All-Pairs Shortest Paths

David Bindel

2015-10-19

Introduction

The Floyd-Warshall Algorithm

The Floyd-Warshall algorithm for computing all pairwise shortest path lenghs in a graph has a computational pattern much like the one for Gaussian elimination. There is a closely related algorithm which is slightly more expensive – $O(n^3 \log n)$ time in general rather than the $O(n^3)$ time required by Floyd-Warshall – but which looks very much like matrix multiplication. In this assignment, you will analyze the performance of a reference OpenMP implementation of this method, and then implement and analyze your own version using MPI.

As usual, you are allowed to use any references that you find, with appropriate citations. I know that people have worked on fast Floyd-Warshall on GPUs; you may also find prior work from when I taught the class in 2011 and used this assignment!

Your mission

You are provided with a reference OpenMP implementation (`path.c`). For this assignment, you should attempt three tasks:

1. *Profiling*: The current code is not particularly tuned, and there are surely some bottlenecks. Profile the computation and determine what parts of the code are slowest. I encourage you to use profiling tools (e.g. VTune Amplifier), but you may also manually instrument the code with timers.
2. *Parallelization*: The current code is parallelized with OpenMP. You should also parallelize your code using MPI, and study the speedup versus number of processors on both the main cores on the nodes and on the Xeon Phi boards. Set up both strong and weak scaling studies, varying the number of threads/processes you employ.

3. *Tuning*: You should tune your code in order to get it to run as fast as possible. For tuning, you may focus on either the OpenMP or the MPI version of the code. The computational pattern is much like that of parallel Gaussian elimination, and in addition to tuning the parallelism, I encourage you to use the tools you learned about in matrix multiply (vectorization, blocking).

The primary deliverable for your project is a report that describes your performance experiments and attempts at tuning, along with what you learned about things that did or did not work. Good things for the report include:

- Profiling results
- Speedup plots and scaled speedup plots
- Performance models that predict speedup

In addition, you should also provide the code, and ideally scripts that make it simple to reproduce any performance experiments you've run.

Logistical notes

Timeline

As with the previous assignment, this assignment involves two stages. By Nov 3, you should submit your initial report (and code) for peer review; reviews are due by Nov 5. Final reports are due one week later (Nov 12). I hope this project is more straightforward than the shallow water equation, so that many of you will be able to wrap up early.

Peer review logistics

Since the first assignment, GitHub has added a feature to attach PDF files to issues and pull request comments. You should take advantage of this feature to submit your review as a comment on the pull request for the group you are reviewing. You should still look at the codes from the other groups, though!

Notes on MPI on the Phi boards

I have succeeded in running MPI jobs on the Phi boards, but it seems to take quite a while for jobs to start. We have also had some difficulties getting authentication working properly, and it's possible that you will run into hiccups. Please give it a try, but if you start running into trouble with MPI on the Phi, ask questions early and often – on Piazza, so we can all figure it out together!

The basic recurrence

At the heart of the method is the following basic recurrence. If l_{ij}^s represents the length of the shortest path from i to j that can be attained in at most 2^s steps, then

$$l_{ij}^{s+1} = \min_k \{l_{ik}^s + l_{kj}^2\}.$$

That is, the shortest path of at most 2^{s+1} hops that connects i to j consists of two segments of length at most 2^s, one from i to k and one from k to j. Compare this with the following formula to compute the entries of the square of a matrix A:

$$a_{ij}^2 = \sum_k a_{ik} a_{kj}.$$

These two formulas are identical, save for the niggling detail that the latter has addition and multiplication where the former has min and addition. But the basic pattern is the same, and all the tricks we learned when discussing matrix multiplication apply – or at least, they apply in principle. I'm actually going to be lazy in the implementation of `square`, which computes one step of this basic recurrence. I'm not trying to do any clever blocking. You may choose to be more clever in your assignment, but it is not required.

The return value for `square` is true if `l` and `lnew` are identical, and false otherwise.

```
int square(int n,               // Number of nodes
           int* restrict l,     // Partial distance at step s
           int* restrict lnew)  // Partial distance at step s+1
{
    int done = 1;
    #pragma omp parallel for shared(l, lnew) reduction(&& : done)
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int lij = lnew[j*n+i];
            for (int k = 0; k < n; ++k) {
                int lik = l[k*n+i];
                int lkj = l[j*n+k];
                if (lik + lkj < lij) {
                    lij = lik+lkj;
                    done = 0;
                }
            }
            lnew[j*n+i] = lij;
        }
    }
    return done;
}
```

The value l_{ij}^0 is almost the same as the (i,j) entry of the adjacency matrix, except for one thing: by convention, the (i,j) entry of the adjacency matrix is zero when there is no edge between i and j; but in this case, we want l_{ij}^0 to be "infinite". It turns out that it is adequate to make l_{ij}^0 longer than the longest possible shortest path; if edges are unweighted, $n+1$ is a fine proxy for "infinite." The functions `infinitize` and `deinfinitize` convert back and forth between the zero-for-no-edge and $n+1$-for-no-edge conventions.

```
static inline void infinitize(int n, int* l)
{
    for (int i = 0; i < n*n; ++i)
        if (l[i] == 0)
            l[i] = n+1;
}

static inline void deinfinitize(int n, int* l)
{
    for (int i = 0; i < n*n; ++i)
        if (l[i] == n+1)
            l[i] = 0;
}
```

Of course, any loop-free path in a graph with n nodes can at most pass theough every node in the graph. Therefore, once $2^s \geq n$, the quantity l_{ij}^s is actually the length of the shortest path of any number of hops. This means we can compute the shortest path lengths for all pairs of nodes in the graph by $\lceil \lg n \rceil$ repeated squaring operations.

