High Performance Computing: Homework 1

Dmitriy (Tim) Kunisky [dk3105]

Problem 2: Matrix-Matrix Multiplication

- Processor: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
  - Maximum flop rate: 19.76 Gflop/s, or 4.94 Gflop/s per core
  - Maximum main memory bandwidth: 25.6 GB/s
- Compiler: Apple LLVM version 9.1.0 (clang-902.0.39.2)

	Flag `-O0`		Flag `-O3`	
Matrix Size	Flop Rate (Gflop/s)	Bandwidth (GB/s)	Flop Rate (Gflop/s)	Bandwidth (GB/s)
100	0.257	4.121	2.295	36.722
150	0.273	4.379	2.087	33.405
200	0.268	4.301	1.966	31.467
250	0.269	4.309	1.951	31.219
300	0.274	4.395	1.889	30.228
350	0.273	4.383	1.882	30.114
400	0.272	4.364	1.749	27.986
450	0.274	4.386	1.879	30.079
500	0.263	4.219	1.792	28.673
550	0.257	4.125	1.658	26.541

(Note that with the flag `-O3` the memory bandwidth exceeds the CPU's reported maximum. This is probably because it is inaccurate at this level of optimization to count the memory accesses in the code as written, since the repeated access to `c[...]` in each iteration is redundant and can likely be optimized out.)

Problem 3: Laplace Equation in 1D

Architecture details:

- Processor: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
- RAM: 16GB 1867 MHz DDR3 (maximum bandwidth $\sim$ 25.6 GB/s)

Typically, it appears that for the N values given neither of the two methods converges with the specified criterion. So, instead I measure the progress they make by taking the ratio of the initial residual with the final residual after 5,000 iterations. My code prints this out as the "residual shrinkage factor." The measurements of this number for each method and for the two given values of N are presented in the table below. Clearly, the Gauss-Seidel method is more effective per iteration than the Jacobi method in this sense.

N	Jacobi	Gauss-Seidel
100	1641.72	23492.83
10,000	791.66	1871.79

The timing results (in seconds) with the number of iterations restricted to 100 and $N = 10{,}000$ for the two given compiler flags are presented in the table below. The two methods are similar at the lowest level of optimization, but at the highest level of optimization the Jacobi method is substantially faster. This might (speculatively) be due to the updated and previous values of the $\boldsymbol{u}$ vector not sharing the same memory in the Jacobi method, allowing low-level parallelization of the operations performed on these values.

Flag	Jacobi	Gauss-Seidel
`-O0`	0.0145	0.0157
`-O3`	0.0055	0.0084

The source code is included starting on the following page.

```
// Usage:
// $ g++ -O3 -std=c++11 Laplace.cpp && ./a.out 10000 Jacobi
//
// The first argument is N and the second is the method. Optionally, a third
// parameter may be passed which sets the maximum number of iterations (5000
// by default):
// $ g++ -O3 -std=c++11 Laplace.cpp && ./a.out 10000 Jacobi 100

#include <stdio.h>
#include <math.h>
#include "utils.h"


// Computes \ell^2 norm of difference of u and v
double norm_diff(long n, double *u, double *v) {
  double ret = 0.0;

  for (long i = 0; i < n; i++) {
    ret += (u[i] - v[i]) * (u[i] - v[i]);
  }

  return sqrt(ret);
}


// Implements multiplication by discretized operator A
void MultA(long n, double *u, double *out) {
  double pre_factor = (n + 1) * (n + 1);

  out[0] = pre_factor * (2.0 * u[0] - u[1]);

  for (long i = 1; i < n - 1; i++) {
    out[i] = pre_factor * (2.0 * u[i] - u[i + 1] - u[i - 1]);
  }

  // -1.0 here is for boundary condition:
  out[n - 1] = pre_factor * (2.0 * u[n - 1] - u[n - 2] - 1.0);
}


// If u_old and u_new point to distinct arrays of double[n], then this
// implements the Jacobi method, where u_new is the next iterate. If u_old
// and u_new point to the same array, then this implements the Gauss-Seidel
// method, where u is overwritten with the updated values.
void SolverStep(long n, double *f, double *u_old, double *u_new) {
  u_new[0] = 0.5 * (f[0] / (n + 1) / (n + 1) + u_old[1]);

  for (long i = 1; i < n - 1; i++) {
    u_new[i] = 0.5 * (f[i] / (n + 1) / (n + 1) + u_old[i - 1] + u_old[i + 1]);
  }

  // +1.0 here is for boundary condition:
  u_new[n - 1] = 0.5 * (f[n - 1] / (n + 1) / (n + 1) + u_old[n - 2] + 1.0);
}


int main(int argc, char** argv) {
  // Read arguments
  long n = atoi(argv[1]);
```

```
long max_iters = 5000;
if (argc >= 4) {
  max_iters = atoi(argv[3]);
}
bool is_jacobi = (strcmp(argv[2], "Jacobi") == 0);

if (is_jacobi) {
  printf("Using␣method:␣Jacobi\n\n");
} else {
  printf("Using␣method:␣Gauss-Seidel\n\n");
}

// Allocate memory, with either one vector for Gauss-Seidel or two for Jacobi
double* f = (double*) malloc(n * sizeof(double));
double* u_old = (double*) malloc(n * sizeof(double));
double* lhs = (double*) malloc(n * sizeof(double));
double* u_new;
if (is_jacobi) {
  u_new = (double*) malloc(n * sizeof(double));
} else {
  u_new = u_old;
}

// Initialize data
for (long i = 0; i < n; i++) {
  u_old[i] = 0.0;
  u_new[i] = 0.0;
  f[i] = 1.0;
}
MultA(n, u_new, lhs);

double init_res = norm_diff(n, lhs, f);
double this_res = 0.0;

printf("Iteration␣␣␣␣␣Residual␣Norm␣\n");

Timer t;
t.tic();

// Main loop
for (long it = 0; it < max_iters; it++) {
  // Swap u_old and u_new pointers (in case of Jacobi method)
  double *tmp = u_old;
  u_old = u_new;
  u_new = tmp;

  SolverStep(n, f, u_old, u_new);
  MultA(n, u_new, lhs);
  this_res = norm_diff(n, lhs, f);
  printf("%9d␣␣␣␣␣%5.6f\n", (int) it, this_res);

  if (init_res > 1e6 * this_res) {
    printf("Reached␣stopping␣condition;␣terminating.\n");

    break;
  }
}

double time = t.toc();
```

```
  printf("\nCumulative␣residual␣shrinkage␣factor:␣%f\n", init_res / this_res);
  printf("\nTime␣(seconds):␣%f\n", time);

  // Cleanup
  free(f);
  free(u_old);
  free(lhs);
  if (is_jacobi) {
    free(u_new);
  }

  return 0;
}
```