Simulated Annealing, an exploration of algorithm modification for the purpose of parallelization

Module 4.10

Simulated Annealing

- The simulated annealing algorithm is known as a method for finding the optimum of a multi-valued function $f(\vec{x})$. The process proceeds by having an "annealer" which is able to take a random walk in $\vec{x}$, guided such that steps that reduce the value of $f(\vec{x})$ are preferred. The degree to which steps that increase $f(\vec{x})$ are allowed are controlled by a "temperature" parameter T which is gradually reduced over the course of the minimization.
- The classic algorithm involves a real valued function of real valued dependent variables. Steps are taken with some step size in that space, which might be done one dimension at a time or might be done in all dimensions at once. Each step is subjected to a Metropolis Hasting's algorithm, with an acceptance probability of
  - $a(\vec{x}_{new}, \vec{x}_{old}) = \max\left[1, \exp\left(\frac{f(\vec{x}_{new}) - f(\vec{x}_{old})}{T}\right)\right]$
- The step size is adjusted automatically in batches of size n_{batch}, so that the fraction of accepted steps is within some preset threshold of a desired ratio (typically $50\% \pm 10\%$).
- When this has been achieved, the temperature is then allowed to "cool" by a small amount, and the random walk proceeds iteratively until the temperature and step size are both small.

Serial Code Overview

- The sa.c code provided has a serial implementation of a simulated annealing optimization. Sample functions are provided for optimization, a simple 2D function with multiple minimums, and an extension of the same test function to N dimensions. An additional delay function is added to simulate the effect of optimizing a problem which requires substantially more cpu time per function call.

- The main routine contains a primary iteration loop on SA_check_temperature.

```
while(!SA_check_temperature(&model,funcN,NULL)) {
        // while not converged
           // while not at thermal equilibrium
              // adjust step
           // reduce temperature
    }
```

Serial Code Overview

- SA_check_temperature, in return, runs a batch of random walks to determine temperature convergence. It does this using the method SA_adjust_step.

```
probe = SA_adjust_step(model,func,opts);
   if(probe==0) {
      // thermal equilibrium reached, drop temperature
      model->temp*= model->cooling_factor;
      if(model->step<model->epsilon) {
         // if step size small at thermal equil, converge
         return 1;
      }
   }
```

Serial Code Overview

- SA_adjust_step, in turn, runs a batch of steps and determines the number of times a step has been successful.

```
for(iter=0;iter<model->ntrial;iter++) {
        count_success += SA_step(model,func,opts);
    }

        ... (additional logic related to use of count_success)
```

- The number of successful steps is used to determine whether or how to adjust the step size.

Serial Code Overview

- A single step is calculated in the SA_step method, with state information on the process passed in the model structure.
- Note that nowhere in this process is there any clear concurrency--this is fundamentally tracking of a single random walk process, with loop carried dependency from one step to another.

```
int SA_step(SAstruct * model, double func(int, double *, void *), void * opts) {
        …
  // pick a random step;
  for(i=0;i<model->n;i++) {
     model->dx[i] = drand_range(-model->step,model->step);
  }
 // try out new value
  for(i=0;i<model->n;i++) {
     model->trial_x[i] = model->x[i] + model->dx[i];
  }
  trial_energy = func(model->n,model->trial_x,opts);
  model->num_func_calls++;
  double deltaE = trial_energy-model->energy;
  if(deltaE<0) {
     …
   } else {
     …
  }
```

Ensemble Based Simulated Annealing

- A method that is used for parallelization of the SA algorithm is to modify the algorithm to introduce concurrency in the determination of step size. Instead of a single random walker moving n_{trials} steps, multiple walkers use fewer steps, with the total number of steps maintained between the walkers. Each random walk process is shorter, and at some point a choice must be made as to which of the walkers will continue to the next iteration, so the total number of steps per convergence used is less--this potentially represents a different algorithm.
- This modified algorithm is Ensemble-Based Simulated Annealing (EBSA). It parallelizes a single random walk that is n_{batch} long and replaces it with n_{thread} random walks that are $n_{batch}/n_{threads}$ long.

Student Activity

1. Compile and run sa.c. Vary the value of N, and investigate the total running time and number of iterations required for convergence as a function of the dimensionality of the problem. How does the number of iterations required to converge change with the number of free parameters?
2. Looking at the two key loops in the code, the first being the while loop in the main routine and the second being the for loop in SA_adjust step, do you see any concurrency that can be described? Discuss the loop carried dependencies present in both loops and throughout the SA algorithm.

Student Instructions

3. List at least 3 concerns regarding how this algorithmic change might affect convergence, and regarding how one would need to implement a change of multiple annealers instead of a single annealer in the SA_adjust_step method.
4. Compare the ebsa.c code to the sa.c code. Note the change of random number generator from rand() to drand48_r(). Look up the definition of drand48_r, and describe why this change is necessary. Why is it necessary to keep an array of state variables for random number generators?
5. Compare the ebsa.c code to the sa.c code. Note the use of a global memory structure and copy routine. Describe why you can not simply declare the memory structure to be private.
6. Compile and run ebsa.c, and compare ebsa.c to sa.c for equivalent problems. Do this for varying values of N. Consider also changing the artificial delay in funcN. How does the parallel solution of ebsa.c compare to sa.c? Support your answer with simulation results.