Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: Data Parallel Computation

Tasks such as the palindromic word finder program, as well as many scientific computations, can be solved using a *data parallel* programming style. A data parallel program is one in which each process executes the same actions concurrently, but on different parts of shared data.

Contrast this with a *task parallel* approach, where different processes each perform a different step of the computation on the same data. This corresponds to the pipeline model mentioned in Quinn Chapter 1.

An important consideration in a data parallel computation is *load balancing* – making sure that each process/thread has about the same amount of work to do. Otherwise, some would finish before others, possibly leaving available processors idle while other processors continue to work. Load balancing will be an important topic throughout the course. Parallel efficiency and scalability of a data parallel computation will be highly dependent on a good load balance.

This is our first real example of *thread/process synchronization*, which is the main reason parallelism is so hard. These examples were contrived to "encourage" the problem to show up and to emphasize the overhead of the solution, but in a real problem, the interference may be very subtle and show itself only very rarely.

Bag of Tasks Paradigm

One specific way of deciding which processes/threads do the operations on which parts of the data is the *bag of tasks*. In this case, each thread/process is a *worker* that finds a *task* (unit of work) to do (from the "bag"), does it, then goes back for more:

```
while (true) {
  // get a task from the bag
  if (no tasks remain) break;
  //execute the task
}
```

A nice feature of this approach is that load balancing comes for free, as long as we have more tasks than workers. Even if some tasks cost more than others, or some workers work more slowly than others, any available work can be passed out to the first available worker until no tasks remain.

Back to our matrix multiplication example, we can break up the computation into a bag of tasks. We'll choose a fine-grained parallelization, where the computation of each entry is one of our tasks.

See: `/cluster/examples/matmult_bagoftasks`

Run this on a four-processor node. You should see that it is still pretty slow. Perhaps the granularity of the computation is too small – too much time picking out a task, not enough time doing it. We created 562,500 tasks. This means we have to acquire the lock 562,500 times. How long does this take to run?

We can easily break up our computation by row or column of the result matrix, as well. Here is a row-wise decomposition:

See: `/cluster/examples/matmult_smallbagoftasks`

You should find that this is much more efficient! We coarsened the parallelism, but kept it fine enough that all of our threads could keep busy. We still had 750 tasks in the bag. How long does this take to run?

Explicit Domain Decomposition

If we could improve things significantly by coarsening the parallelism in the bag of tasks, perhaps we can do even better by dividing up the entire computation ahead of time to avoid any selection from the bag of tasks whatsoever.

With the matrix-matrix multiply example, this is easy enough – we can just give `SIZE/numworkers` rows to each thread, they all go off and compute, and they'll all finish in about the same amount of time:

See: `/cluster/examples/matmult_decomp`

Some things to notice about this example:

- During the setup, we compute the range of rows that each thread will be responsible for. We can't simply give every thread the same number of rows, in case the number of threads does not divide the matrix size evenly. The computation as shown also guarantees no more than a one-row imbalance.
- We need to tell each thread its range of rows to compute. But thread functions are restricted to a single parameter, of type `void *`. We can use this pointer to pass in anything we want, by putting all of the parameters into a structure.

  Notice that each thread needs its own copy of this structure – we can't just create a single copy, send it to `pthread_create()`, change the values, and use it again. Why?

Explicit domain decomposition works out well in this example, since there's an easy way to break it up (by rows), and each row's computation is equal in cost.

Is there any advantage to breaking it down by columns instead? How about, in the case of 4 threads, into quadrants?

In more complicated examples, load balancing with a domain decomposition may be more difficult. We will consider such examples later in the semester.

Divide and Conquer or Recursive Parallelism

A third major approach to data parallel computation is to take the divide and conquer approach of many of your favorite algorithms, but instead of taking each subproblem and solving one after the other, we solve the subproblems concurrently, with multiple threads/processes. Recursive calls can be made concurrently only if the calls write to disjoint parts of the program's data.

