CSCE 5160 Parallel Processing

Project Reports Due on the last day of classes: May 1, 2019
Final Exam: Monday May 6, 2019: 1:30-3:30, F280
Complete Course Evaluations using the new SPOT (Student Perception Of Teaching)

Review

Discrete optimizations

Depth First Algorithms

**Simple backtracking.**
Each time an infeasible solution is found backtrack to previous node and try a different alternative.

**Branch and Bound.** We generate all feasible solutions.
_Even when a solution is found we continue our search._
If a new better solution is found, update but continue

A* and IDA*

Iterative Deepening A* (IDA*)
When the solution trees are very deep, DFS may waste time by going deep in one part of the tree, where no solution exist

So, here we perform DFS to only some depth
If no solution is found, increase the acceptable depth and try again.

Suppose we know how to estimate the cost of a feasible solution from any node already in the tree, then we can decide whether to continue the depth first search or not.

**Best First Search** (not breadth first)
In this method we will try to identify the best potential node to traverse.

For example, suppose it is always better to move to the left in a 8 puzzle we then try to expand that node of the search tree first.
Parallelizing Depth First Search

A process searches all nodes to depth $k$
It then explores only one of sub-trees rooted at level $k$
If $d$ (depth of search) $> 2k$, time required by each process to traverse first $k$ levels of state space tree inconsequential

Disadvantages of allocating one sub-tree per process

- In most cases state space tree is not balanced
- Example: in crossword puzzle problem, some word choices lead to dead ends quicker than others
- Alternative: make sequential search go deeper, so that each process handles many sub-trees (cyclic allocation)
Other methods like Min-Max are used for game programming (for example Chess)

Alternates between your moves and opponents moves
on your move, optimize profit
on opponent’s move minimize loss to you

Note we alternate between maximizing and minimizing since we alternate between our moves that of the opponent

One observation. Parallel Search Overhead

Parallel implementations may do more work because they may search more nodes in the tree

If \( W \) is the work done by one processor, we can equate the work to the number of tree nodes searched \( T_s \).

Likewise \( W_p \) is the work done by \( p \) processor and we an equate this to \( T_p \) as the number of nodes searched

If \( T_p / T_s \) is less than 1; we get superlinear speedup

Otherwise we get less than \( p \) times speedup.

Note this does not include other overheads such as communication and synchronization.

We can still use scalability analyses (Iso-efficiency functions etc)
CSCE 5160 Parallel Processing

Two ways of representing the tree
(b). Push nodes on to a stack
Parent is not kept (placed on closed)

(c). Keep children with parent nodes on together

Nodes are added to Open only if the node is not already in the Open or Closed lists

Once a node is expanded, it is moved to closed list

Node 2 and 6 are explored and removed (or moved to closed). Next node 3 is being explored.

In shared memory, we can keep this stack in shared memory

CSCE 5160 Parallel Processing

In message passing, we need to distribute work or stack to other nodes. We can either send top portion or bottom portion of the stack

Bottom of stack: bigger unexplored tree At cut off, works if search tree is uniform
Also; distribute half of the nodes
We do some initial load distribution
The assigned nodes will grow their stacks (or tree)
When a processor runs out of work it requests one of the other processors for work
Which processor to ask for work?

Asynchronous round robin method:
Each processor is responsible for selecting a target to ask for work.
Each time a request is sent, a new target processor is selected for the next time
It may be possible for more than one processor sending requests to a single target.

Global Round Robin:
A centralized processor determines the target.
All processors send their requests to the centralized processor which selects a target for the request.

Congestion at the central node

Random polling. Each time a processor needs work it selects a processor at random to request for work

How much work to donate? Half splitting and Cut-off depth

Half splitting divides existing work between the the local processor and the requesting processor

Cut of splitting divides work only up to a specified tree depth since the work at lower depths is lighter than that at higher levels

In general, let us assume that a processor with a work of w will keep $\phi w$ work and gives $(1-\phi)w$ to the requesting processor

Note, that successive splits may lead to very small work at a node and no work splitting should occur beyond a threshold value
Now let see how we can analyze the performance of load balancing algorithms.

Note that after receiving a request for work (or distributing work), a processor’s load is reduced by \((1-f)\).

After successive requests, the processor load will eventually drop below the threshold and no more work will be exchanged.

So we need to find out how long this takes to reach the point of no more exchanges.

Let \(V(p)\) be the number of requests generated by \(p\) processors before every processor received at least one request -- \(V(p) \geq p\).

