ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

Approximating Li_2 using a numerical quadrature

- the second homework assignment involved $\mathrm{Li}_2(z)$, defined by

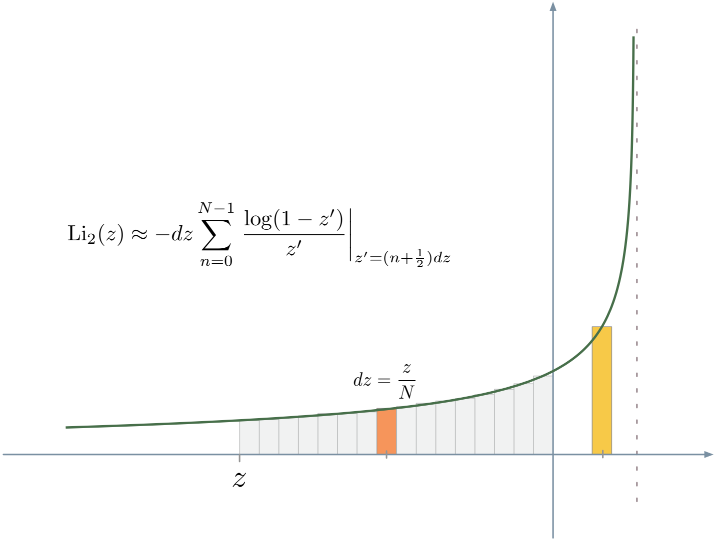
$$\mathrm{Li}_2(z) := -\int_0^z dz' \, \frac{\log(1-z')}{z'}$$

- the assignment asked for approximating this integral using a simple quadrature based on the mid-point rule

$$\mathrm{Li}_2(z) \approx \mathrm{Li}_2(z,N) := -\frac{z}{N} \sum_{n=0}^{N-1} \left. \frac{\log(1-z')}{z'} \right|_{z'=(n+\frac{1}{2})\frac{z}{N}}$$

Quadrature rule

$$\mathrm{Li}_2(z) \approx -dz \sum_{n=0}^{N-1} \frac{\log(1-z')}{z'}\bigg|_{z'=(n+\frac{1}{2})dz}$$



Implementations

- three implementations
  - sequential: to get a feeling for how to convert the algorithm into a functioning program
  - parallel using threads: to walk through the parallelization steps and use `pthreads` to get better performance
  - parallel using `MPI`: to get a feel for how `MPI`-based programs solve the task partitioning problem
- let's walk through composing, building and running
  - on my desktop, and on **`shc.cacr.caltech.edu`**

Sequential implementation - part 1

- the preamble

```
#include <getopt.h> // for getopt and friends
#include <cstdlib> // for atof
#include <cmath> // for the correct abs, log

#include <map>
#include <iostream>
#include <iomanip>
```

- quadrature using the midpoint rule to avoid the singularities

```
// dilog
double dilog(double z, long N) {
   // initialize
   double dx = z/N;
   double x = dx/2;
   double sum = 0;
   // loop
   for (long i=0; i < N; i++) {
      sum += std::log(1-x)/x;
      x += dx;
   }
   // return; don't forget the sign
   return -dx * sum;
}
```

Sequential implementation - part 2

- using the command line to set z and the number of subdivisions N

```
// main program
int main(int argc, char* argv[]) {
   // default values for the command line options
   long N = 1000;
   double z = 1.0;

   // read the command line
   int command;
   while ((command = getopt(argc, argv, "z:N:")) != -1) {
      switch (command) {
      // get the argument of the dilogarithm
      case 'z':
         z = atof(optarg);
         break;
      // get the number of subdivisions
      case 'N':
         N = (long) atof(optarg);
         break;
      }
   }
```

Sequential implementation - part 3

- error checking and computation of the numerical integral

```
   // error checking
   // abort if N < 1
   if (N < 1) {
      std::cout
         << "the number of subdivisions must be positive"
         << std::endl;
      return 0;
   }

   // abort for z > 1 to avoid dealing with the imaginary part
   if (z > 1.0) {
      std::cout << "math domain error: z > 1" << std::endl;
      return 0;
   }

   // compute
   double value = dilog(z, N);
```

Sequential implementation - part 4

- computing the error and printing out the results

```
   // build a naive database of the known dilogarithm values
   const double pi = M_PI;
   std::map<double, double> answers;
   answers[1.0] = pi*pi/6;
   answers[-1.0] = -pi*pi/12;

   // print out the value
   std::cout << "Li2(" << z << ")="
      << std::setprecision(17) << std::endl
      << " computed: " << value << std::endl;
   // check whether we know the right answer
   std::map<double,double>::const_iterator lookup = answers.find(z);
   if (lookup != answers.end()) {
      // and if we do, print it out
      double exact = lookup->second;
      std::cout << " exact: " << exact << std::endl;
      // compute the approximation error and print it out
      double error = std::abs(exact-value)/exact;
      std::cout
         << std::setiosflags(std::ios_base::scientific)
         << "  error: " << error << std::endl;
   }

   return 0;
}
```

Building and running the sequential driver

```
#> g++ dilog-quadrature_sequential
#> dilog-sequential -N 1e9 -z 1.0
Li2(1)=
 computed: 1.6449339414016682
    exact: 1.6449340668482264
    error: 7.6262362595887189 8e-08

real   0m19.885s
user   0m19.877s
sys    0m0.003s
#>
```

Threaded implementation - part 1

- the preamble

```
#include <getopt.h> // for getopt and friends
#include <pthread.h>

#include <cstdio>
#include <cstdlib> // for atof
#include <cmath>

#include <map>
#include <iostream>
#include <iomanip>
```

Threaded implementation - part 2

- private and shared data structures

```
// shared information
struct problem {
   int workers;      // total number of threads
   double dz;        // the width of each subdivision
   double sum;       // storage for the partial computations

   pthread_mutex_t lock; // mutex to control access to the sum
};

// thread specific information
struct context {
   // thread info
   int id;
   pthread_t descriptor;
   // the workload for this thread
   long subdivisions; // number of subdivisions
   double z_low;  // the lower limit of integration
   double partial; // record the partial sum computed by this thread
   // the shared problem information
   problem* info;
};
```

Threaded implementation - part 3

- the coarse grain task

```
// worker
void* worker(void* arg) {
   context* ctxt = (context *) arg;
   // pull the problem information from the thread context
   double dz = ctxt->info->dz;
   double z = ctxt->z_low + dz/2;
   // loop over the subdivisions assigned to this thread
   double sum = 0.0;
   for (long i=0; i < ctxt->subdivisions; i++) {
      sum += std::log(1-z)/z;
      z += dz;
   }
   // multiply by the width of each subdivision and adjust the sign
   sum *= -dz;

   // grab the lock
   pthread_mutex_lock(&(ctxt->info->lock));
   // store the result
   ctxt->info->sum += sum;
   // and release the lock
   pthread_mutex_unlock(&(ctxt->info->lock));

   // all done
   return 0;
}
```