Lecture 19: Search Problems

CS178: Programming Parallel and Distributed Systems

April 11, 2001
Steven P. Reiss

I. Overview

A. Last time we looked at n-body problems

B. This time I want to start by continuing this

C. And then move on to search problems

II. N-Body Communication

A. Last time I showed how to do n-body simulations

1. **However the algorithm I showed you spends 80% of its time waiting for communication rather than computing**
2. **Can we do better**

B. Recall the algorithm that is used

1. **Divide the nodes among processors using a tree**
2. **Have processors compute their tree weights**
3. **Send all to the root**
4. **Root broadcasts to all processors**
5. **Compute next positions using update tree**

C. One idea is to allow communication to overlap

1. **Here we apply one of the principles we used before -- pipelining**
2. **Lets line the processors up and assume each has a subtree**
   a) Then rather then sending their tree to the root, suppose they just send it to next neighbor
   b) Neighbor computes its nodes against that tree and then forwards the tree to the subsequent neighbor

c) After p cycles, all computed; moreover communication can occur during computation using nonblocking sends

d) Could also use multiple threads to handle the I/O overlap

D. Other ideas?

III. Search problems

A. What other problems are known difficult & important

1. NP-complete problems

a) Traveling salesman problem

b) Facilities layout problem

c) Integer programming

2. Search problems

a) Matching up gene fragments

b) Game playing

c) Picture/speech/language recognition

d) Robotics

e) Planning (job allocation, etc.)

f) Chip layout

3. What do these have in common

a) They typically have a solution space and an evaluation function

b) The problem can be translated into one of moving through the solution space to finding the maximum value of the evaluation function

4. Consider a particular problem for now

a) Simple instance of the TSP

B. Techniques that are used here

1. Brute force techniques to find the best solution

2. Approximation algorithms

a) These work well for some problems, not for others

3. Restricted searches

a) Look through a portion of the search space in an intelligent order

b) Best-first search

c) Branch and bound

d) Successive refinement

4. Dynamic programming

a) Caching intermediate results that are reused

b) Works for some problems, not NP-complete ones

5. Local search algorithms

a) Start at a given point, find best position from there using local changes

b) This is effectively hill-climbing (gradiant methods)

c) Actually need to avoid local optimums

(1) Simulated annealing

(2) Multiple starting points

6. Genetic algorithms

a) These are an alternative way of navigating the search space

C. Lets look at these in turn on our sample problem

IV. Tree search methods

A. Here we want to do a tree search

1. Each node gives the options

2. Tree can be quite large

3. Show part of the tree for sample problem

4. Note that this can be done without recursion using a list of nodes to search

B. Typical approach here is depth-first search

1. Recall how that works

2. You can do better if you can bound the cost of a node

a) Evaluation function for the tree node thus far

b) Minimum cost estimation for building the remainder

c) This gives a lower bound on the cost for the node

d) If this is worse than the best known solution, we can ignore the node and all its subtrees

e) This is Branch and bound search

3. This works best if you find a good solution early

a) Since you have estimated values at each node

b) You can use these to direct the search

c) Look at best potential node next

d) This is Best-first search

e) Can be implemented by using a priority queue for work

C. Parallel depth first search

1. There are trivial ways of parallelizing search

a) Each processor gets a piece of the tree

b) These work in parallel

c) Best solution is optained at the end

2. What are the problems

a) First for branch-and-bound, you need to know the best current solution to optimize search

(1) These need to be shared among processors

(2) But they come up at random times

b) Second workload may not be balanced

3. How are we going to balance workload

a) Two techniques are typically used

b) First, using a work queue

(1) Split the task into small units

(2) Put these tasks on a queue

(3) Each processor takes tasks off the queue, processes them, puts new tasks on the queue, and continues

(4) This proceeds until all work is done

c) We've seen this with multithreading, servers

d) This also works in a MPI-style environment

(1) But it means passing messages to handle the queue

(2) This is a little tricky if requests can be asynchronous

(3) Can tie next request with sending results back

(4) Must send environment to work with as well

e) How might this work here

f) Problem -- contention for the queue

4. Dynamic load balancing

a) Here the processors periodically split there workloads among the other processors

b) This is sometimes done via sequential oversight

c) Done in parallel if possible

D. A dynamic implementation

1. First generate a list of nodes to search, one per process

a) This is done by the root

b) Distribute these to the processes (using MPI_Scatter)

2. Let each process do a partial dfs on this root

a) Maintaining a work queue

```
Initialize worklist
Repeat {
   PAR_DFS(worklist)
   SERVICE_REQUESTS(worklist)
 }
While WORK_REMAINS(worklist)
```

b) PAR_DFS does a partial search

c) SERVICE_REQUESTS handles requests from other processors to do thier work

d) WORK_REMAINS asks other processors for work

3. Partial depth first search

```
count = 0
WHILE !worklist.empty() && count < MAX_WORK DO
   node = worklist.front()
   IF node is a solution THEN
      IF it is a better solution than our best THEN
         Save this solution
         Broadcast the new best solution
      FI
   ELSE IF node is a feasible solution THEN
      Add subnodes of node to the worklist
   FI
   ++count
   Check for broadcast solution
```

```
END
```

a) Note that broadcasting needs to be done with a loop

b) Check for broadcast solution in MPI

(1) MPI_Iprobe(int source,int tag,MPI_Comm comm,
int * flag,MPI_Status * sts)