Note after one \(V(p)\) every processor’s work has been reduced by \((1-f)\).
After 2 \(V(p)\), the work has been reduced by \((1-f)^2\) and so on.

If \(f\) is 0.5, after \(\log(W)\) rounds of \(V(p)\) requests, no processor has more than the threshold of work. So, we can use this as the number of communications.

The complexity due to communication = \(O(V(p) \log W)\)

If each communication and exchange takes \(t_{\text{comm}}\), the overhead due to parallelization is \(T_o = t_{\text{comm}} \cdot V(p)^* \log(W)\)

(eq. 11.2, page 487)

Note that efficiency is given by

\[
E = \frac{1}{1 + \frac{T_o}{W}} = \frac{1}{1 + \left(t_{\text{comm}} \cdot V(p)^* \log(W)\right)/W}
\]

So, we can find efficiency and iso-efficiency functions if we can find the expression for \(V(p)^*\) and if we know cost of communication \(t_{\text{comm}}^*\).

\(V(p)^*\) depends on the load balancing scheme
-- Asynchronous round robin, Global round robin or random polling.

It is often very difficult to find exact expressions for \(V(p)^*\) but we will use either worst case or average case values

**Asynchronous Round Robin.** Let us consider the worst case value for \(V(p)^*\).
This happens when all processors send their requests to the same processor at the same time.

So, we will have \(p-1\) requests to one processor (say 0), then we will need another \((p-2)\) requests before two processors receive requests, etc.

So \(V(p)^* = O(p^2)\)
Global Round Robin: All processors receive requests in sequence and thus after \( p \) requests, each processor received one request. So \( V(p) = p \)

Random Polling: The worst case value of \( V(p) \) is unbounded; but we can use average case

Let \( F(i,p) \) represent the state that \( i \) of the \( p \) processors received requests and \((p-i)\) processors have not. So, the next request can go either to a new processor with probability \( [(p-i)/p] \) or to one that already received a request with probability \((i/p)\). If the request is to a new processor then we go from \( F(i,p) \) to \( F(i+1,p) \) state.

How many requests are needed to effect this state change? Remember you probability class

\[
E\left(\frac{1}{p} \cdot \frac{p-i}{p}\right) = \frac{1}{1 - \frac{i}{p}} = \frac{p}{p-1}
\]

Let \( f(i,p) \) designate the expected number of requests to go from \( F(i,p) \) to \( F(p,p) \). Thus we can equate \( V(p) = f(0,p) \)

How do we find \( f(i,p) \) -- we will describe using a recursive equation.

\[
f(i,p) = \left(\frac{i}{p}\right) \cdot \left(1 + f(i,p)\right) + \left(\frac{p-i}{p}\right) \cdot \left(1 + f(i+1,p)\right)
\]

The first term gives the probability of staying in the same state \( F(i,p) \) while the second term gives the probability of going to the state \( F(i+1,p) \)

Solving for \( f(i,p) \) we get

\[
f(i,p) = \frac{p}{p-i} + f(i+1,p)
\]

Not we need to find \( V(p) = f(0,p) \) \( \rightarrow \) going from state zero to \( p \)

\[
V(p) = f(0,p) = \sum_{i=0}^{p} \left(\frac{p}{p-i}\right)
\]

This expression is approximately \((p)^* (1.69 \ln p)\) and we will approximate \( V(p) \) to \( p \log p \).

So far we have solved \( V(p) \) for 3 different types of load balancing schemes.

Now we will try find the communication cost \( t_{\text{comm}} \) for hypercube
We are assuming that the work distribution is in fixed-size messages (this can be modified for more complex analysis).

In a hypercube, the average distance between any pair of processors is $\log p$.
So, $t_{\text{comm}} = O(\log p)$

Since now we know both $t_{\text{comm}}$ and $V(p)$ we can try to solve the iso-efficiency functions.

Note the overhead in parallel computation $T_o = t_{\text{comm}} \cdot V(p) \cdot \log (W)$

Asynchronous Round Robin:
$T_o = O(\log p \cdot (p^2) \cdot \log (W))$

To balance the work with communication -- so that neither communication nor workload imbalances dominate (we are finding iso-efficiency function)

$W = T_o = O(\log p \cdot (p^2) \cdot \log (W))$

Solving for $W$ we get
$W = O( p^2 \log p \cdot \log p + p^2 \log p \cdot \log \log p + p^2 \log p \cdot \log \log W)$

The two $\log \log$ terms are smaller than other terms.
So the iso-efficiency function is dominated by $O(p^2 \log^2 p)$

Global Round Robin. Note $V(p) = O(p)$, so as before we try to solve for $W$;
we have the iso-efficiency function as $O(p \log^2 p)$

This looks better than with Asynchronous round robin. However this does not take into account the "congestion" caused by the use of a single processor to arbitrate the requests.

