CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 11: March 15

Lecturer: Emery Berger *Scribe: Ed Walters*

This lecture covers multiprogrammed processors and the Hood library.

11.1 MultiProgramming Techniques

- *Static Partitioning*: Static partitioning partitions serial work T_1 (the time it takes a single processor to do all the work) evenly among P lightweight processes. Therefore, if each process runs in parallel, each process performs T_1/P worth of work. This produces a linear speedup. Static partitioning is appropriate when the user is the sole process running on the system, and is often accomplished using shared memory/MPI techniques.

  In terms of performance, static partitioning suffers a sharp decrease in performance when the amount of work exceeds the number of processors. Static partitioning also suffers with an unknown or dynamic number of processors.

- *Multiprogramming*: When another process is concurrently running on the system, the number of processes available P_A will be less than the total P. The most desirable result is to fully utilize all of the available parallelism, i.e. $T = T_1/P_A$. Unfortunately, some of the work may not be fully parallelizeable, resulting in leftover chunks that increase the execution time. Multiprogramming used to be acceptable for use with applications such as scientific programming, but this is no longer the case.

- *Dynamic Scheduling*: This scheme partitions the work into many (potentially millions) threads of small granularity. Generally the scheme is implemented by creating a user-level thread scheduler and library, since pthreads would be prohibitively expensive. If we define the *processor average* P_A as the time average number of processors available for execution (as determined by the kernel), then this scheme aims to achieve an execution time of $T = T_1/P_A$, regardless of kernel scheduling.

11.2 Implementation of Dynamic Scheduling and Hood

- *The DAG Model:* Mulit-threaded communication is often modeled as a **D**irected **A**cyclic **G**raph, where each node represents a single unit of work (instruction) that takes one cycle to execute. The edges represent the dependencies between two units of work, where for an (u, v), u executes, then v. We limit the graph to a single source and a single potential spawn for each node (limiting outdegree to two). Using this model, if our work T_1 is the number of nodes, the maximum speed-up possible T_∞ (the run time on an infinite number of processes) is the length of a longest path through the DAG. Note that a node can only be executed if it is *ready*, i.e. all of its ancestors in the graph have been executed.

- *Hood's Non-blocking work stealer - Theory:* The Hood user-level threads library uses a non-blocking work stealer. As opposed to Cilk, Hood does not use locks, so Hood does not have to worry about situations such as: swapped out locks, or a changing number of processors. Therefore, Hood can provide tighter scheduling guarantees. In theory, the expected value of the execution time is:

$$E[T] = O(T_1/P_A + T_\infty P/P_A)$$

where the kernel is adversarial, and the bound is optimal to a constant factor, i.e. for any $\epsilon > 0$,

$$T = O(T_1/P_A + (T_\infty + lg(1/\epsilon))P/P_A)$$

with probability at least $1 - \epsilon$. In practice, $T \approx T_1/P_A + T_\infty P/P_A$, and furthermore $T \approx T_1/P_A$ whenever P is small compared to the *average parallelism*, T_1/T_∞.

- *Basics of Work Stealing:*

  1. *Deques:* In work stealing, each process maintains a deque of ready threads. A process grabs work by popping a thread from the bottom of the deque and executing it. If a subject thread blocks or terminates, the processor can pop another one. If the processor spawns a thread, it pushes one of the two threads onto the deque bottom and executes the other.
  2. *Stealing:* Stealing occurs when an idle process takes a thread from the top of the deque of a random victim process.
  3. *Non-blocking operations:* Since the Hood deques are non-blocking, atomic load-test-store operations must be used to maintain synchronization. In addition, there will always be a constant number of attempts to any deque op before success. Someone will eventually succeed.

     If a process has completed one steal attempt and is about to immediately attempt another, then the process *yields*. This yield is important to ensure progress because a process can be working and subsequently swapped out. In the worst case, all the remaining processes can be empty, spin-locking, and making steal attempts on empty queues. If yielding is present, then the working processes will not be starved. Without yielding, steal attempts rise dramatically with increasing number of processes, resulting in a precipitous drop in performance if the number of processes exceeds available processors.

- *Scheduling - Lower Bounds:* Using this model, we can obtain lower bounds for execution time. At each time step $i = 1, \ldots, T$ the kernel decides to schedule a specific subset of the P processes, each of which executes a single instruction. At step i, let p_i denote the number of processes scheduled by the kernel.

  We can now define the processor average as $P_A = \frac{1}{T}\sum_{i=1}^{T} p_i$, and the average execution time as $T = \frac{1}{P_A}\sum_{i=1}^{T} p_i$. Therefore, as the processor average increases, the average execution time decreases.

  We also now can see that $T \geq \frac{T_1}{P_A}$, since $\sum_{i=1}^{T} p_i \geq T_1$. In other words, we have to at the very least do all of the work available. We can also get the bound $T \geq T_\infty P/P_A$, because the kernel can just force us to run the critical path information, so $\sum_{i=1}^{T} p_i \geq T_\infty P$.

  In summary, our bounds dictate that there must be at least the critical path lengths (T_∞) worth of steps i where $p_i \neq 0$. For each such step, the kernel can potentially schedule $p_i = P$ processes.

