842173   1.124551
      360  16.570923   0.843487   1.126214
      380  19.782859   0.831019   1.109486
      400  23.500127   0.815996   1.089356
      420  26.449047   0.839347   1.120464
```

```
         440  30.684599   0.831888   1.110446
         460  35.834093   0.814003   1.086518
         480  42.545943   0.778994   1.039742
         500  48.110461   0.778677   1.039275
         520  55.696070   0.756639   1.009824
         540  64.705882   0.729384   0.973414
         560  74.624145   0.705372   0.941336
         580  84.678332   0.690651   0.921662
```

(b) This is with $-O3$.

```
   Dimension        Time    Gflop/s       GB/s
          20    0.000471   4.971076   6.798053
          40    0.004728   4.009766   5.414031
          60    0.008584   7.485864  10.065028
          80    0.018819   8.111148  10.882882
         100    0.039227   7.609594  10.197111
         120    0.069211   7.458908   9.986822
         140    0.108867   7.534488  10.081991
         160    0.189460   6.465538   8.647742
         180    0.234924   7.426823   9.930014
         200    0.329532   7.264849   9.710742
         220    0.429543   7.419845   9.915663
         240    0.574186   7.207700   9.630330
         260    0.707709   7.436195   9.934030
         280    0.902064   7.287552   9.734119
         300    1.123595   7.196988   9.612004
         320    1.744750   5.625470   7.512364
         340    1.646584   7.150475   9.548008
         360    1.981027   7.055612   9.420566
         380    2.288765   7.182888   9.589802
         400    3.252054   5.896580   7.871947
         420    3.168978   7.005395   9.351660
         440    3.664246   6.966278   9.298938
         460    4.171241   6.992897   9.334009
         480    5.880642   5.635956   7.522444
         500    6.201603   6.040777   8.062432
         520   11.679547   3.608174   4.815529
         540   24.118377   1.956826   2.611519
         560   40.962487   1.285024   1.714896
         580   60.356087   0.968968   1.293073
```

These were computed with the changes:

```c
double flops = 3*n*m*(2*k-1)*NREPEATS/ time / 1e9;
double bandwidth = NREPEATS*n*m*k*sizeof(double) / time / 1e9;
```

2 Laplace equation in one space dimension

The error metric I used it the Euclidean Norm. For part (c)

1. Choosing $N = 100$.

   (a) For Gauss-Seidel, I found that it took 14176 iterations to take the initial error of $10 \to 10(1e-6)$ in about .12 seconds. In addition it took about 2994 to go from an initial error of $10 \to .5$ in about .0027 seconds.

   (b) For Jacobi, I found that it took 28348 iterations to take the initial error of $10 \to 10(1e-6)$ in about .1 seconds. In addition it took about 5985 to go from an initial error of $10 \to .5$ in about .0044 seconds.

2. Choosing $N = 10,000$, I wasn't able to get the error to come down from $100 \to 100*(1e-6)$ since it stopped getting finer at around .07. Instead I computed the error going down to .5. My results were:

```
Jacobi Laplace Solve for N = 10000 and 105260486 iterations: 3393.516600
Gauss-Seidel Laplace Solve for N = 10000 and 52630244 iterations: 3950.726431s.
```

Regarding running $N = 10,1000$ for 100 iterations using $-O0$ and $-O3$: For $-O0$ Jacobi took .026 secondsand Jacobi took .027 seconds; for $-O3$ Jacobi took .0055 seconds and Gauss-Seidel took .011 seconds. It seems like the optimization is doing some work on the Jacobi method. Included are the relevant bits of the code.

```
double ResidLaplace(long N, double *u, double *f)
{ //Compute ||Au - f|| under the Euclidean Norm
    double hsq = 1.0/((N+1)*(N+1)), a_diag = (2.0/(hsq)), a_bdiag = -1.0/hsq;
    double v = 0.0, tmp = 0.0;

    tmp = u[0]*a_diag + u[1]*a_bdiag - f[0]; v += tmp*tmp;
    tmp = ( u[N-1]*a_diag + u[N-2]*a_bdiag ) - f[N-1]; v += tmp*tmp;
    for (int i = 1; i < N-1; i++)
    {
        tmp = ( a_diag*u[i] + a_bdiag*(u[i-1] + u[i+1]) ) - f[i];
        v += tmp*tmp;
    }
    return sqrt(v);
}

/* Numerically approximates the solution to:
 * -u'' = f in (0,1) with u(0) = 0 and u(1) = 0
 * using the Jacobi method.
 * INPUT: long N: Number of discretization points
 *        double tol: Tolerance to compute solution to.
 *        double *f: discretization of f with N points
 *        double *u0: Initial guess vector for u
 * OUTPUT: double *u: Array of size N holding solution to equation.
 *         long it: Number of iterations needed to hit tol.
 */
long Laplace1DJacobi(long N, double tol, double *f, double *u0, double *u)
{ //Proceed via Jacobi iteration:
    double hsq = 1.0/((N+1)*(N+1)), a_diag = (2.0/(hsq)), a_bdiag = -1.0/hsq;
    double nu[N];
    for (long k = 0; k < N; k++) { u[k] = u0[k]; nu[k] = u0[k]; }

    //Compute u_i^{k+1} = 1/a_ii (f_i - sum_{j \neq i} a_ij u_j^k
    long it = 0;
    double cur_res = 0.0;
    while (ResidLaplace(N, u, f) > tol && it < 100)
    { //Note for laplace, the summation only has 2 terms max.
        nu[0] = (1/a_diag)*(f[0] - a_bdiag*u[1]);
        nu[N-1] = (1/a_diag)*(f[N-1] - a_bdiag*u[N-2]);
        for (long i = 1; i < N-1; i++)
        {
            nu[i] = (1/a_diag)*( f[i] - a_bdiag*(u[i-1] + u[i+1]) );
        }
        for (long j = 0; j < N; j++) { u[j] = nu[j]; }
        cur_res = ResidLaplace(N,u,f);
        it = it + 1;
    }
    return it;
}

/* Numerically approximates the solution to:
 * -u'' = f in (0,1) with u(0) = 0 and u(1) = 0
 * using the Gauss Siedel method.
 * INPUT: long N: Number of discretization points
 *        double tol: Tolerance to compute solution to.
 *        double *f: discretization of f with N points
 *        double *u0: Initial guess vector for u
 * OUTPUT: double *u: Array of size N holding solution to equation.
 *         long it: Number of iterations needed to hit tol.
 */
long Laplace1DGaussSeidel(long N, double tol, double *f, double *u0, double *u)
{ //Proceed via Seidel iteration:
    double hsq = 1.0/((N+1)*(N+1)), a_diag = (2.0/(hsq)), a_bdiag = -1.0/hsq;
    double nu[N];
    for (long k = 0; k < N; k++) { u[k] = u0[k]; nu[k] = u0[k]; }
    long it = 0;
    double cur_res = 0.0;
    while (ResidLaplace(N, u, f) > tol && it < 100)
    { //Note for laplace, the summation only has 2 terms max.
        nu[0] = (1/a_diag)*(f[0] - a_bdiag*u[1]);
        nu[N-1] = (1/a_diag)*(f[N-1] - a_bdiag*nu[N-2]);
        for (long i = 1; i < N-1; i++)
        {
            nu[i] = (1/a_diag)*( f[i] - a_bdiag*(nu[i-1] + u[i+1]) );
        }
        for (long j = 0; j < N; j++) { u[j] = nu[j]; }
        cur_res = ResidLaplace(N,u,f);
        it = it + 1;
    }
    return it;
}
```