

ACM/CS 114

Parallel algorithms for scientific applications

Michael A. G. Aivázis

California Institute of Technology

Winter 2010

Functional decomposition

- ▶ *functional decomposition* determines the fine grain parallel tasks by partitioning the problem into semi-independent tasks that can be executed in parallel
- ▶ our numerical integration examples fall in this category
 - ▶ partitioning identified the finest grain work unit as the evaluation of the integrand; no need for fancy domain decomposition
 - ▶ little or no communication/synchronization is required among the tasks, i.e. they are embarrassingly parallel
 - ▶ coarsening consists of grouping fine grain tasks into larger work units in a straight forward manner
 - ▶ the mapping of the coarse grain tasks onto processing units is trivial
- ▶ in general, the computations involved in carrying out the coarse tasks are computationally equivalent
 - ▶ the computation is *self balancing*
 - ▶ or there is no need for sophisticated load balancing
- ▶ scalability and parallel efficiency are determined by the particulars of the problem, such as inherent limitations on the largest problem size of interest

Monte Carlo integration

- ▶ let f be sufficiently well behaved in a region $\Omega \subset \mathbb{R}^n$ and consider the integral

$$I_{\Omega}(f) = \int_{\Omega} dx f \quad (1)$$

- ▶ the *Monte Carlo* method approximates the value of the integral in Eq. 1 by sampling f at random points in Ω
- ▶ let X_N be such a sample of N points; then the Monte Carlo estimate is given by

$$I_{\Omega}(f; X_N) = \Omega \cdot \langle f \rangle = \Omega \frac{1}{N} \sum_{x \in X_N} f(x) \quad (2)$$

where $\langle f \rangle$ is the sample mean of f , and Ω is used as a shorthand for the volume of the integration region.

- ▶ the approximation error falls like $1/\sqrt{N}$
 - ▶ rather slow
 - ▶ but dimension independent!

Implementation strategy

- ▶ computer implementations require a pseudo-random number generator to build the sample
- ▶ most generators return numbers in $(0, 1)$ so
 - ▶ find a box B that contains Ω
 - ▶ generate n numbers to build a point in the unit \mathbb{R}^n cube
 - ▶ stretch and translate the unit cube onto B
- ▶ the integration is restricted to Ω by introducing

$$\Theta_{\Omega} = \begin{cases} 1 & x \in \Omega \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

to get

$$I_{\Omega}(f) = \int_B dx \Theta_{\Omega} f \quad (4)$$

Recasting Monte Carlo integration

- ▶ there are now two classes of points in the sample X_N
 - ▶ those in Ω
 - ▶ and the rest
- ▶ let \tilde{N} be the number of sample points in Ω ; Eq. 2 becomes

$$I_{\Omega}(f; X_N) = \Omega \frac{1}{\tilde{N}} \sum_{x \in X_{\tilde{N}}} f(x) \quad (5)$$

- ▶ let B be the volume of the sampling box; observe that the volume of the integration region can be approximated by

$$\Omega = \frac{\tilde{N}}{N} B \quad (6)$$

and the sum over the points $x \in X_{\tilde{N}}$ can be extended to the entire sample X_N by using the filter Θ_{Ω}

$$I_{\Omega}(f; X_N) = B \frac{1}{N} \sum_{x \in X_N} \Theta_{\Omega} f(x) \quad (7)$$

Requirements

- ▶ to summarize, the Monte Carlo approximation is computed using

$$I_{\Omega}(f; X_N) = B \frac{1}{N} \sum_{x \in X_N} \Theta_{\Omega} f(x) \quad (8)$$

- ▶ using
 - ▶ an implementation of the function f to be integrated over Ω
 - ▶ an n -dimensional box B that contains Ω
 - ▶ a good pseudo-random number generator to build the sample $X_N \in B$
 - ▶ a routine to test points $x \in X_N$ and return `false` if they are exterior to Ω and `true` otherwise
- ▶ to sum the values of the integrand on points interior to Ω , and scale by the volume of the bounding box B over the sample size N
- ▶ essentially a reduction, similar to our other examples
 - ▶ should be straightforward to implement in parallel
 - ▶ see homework assignment

