ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2010

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

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

- 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 my solutions
  - on my desktop, on **`mind-meld.cacr.caltech.edu`**, 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);
```

- computing the error and printing out the results

```
   // build a 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_sequential.cc -o dilog_sequential
#> dilog_sequential -N 1e7 -z 1.0
Li2(1)=
 computed: 1.6449340282186398
    exact: 1.6449340668482264
    error: 2.3483972652227854 6e-08
#> time 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;
}
```

Threaded implementation - part 4

- the task master – interface and allocation of storage

```
// driver
double dilog(double z, long N, int threads) {
   // the width of each interval subdivision
   const double dz = z/N;

   // setup the problem context
   problem info;
   info.workers = threads;
   info.dz = dz;
   info.sum = 0.0;
   pthread_mutex_init(&info.lock, 0);

   // and an array to hold the thread contexts
   context thr_info[threads];
   // partition the number of subdivisions
   long nominal_load = N/threads;
```

Threaded implementation - part 5

- the task master – spawning the threads

```
    // spawn the workers
    for (int tid=0; tid<threads; tid++) {
        // store the thread id
        thr_info[tid].id = tid;
        // point to the shared problem info
        thr_info[tid].info = &info;

        // compute the starting point of the partial integral
        thr_info[tid].z_low = tid*nominal_load*dz;
        // compute the number of subdivisions for this thread
        if (tid == threads - 1) {
            // the last thread gets the leftovers
            thr_info[tid].subdivisions = N - tid*nominal_load;
        } else {
            thr_info[tid].subdivisions = nominal_load;
        }

        // create the thread
        int status = pthread_create(
            &(thr_info[tid].descriptor), 0, worker, &thr_info[tid]);
        if (status) {
            printf("error %d in pthread_create\n", status);
        }
    }
```

Threaded implementation - part 6

- the task master – harvesting the threads and returning the result

```
    // harvest the threads
    for (int tid=0; tid<threads; tid++) {
       pthread_join(thr_info[tid].descriptor, 0);
    }

    // all done
    return info.sum;
}
```

Threaded implementation - part 7

- the main program – reading the command line

```
// main program
int main(int argc, char* argv[]) {
   // default values for the command line options
   long N = 1000;
   double z = 1.0;
   int threads = 8;

   // read the command line
   int command;
   while ((command = getopt(argc, argv, "z:N:t:")) != -1) {
      switch (command) {
      case 'z':
         // get the argument of the dilogarithm
         z = atof(optarg);
         break;
      case 'N':
         // get the number of subdivisions
         N = (long) atof(optarg);
         break;
      case 't':
         // get the numberof threads
         threads = atoi(optarg);
         break;
      }
   }
```

Threaded implementation - part 8

- error checking and the invocation of the task master

```
    // 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, threads);
```

Threaded implementation - part 9

- the task master – printing out the answers

```
    // build a 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 threaded driver

```
#> g++ dilog_threads.cc -o dilog_threads -pthread
#> dilog_threads -N 1e7 -z 1.0 -t 4
Li2(1)=
 computed: 1.6449340301295035
    exact: 1.6449340668482264
    error: 2.2322306827430405 8e-08
#> time dilog_threads -N 1e9 -z 1.0 -t 8
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.7910252031577389 2e-08

real   0m2.803s
user   0m20.693s
sys    0m0.006s
#>
```

MPI implementation - part 1

- the preamble

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

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

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

MPI implementation - part 2

- coarse grain task

```
double dilog(double zprime, long N, int id, int processes) {
   // the width of each interval subdivision
   const double dz = zprime/N;
   // compute the starting point of the partial integral
   const double z_low = id*zprime/processes;
   // partition the number of subdivisions
   long nominal_load = N/processes;
   // the last process gets the leftovers
   if (id == processes - 1) {
      nominal_load = N - id*nominal_load;
   }
   // initialize the partial sum
   double sum = 0.0;
   double z = z_low + dz/2;
   // loop over the subdivisions assigned to this thread
   for (long i=0; i < nominal_load; i++) {
      sum += std::log(1-z)/z;
      z += dz;
   }
   // collect the partial answers from all the processes
   double value;
   MPI_Allreduce(
      &sum, &value, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   // multiply by the width of each subdivision and adjust the sign
   return -dz*value;
}
```

MPI implementation - part 3

- the main program – setting up `MPI`

```
// main program
int main(int argc, char* argv[]) {
   // initialize MPI
   int status = MPI_Init(&argc, &argv);
   if (status != MPI_SUCCESS) {
      std::cout << "error in MPI_Init; aborting..." << std::endl;
      return status;
   }
   // get process information from the world communicator
   int id, processes;
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &processes);
```

MPI implementation - part 4

- reading the command line

```
    // 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) {
       case 'z':
          // get the argument of the dilogarithm
          z = atof(optarg);
          break;
       case 'N':
          // get the number of subdivisions
          N = (long) atof(optarg);
          break;
       }
    }
```

MPI implementation - part 5

- error checking and computation

```
   // error checking
   // abort if N < 1
   if (N < 1) {
      if (id == 0) {
         std::cout
            << "the number of subdivisions must be positive"
            << std::endl;
      }
      MPI_Finalize();
      return 0;
   }
   // abort for z > 1 to avoid dealing with the imaginary part
   if (z > 1.0) {
      if (id == 0) {
         std::cout << "math domain error: z > 1" << std::endl;
      }
      MPI_Finalize();
      return 0;
   }
   // compute
   double value = dilog(z, N, id, processes);
   if (id != 0) { // let all but processor 0 die
      // shut down MPI
      MPI_Finalize();
      return 0;
   }
```

MPI implementation - part 6

- printing out the results

```
    // build a 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;
    std::cout << " 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;
    }

    // shut down MPI
    MPI_Finalize();
    return 0;
}
```

Building and running the MPI driver

- on my desktop, or **mind-meld.cacr.caltech.edu**
  - where there is no queue manager

```
#> mpic++ dilog_mpi.cc -o dilog_mpi -lmpi_cxx -lmpi
#> mpirun -np 4 dilog_mpi -N 1e7 -z 1.0
Li2(1)=
 computed: 1.6449340301295035
    exact: 1.6449340668482264
    error: 2.23223068274304058e-08
#> time mpirun -np 8 dilog_mpi -N 1e9 -z 1.0
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.79102520315773892e-08

real   0m3.697s
user   0m0.018s
sys    0m0.015s
#>
```

Running the MPI driver on a shared resource

- on `shc.cacr.caltech.edu` there is a queue manager
  - don't use `mpirun`: you are running on the head node
  - instead, request a dedicated node

```
# shc-a> mpic++ dilog_mpi.cc -o dilog_mpi
# shc-a> qsub -I -l nodes=1:core8 -l walltime=0:15:00
qsub: waiting for job 105059.mistress to start
qsub: job 105059.mistress ready
Logging in as aivazis on shc168, a Linux-2.x_x86_64 system
 setting up: (environment) (aliases) (machines) (tools: Linux-2.x_x86_64)
# shc168> time mpirun -np 8 dilog_mpi -N 1e9 -z 1.0
Li2(1)=
 computed: 1.6449340044883614
    exact: 1.6449340668482264
    error: 3.7910252031577389 2e-08

real  0m10.501s
user  1m14.642s
sys   0m0.273s
# shc168> exit
logout
qsub: job 105059.mistress completed
# shc-a>
```