(2) Checks for incoming message matching source & tag

(3) Source can be MPI_ANY_SOURCE

(4) Tag can be MPI_ANY_TAG

(5) Sets flag to true if there is one, false otherwise

(6) Status tells source and tag

c) This makes the check easy to do

d) Then you can do a MPI_Recv

4. Servicing requests from other processors

```
while WORK_REQUESTS_PENDING() {
   dest = source of work request
   if (worklist has multiple nodes on it) then
      split the worklist in half
      Send one half to dest
   else
      Send REJECT to dest
   end
end
```

a) Work requests pending using MPI_Iprobe

(1) Here is where tags come in handy

5. Handling intermediate checks conditions

```
IF !worklist.empty() THEN RETURN true
request_sent = false;
out_of_work()
WHILE (true) {
   send_all_rejects()
   IF (search_completed()) RETURN false;
   IF (!request_sent) THEN
      dest = new_request()
      send_request(dest)
      request_sent = TRUE
   ELSE if (reply_received(dest,&ok,&worklist)) THEN
      if (ok) return TRUE
      else request_sent = FALSE
   FI
END
```

a) Out_of_work and search_complete are used to detect termination

b) Send_all_rejects sends rejection messages to all processes that are requesting work from us

(1) Needed to avoid deadlocks

c) New_request gets the processor to ask for work from

(1) This could be done randomly

(2) It could be done sequentially

(3) It could be done by using one processor as an arbitrator

d) Send_request sends the request for work

e) Reply_received handles the reply

(1) If work provided, updates worklist and sets ok = true

6. Handling termination detection

a) This is not as easy as it seems

b) One way is to think in terms of energy

c) At start process 0 holds energy = 1.0

d) At initial distribution, each process gets 1/P energy

e) When a process fills a request for work, it divides its energy in half, and gives half to the receiving process

f) When a process runs out of work, it gives its energy back to process 0

g) When process 0 gets all the energy back (and it done), the search is complete

h) Note that you have to worry about precision here

(1) Trick -- use rational arithmetic (a,b) == a/b

V. Genetic Algorithms

A. Basic Concepts

1. Nature does a good job of finding near-optimal solutions to problems through evolution

a) Using lots of time

2. It does this through a variety of mechanisms

a) First selecting parents (survival of the fittest)

b) Then by merging genes from the parents (crossover)

c) Finally by introducing random mutations

3. Generic approach to search tries to mimic this

a) Need to define analogs to fitness, crossover, mutation

b) But then just simulate lots of generations

4. This is useful here because it is easy to parallelize

B. Algorithm

```
generation = 0
setup initial Population(generation)
evaluate Population(generation)
while (not terminationCheck()) {
   ++generation;
   select Parents(generation) from Population(generation-1)
   apply crossover to Parents(generation) to get
         Offspring(generation)
   apply mutation to Offspring(generation) to get
         Population(generation)
   evaluate Population(generation)
end
```

1. Representation

a) For this to work you need to create a compact representation of a solution

(1) Typically a string of bits

b) Must allow the above operations

(1) Random generation of an initial solution

(2) Ability to do consistent crossover of solutions

(3) Ability to apply mutations in consistent ways

c) Must keep legal in some way

2. Example -- suppose we want to find 3 numbers

a) Crossover methods -- single, multiple splits

b) Mutation -- change random bit

3. Example -- how might you to TSP

a) Numbers giving priority

b) Numbers giving next (done with checking for used, find next using rehash methods)

4. Choosing parents

a) Want some bias on fitness, but not too much

b) k-tournaments (choose k at random, pick best)

5. **Termination conditions**
    a) When best changes little
    b) After a fixed number of generations
6. **Variations**
    a) Keep the best parents in the next generation
    b) Population size can be fixed or vary
    c) Mutation rate

C. Parallel algorithm

1. **Much of this can be parallelized**
    a) But not necessarily directly
    b) Don't want to distribute everything each generation
    c) But then nature doesn't either
2. **Work with islands (separated populations) that occassionally intermingle**
    a) Add a migration component to the loop
    b) Migrate best versus random
        (1) Best might put too much pressure on
        (2) Best every k times, random otherwise
    c) This can be applied each time, every k times, or random
3. **Migration to where**
    a) Could broadcast the best items to all other nodes
        (1) Island model
    b) Alternatively, just pass it on to local nodes
        (1) Stepping-stone model

VI. Hill Climbing Algorithms

A. Basic idea

1. **Start at a random solution**
2. **Use local transformations to find a better solution**
    a) Increasing/decreasing values
    b) Interchanging pairs of cities
3. **Choose the local transformation that yields the best solution**

4. **Repeat until a best is found**
    a) This finds a local optimum
5. **This can be done with multiple starting points to find the best local optimum, hoping for global**

B. Parallelizing

1. **Each processor does some random set of starting points**
2. **They can then operate in parallel until they find their optimum**
3. **Then results are compared**

VII. Simulated Annealing

A. This is similar to hill-climbing

1. **But one allows moves that are non-optimal at times**
2. **Probability of non-optimal move depends on temperature**
    a) Starts off high, then decreases
    b) Done randomly
3. **The idea is that you should be able to avoid some local maxima and are more likely to find global optima**

B. Again can be parallelized fairly easily

VIII. Successive Refinement

A. Basic concept

1. **Look at the search space in a coarse way**
    a) Bit-wise, etc.
2. **Do a full search here**
3. **Choose the best solution(s), and do a search**
4. **Essentially the same as above, but with intelligent choice of starting points**

B. Parallelization

1. **Initial grid can be handed out to different processors**
2. **Find best K solutions after these are done**
3. **Then these are redistributed for more refined search**
4. **The process is repeated**