First, consider a parallel summation. We have an array of size n and we want to compute the sum. Sequentially, this takes $n-1$ steps, but if we can do things in parallel, we can reduce this to $\log_2 n$ steps. How would you approach this? How many processors do you think you could make use of?

An example of a more interesting computation that we might consider parallelizing using recursive parallelism is the *adaptive quadrature* problem. Quadrature can be used to approximate the value of an integral of a function $f(x)$ from a to b (the area between $f(x)$ and the x-axis in the interval $[a, b]$) by a sum of approximations of the area of smaller subintervals. Each subinterval may be approximated by a trapezoidal rule.

The straightforward (non-adaptive) approach uses fixed width subintervals, and works like this:

```
double f_left = f(a), f_right, area = 0.0;
double width = (b-a) / NUM_INTERVALS;
for (x=a+width; x<=b; x+=width) {
  f_right = f(x);
  area += (f_left+f_right) * width / 2;
  f_left = f_right;
}
```

The more subintervals, the more accurate the approximation.

We can parallelize this in a straightforward fashion. Since all of the intervals can be computed independently, we can break up the intervals among our threads, each computes the areas of its own intervals, and we combine the results at the end.

However, the integrals of some functions are much more difficult to approximate than others. In areas of larger curvature, larger subintervals will lead to larger approximation errors. But adding lots of subintervals everywhere will increase computational cost, even for those parts of our function where larger intervals would have been sufficient.

If we allow different sized subintervals in different parts of the large interval, we can get the same accuracy, but with much less work. We can compute appropriately-sized intervals *adaptively* with a routine such as this:

```
double adapt_quad(double left, double right,
            double f_left, double f_right,
            double lr_area) {
  double mid = (left + right) / 2;
  double f_mid = f(mid);
  double l_area = (f_left+f_mid) * (mid-left) / 2;
  double r_area = (f_mid+f_right) * (right-mid) / 2;
```

```
    if (abs((l_area+r_area) - lr_area) > EPSILON) {
       // recurse to integrate both halves
       l_area = adapt_quad(left, mid, f_left, f_mid, l_area);
       r_area = adapt_quad(mid, right, f_mid, f_right, r_area);
    }
    return (l_area + r_area);
  }
```

Which is called initially as:

```
area = adapt_quad(a, b, f(a), f(b), (f(a)+f(b))*(b-a)/2);
```

This procedure starts with an approximation using a single subinterval, then recursively refines the approximation until it is determined that a recursive step has not improved the approximation enough to justify the next recursive step. This happens when the difference between the new refined approximation and the previous approximation fall below the threshold `EPSILON`. At each recursive step, we divide the current subinterval in half.

A simple program to compute the integral of a function using adaptive quadrature:

See: `/cluster/examples/adapt_quad`

How can we think about parallelizing this? If we want to take the recursive parallelism approach, the place to look is the recursive calls.

Are these calls independent? Yes, except for the counter of the number of function calls, but that is only to get an feel for how much work the computation is doing, and can be eliminated. Each recursive call operates only with its formal parameters and local variables. So we can execute the recursive calls concurrently. We just need to make sure both calls finish before we return the values.

Suppose we create a new thread for each recursive calls. That's a lot of threads! Once we get much beyond the number of processors available, extra threads will not add much besides extra overhead. To program this efficiently, we would need to stop creating new threads after a few steps and instead make regular recursive function calls.

Other Approaches to Adaptive Quadrature

What about tackling this problem with other approaches to parallelism? Bag of tasks? Explicit decomposition? How hard would it be to program each of these? What efficiency concerns arise?

A few things to consider for the bag of tasks approach:

- We don't know all the tasks at the start! Each worker would grab a task from the bag, but may generate two new tasks!
- How do we decide when we're done? Just because the bag of tasks is empty when a worker looks doesn't mean than another worker won't put a new task into the bag.

- How do we accumulate the result? We can't "wait" for the tasks to compute subintervals to finish, since they just get into the bag of tasks. We need to accumulate the result only when we get to a task that doesn't generate two new tasks.