- *Scheduling - Upper Bounds:* We now prove two theorems involving the run time of the non-blocking work stealer:

  Theorem 11.1 *A scheduler is* greedy *if at each step i, the number of nodes executed is* $min(p_i, \texttt{\# of ready nodes})$. *Any greedy schedule in this context has an upper time bound of* $T_1/P_A + T_\infty P/P_A$.

  Proof: Since $T = \frac{1}{P_A}\sum_{i=1}^{T} p_i$, we want to prove that $\frac{1}{P_A}\sum_{i=1}^{T} p_i \leq T_1/P_A + T_\infty P/P_A$, or $\sum_{i=1}^{T} p_i \leq T_1 + T_\infty P$.

  During each step, each scheduled process pays one token, which is placed in one of two buckets: the *work bucket*, which is used when a processes executes a node, or the *idle bucket*, which is used when the process does not execute.

At the end of the execution process, execution will end up with T_1 tokens in the work bucket. On the other hand, here are at most T_∞ steps where a process places a token in the idle bucket, and at each step at most P tokens are placed in the idle bucket.

This provides an upper bound of $T_1 + T_\infty P$.

■

Theorem 11.2 *The non-blocking work stealer runs in expected time* $O(T_1/P_A + T_\infty P/P_A)$.

Proof: If we have S steal attempts, then we can break the work buckets into two different categories. If the process is executing, then it places a token the work bucket as above, finishing with $O(T_1)$ tokens at the end of execution.

Otherwise, the process tries to steal, and therefore puts a token in the *steal bucket*, finishing with $O(S)$ tokens in the steal bucket.

We define an *enabling edge* as an edge in our DAG Graph (u, v) such that the execution of u causes v to be ready. The node u is termed the *designated parent* of v. These edges in turn form an *enabling tree*. Note that the enabling tree is a dynamically determined structure.

Next we (informally) prove the following lemma:

Lemma 11.3 *For any deque, at all times during the execution of the work-stealing algorithm, the parents of the nodes in the deque lie on a root-to-leaf path in the enabling tree.*

Proof: For any process at any time during its execution:

- v_0 is the ready node of the thread being executed.
- $v_1, v_2, \ldots, v_k$ are the ready nodes of the rest of the threads in the process's deque (ordered from bottom to top).
- For $i = 0, 1, \ldots, k$, the node u_i is the designated parent of v_i.

Everything currently in the deque is a child of an element in the enabling tree. Every time an element in the queue is run, and therefore something is put on the queue, you move down the enabling tree. Therefore, since all of the nodes v_i are ordered, so are their corresponding u_i nodes, so each u_i is an ancestor of u_{i-1} in the enabling tree. ■

We can use the above lemma to determine the potentials: at each step i, each ready node u has the potential $\phi_i(u) = 3^{T_\infty - d(u)}$, where $d(u)$ is the depth of u in the enabling tree. The full potential at step i is $\Phi_i = \sum_u (\phi_i(u))$, for all such ready nodes u. These potentials represent the fact that when you attempt to steal something, the potential for stealing anything else decreases by roughly $1/3$. Including the depth of the enabling tree is important because the nodes that are higher on the critical path have much more of an effect on the readiness of subsequent nodes than nodes lower on the critical path. The top-most node itself contributes a constant fraction, with an initial potential of $\Phi_0 = 3^{T_\infty}$.

Each of P steal attempts cause the potential to decrease by a constant fraction, until potential decreases to 0 after $O(T_\infty P)$ steals. Therefore, the expected number of steals is $E[S] = O(T_\infty P)$.

For our description above, we know that the total amount of tokens is $O(T_1 + S) = O(T_1 + T_\infty P)$, since we either put a token in the work bucket or the steal bucket. Therefore, the expected run time is $O(T_1/P_A + T_\infty P/P_A)$.

■

- *Performance:* Since we know our execution time is $T \leq c_1 T_1/P_A + c_2 T_\infty P/P_A$, we can provide a utilization metric $\frac{T_1}{P_A T} \geq \frac{1}{c_1 + c_2 P/(T_1/T_\infty)}$. The ratio $P/(T_1/T_\infty)$ is called the *normalized number of processes.* Therefore, for all applications and inputs, we can provide a lower bound on the utilization based on the normalized number of processes.

  Looking at the utilization figures for the `knary` benchmark and some real applications shows that the model is a decent fit.

  Hood itself also performs admirably - there is no precipitous drop in performance after the number of processors is exceeded by the number of processes. If we vary the number of processes by testing the subject concurrently with the synthetic application `cycler` that randomly spawns processes, then we find that the model also works very well.

We conclude by stating that the non-blocking work stealer provides good performance with well-defined bounds on commodity systems.

11.3 Related Work

11.3.1 Coscheduling

Coscheduling is a technique whereby all of a computation's processes are scheduled in parallel. It is also called *gang scheduling.* It is a very poor choice for certain computation mixes. For example, running two computations on a four processor machine, one with four processes and one with one only, will produce poor results because the poor fit will cause a number of processing slots to be wasted. On the other hand, this solution works well for resource-intensive applications and heavily communicating processes, i.e. applications with blocking processes.

11.3.2 Process Control

With process control, processes are dynamically created and killed by each computation to fit the number of processors available. This solution complements the non-blocking work stealer because process creation and destruction can be tied to its deque contents, and therefore for process control can keep P close to P_A. On the other hand, since the number of processes is so closely tied to a constant, there is less chance to improve performance through latency hiding.