The `shortest_path` routine attempts to save a little bit of work by only repeatedly squaring until two successive matrices are the same (as indicated by the return value of the `square` routine).

```
void shortest_paths(int n, int* restrict l)
{
    // Generate l_{ij}^0 from adjacency matrix representation
    infinitize(n, l);
    for (int i = 0; i < n*n; i += n+1)
        l[i] = 0;

    // Repeated squaring until nothing changes
    int* restrict lnew = (int*) calloc(n*n, sizeof(int));
    memcpy(lnew, l, n*n * sizeof(int));
    for (int done = 0; !done; ) {
        done = square(n, l, lnew);
        memcpy(l, lnew, n*n * sizeof(int));
```

```
    }
    free(lnew);
    deinfinitize(n, l);
}
```

The random graph model

Of course, we need to run the shortest path algorithm on something! For the sake of keeping things interesting, let's use a simple random graph model to generate the input data. The $G(n, p)$ model simply includes each possible edge with probability p, drops it otherwise – doesn't get much simpler than that. We use a thread-safe version of the Mersenne twister random number generator in lieu of coin flips.

```
int* gen_graph(int n, double p)
{
    int* l = calloc(n*n, sizeof(int));
    struct mt19937p state;
    sgenrand(10302011UL, &state);
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i)
            l[j*n+i] = (genrand(&state) < p);
        l[j*n+j] = 0;
    }
    return l;
}
```

Result checks

Simple tests are always useful when tuning code, so I have included two of them. Since this computation doesn't involve floating point arithmetic, we should get bitwise identical results from run to run, even if we do optimizations that change the associativity of our computations. The function `fletcher16` computes a simple simple checksum. over the output of the `shortest_paths` routine, which we can then use to quickly tell whether something has gone wrong. The `write_matrix` routine actually writes out a text representation of the matrix, in case we want to load it into MATLAB to compare results.

```
int fletcher16(int* data, int count)
{
    int sum1 = 0;
    int sum2 = 0;
```

```
    for(int index = 0; index < count; ++index) {
          sum1 = (sum1 + data[index]) % 255;
          sum2 = (sum2 + sum1) % 255;
    }
    return (sum2 << 8) | sum1;
}

void write_matrix(const char* fname, int n, int* a)
{
    FILE* fp = fopen(fname, "w+");
    if (fp == NULL) {
        fprintf(stderr, "Could not open output file: %s\n", fname);
        exit(-1);
    }
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j)
            fprintf(fp, "%d ", a[j*n+i]);
        fprintf(fp, "\n");
    }
    fclose(fp);
}
```

The main event

```
const char* usage =
    "path.x -- Parallel all-pairs shortest path on a random graph\n"
    "Flags:\n"
    "  - n -- number of nodes (200)\n"
    "  - p -- probability of including edges (0.05)\n"
    "  - i -- file name where adjacency matrix should be stored (none)\n"
    "  - o -- file name where output matrix should be stored (none)\n";

int main(int argc, char** argv)
{
    int n    = 200;            // Number of nodes
    double p = 0.05;           // Edge probability
    const char* ifname = NULL; // Adjacency matrix file name
    const char* ofname = NULL; // Distance matrix file name

    // Option processing
    extern char* optarg;
    const char* optstring = "hn:d:p:o:i:";
    int c;
    while ((c = getopt(argc, argv, optstring)) != -1) {
```

```
        switch (c) {
        case 'h':
            fprintf(stderr, "%s", usage);
            return -1;
        case 'n': n = atoi(optarg); break;
        case 'p': p = atof(optarg); break;
        case 'o': ofname = optarg;  break;
        case 'i': ifname = optarg;  break;
        }
    }

    // Graph generation + output
    int* l = gen_graph(n, p);
    if (ifname)
        write_matrix(ifname,  n, l);

    // Time the shortest paths code
    double t0 = omp_get_wtime();
    shortest_paths(n, l);
    double t1 = omp_get_wtime();

    printf("== OpenMP with %d threads\n", omp_get_max_threads());
    printf("n:     %d\n", n);
    printf("p:     %g\n", p);
    printf("Time:  %g\n", t1-t0);
    printf("Check: %X\n", fletcher16(l, n*n));

    // Generate output file
    if (ofname)
        write_matrix(ofname, n, l);

    // Clean up
    free(l);
    return 0;
}
```