If we use a PRAM or shared memory, we can think of using a mutex variable called target and all processors read this value and increment it.

In this case, due to contention, we need to add a factor $p$ slow down in work.

So, the iso-efficiency function for Global Round robin is $O(p^2 \log^2 p)$, same as that of Asynchronous round robin.
CSCE 5160 Parallel Processing

For **Random Polling**, $V(p) = O(p \log p)$, and solving for $W$ as we did before and balancing work with communication, the iso-efficiency function is $O(p \log^3 p)$

*So the random polling appears to have the best iso-efficiency value.*

Note that these formulations are based sometimes on the worst case and sometimes on averages. Also, we assumed $t_{comm} = O(\log p)$ and ignored the impact of $t_c, t_w$ which depend on an actual architecture.

Consider Problem 11.2 on page 511. This presents a different load balancing idea.

We have an initial load distribution to $p$ processors.
Each processor maintains a local counter which is initialized to zero.

When a processor becomes idle, it selects two processors $p_i$ and $p_{i+1}$ such that the counter at $p_i$ is greater than that at $p_{i+1}$. And the idle processor sends work request to $p_{i+1}$ and $p_{i+1}$ increments its counter if $p_{i+1}$ received more work distributions than $p_i$.

(note that the numbering of processors is a logical numbering)

---

CSCE 5160 Parallel Processing

How to find the counter values. We can consider constructing a logical tree where the leaf nodes (processors) are ordered according the numbers.

We need $O(\log p)$ to find $p_i$ and $p_{i+1}$.

How do we compute $V(p)$ – number of requests by which all processors receive one request?

Note that if $p_{i+1}$ is selected, its counter is incremented (and likely to make it the same as the counter of $p_i$).
And $p_i$ must have received a request previously.

So $V(p) = p$ – the same as Global Round Robin

What about other variations of these ideas?
CSCE 5160 Parallel Processing

Consider processors in a hypercube.
We can randomly select nodes that are at a distance $i$ from requesting node;
i is varied from 1 to log $p$.

$\#$ nodes at a distance if $i$: that is $i$ bits out of $d$ bits are different

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

As with random polling, we need to find estimated number of requests such that all processors at a
distance $i$ receive a request. Repeat this for different $i$ values

So we have $V(p) = \sum_{i=1}^{d} n_i \log n_i$

$n_i$ is the number of nodes at a distance $i$

Using the simplified equation for random polling for $V(p) = p \log p$

CSCE 5160 Parallel Processing

For example if $p = 16$, $d = 4$

$V(p) = 4 \log 4 + 6 \log 6 + 4 \log 4$

If $p = 32$, $d = 5$

$V(p) = 5 \log 5 + 10 \log 10 + 10 \log 10 + 5 \log 5$

Another issue to consider in parallelizing search algorithms is the detection of termination
How do we find termination (when all processors are idle)?
Simple Token passing algorithms (by Dijkstra).

If we assume that once a processor becomes idle, it stays idle.
$p_i$ sends a token when it becomes idle to $p_{i+1}$.
Token starts with $p_0$ and when the token from the last processor returns to $p_0$ which indicates
that all processors are idle
Dirty and Clean tokens (and dirty and clean processors).

When \( p_0 \) becomes idle, it initiates the termination detection by sending clean token around \( p_0 \) is marked clean.
If \( p_j \) sends work to \( p_i \) and \( j > i \); \( p_j \) is marked dirty
If \( p_i \) has the token (either dirty or clean) and becomes idle, passes token to \( p_{i+1} \).
If \( p_i \) is dirty then a dirty token is sent; \( p_i \) becomes clean
If \( p_i \) is clean then a clean token is sent; \( p_i \) stays clean
If \( p_0 \) receives a clean token and \( p_0 \) itself is clean and idle, termination!

What is the complexity of this algorithm?
If we use simple Dijkstra algorithm: \( O(p) \)
What about the extended algorithm?

The token may have to go around \( p-1 \) times and each time it has to go to \( p \) processors = \( O(p^2) \)