Pseudo-random number generators

- ▶ essential when solving problems using stochastic methods
- ▶ most generators are terrible, e.g. `libc`
 - ▶ always check that you are getting the statistics you expect
 - ▶ `/dev/random` is better, but not portable and produces integers only
- ▶ get the GNU scientific library
 - ▶ from <http://www.gnu.org/software/gsl>, or your OS distribution
 - ▶ broad scope, extensive documentation
 - ▶ thread safe
 - ▶ pick RANLUX or something similar
- ▶ there are algorithms that use uniformly distributed random numbers to generate numbers of any distribution function

Monte Carlo estimate of π

```
1 #include <cmath>
2 #include <iostream>
3 #include <gsl/gsl_rng.h>
4
5 int main(int, char*[]) {
6     // the number of points in the sample
7     const long N = (long) 1.0e7;
8     // point counters
9     long interiorPoints = 0, totalPoints = 0;
10
11     // create the random number generator
12     gsl_rng * generator = gsl_rng_alloc(gsl_rng_ranlxs2);
13
14     // integrate by sampling some number of times
15     for (long i=0; i<N; ++i) {
16         // create a random point
17         double x = gsl_rng_uniform(generator);
18         double y = gsl_rng_uniform(generator);
19         // check whether it is inside the unit quarter circle
20         if ((x*x + y*y) <= 1.0) { // no need to waste time computing the square root
21             // update the interior point counter
22             interiorPoints++;
23         }
24         // update the total number of points
25         totalPoints++;
26     }
27
28     // print the results
29     std::cout << "pi: " << 4.*((double)interiorPoints)/totalPoints << std::endl;
30
31     return 0;
32 }
```


Functional decomposition and load balancing

- ▶ *load balancing* refers to a class of algorithms that attempt to provide optimal or near optimal solutions to the *task scheduling* problem
 - ▶ enormous and diverse literature
 - ▶ broad applicability
 - ▶ computer systems: operating systems, parallel computing, distributed computing
 - ▶ theoretical computer science
 - ▶ operations research
 - ▶ and many other application domains, perhaps disguised
- ▶ *scheduling* is a closely related problem: determine the order in which a set of tasks should run
- ▶ problems that parallelize effectively using functional decomposition tend to be *self scheduling* and *self balancing*
- ▶ abstract the essentials of the approach and construct a load balancing technique
 - ▶ static and semi-static load balancing
 - ▶ self scheduling
 - ▶ distributed task queues

Overview of load balancing

- ▶ information relevant to work distribution
 - ▶ number of tasks: is it fixed or are tasks added as the work progresses?
 - ▶ task costs: when is the cost known? what is the cost distribution among tasks?
 - ▶ task inter-dependencies: can they be run in any order? when do we find out?
 - ▶ locality: should some tasks be scheduled close to each other, i.e. on the same processor, or a nearby one? when does this information become available?
- ▶ there is a spectrum of available solutions depending on how information about task details becomes available
 - ▶ static: all necessary information is available initially
 - ▶ off-line algorithms: run before any real computation starts
 - ▶ semi-static: some information available, context changes slowly
 - ▶ off-line algorithms produce acceptable results in most cases
 - ▶ dynamic: little or no information is available at the outset
 - ▶ on-line algorithms, based on code instrumentation to enable decisions

Static and semi-static load balancing

- ▶ common static cases:
 - ▶ dense matrix algorithms: LU factorizations, etc.
 - ▶ most computations on a regular grid: FFT
 - ▶ sparse matrix-vector multiplication: when graph partitioned
- ▶ common semi-static cases:
 - ▶ particle and particle-in-cell methods
 - ▶ where the main problem is locality as particles move from one cell to another
 - ▶ computations structured as tree traversals
 - ▶ dynamic grids that change slowly, e.g. after many time steps

Self scheduling

- ▶ task manager:
 - ▶ maintain a central pool of tasks that are ready to be scheduled
 - ▶ once a processor completes its current task, assign it a new one from the pool
 - ▶ if a computation of a task generates more, add them to the pool
- ▶ works well when
 - ▶ tasks have no dependencies on each other
 - ▶ tasks are very weakly inter-dependent (but not studied extensively)
 - ▶ task cost is not known
 - ▶ locality is not important
- ▶ do not schedule the fine grain parallel tasks directly; coarsen first
 - ▶ larger grains reduce the task queue management overhead
 - ▶ smaller grains even out the finish times
- ▶ variations:
 - ▶ fixed grain size: like our quadrature implementation
 - ▶ guided self scheduling: start out with coarse grains and refine as you approach the end of the queue
 - ▶ distributed task queues: details and an important use case next time