Thread Scheduling for Multiprogrammed Multiprocessors

(Authors: N. Arora, R. Blumofe, C. G. Plaxton)

Geoff Gerfin

Dept of Computer & Information Sciences
University of Delaware
Outline

- Programming Environment Models
  - Problems With Prior Scheduling Methods
- Thread Scheduling Improvements
  - Model of Multithreaded Computation
  - Non-Blocking Work Stealing Algorithm
- Results & Conclusions
• Non-multiprogrammed: Targeting P-fold speedup
  • If there are P processors, then divide a program’s work into P threads, in order to gain P-fold speedup
  • An example: (Program runs in Time: Work/P = Work/4)
Disadvantages of looking for P-fold speedup:

- Most environments are multiprogrammed: multiple processes can occupy the available processors
- Example in a multiprogrammed environment: (Runs in Time: Work/2 → We can do better!)

Sample Program

Sample Program 2
Improving Thread Scheduling

- How do we improve the scheduling?
  - Model multithreaded computation within a program using a directed acyclic graph (\textit{dag})
  - Implement a non-blocking work-stealing algorithm that will create a double-ended queue (\textit{deque}) to hold nodes from the \textit{dag} for each thread of a program
    - Threads pull ready nodes from the bottom of their \textit{deque}
    - Threads with empty \textit{deques} steal nodes from the top of other threads’ \textit{deques}
  - Using a \textit{dag} enables evaluating a program at its lowest level, creating an efficient way to tell which instructions are ready for execution at a given time
Specifications for the dag:

- Each node represents a SINGLE instruction
- Each node in a thread is linked together by edges that define the instruction execution order
- If a node in one thread creates another thread, an edge connects that node to the first node of the created thread
- If an instruction in one thread is dependent on that of another thread, then an edge will connect the two nodes

Example dag:
Additional dag specifications:

- **Ready Nodes** – Only when all parents of a node have been executed can a node be considered ‘ready’. Only ready nodes can be executed.

- The amount of work ($T_1$) for a process is equal to the number of the nodes in the dag.

- The **critical path** ($T_\infty$) of the dag is the length of the longest directed path in the dag.

- The **average parallelism** can be represented by the following ratio: $T_1 / T_\infty$
  - Parallelism within a process will increase if there is a significantly larger amount of work than that defined by the critical path.
Now that we have a model of the computation within a process, how do we improve the scheduling?

- Implement a non-blocking work stealing algorithm
  - Maintain a pool of ready nodes from the dag into a deque for each thread of a program
  - If a pool of ready nodes for a given thread becomes empty, then the thread steals ready nodes from another threads’ deque
  - Once all deques have been depleted, execution is complete.
  - A non-blocking implementation allows for very little overhead when synchronizing the deques of multiple threads
    - Requires the use of atomic instructions provided by the hardware (compare-and-swap, load-linked/store-conditional)
• Work Stealing Implementation (Here, ‘process’ means thread, and ‘thread’ means node)

//Assign root thread to process zero
Thread * assignedThread = NULL;
if (self == processZero)
    assignedThread = rootThread;

//Run scheduling loop
while(!computationDone) {

    // Deque is empty but we may have an assigned thread.
    while (assignedThread != NULL){
        dispatch (assignedThread);                        //Execute until terminate or block.
        assignedThread = self->popBottom();         //Get next thread
    }

    // Deque is empty and we have no assigned thread, so try to steal.
    yield();                                                      //Before steal, yield processor.
    Process* victim = randomProcess();             //Select victim process at random.
    assignedThread = victim->deque.popTop();  //Try to steal thread.
}
Example of Work Stealing Using Ready Nodes:

1. Thread 1 assigns and executes root node
Example of Work Stealing Using Ready Nodes:

1. Thread 1 assigns and executes n4
2. Thread 2 assigns and executes n2
Example of Work Stealing Using Ready Nodes:

1. Thread 1 assigns and executes n5
2. Thread 2 assigns and executes n3
Example of Work Stealing Using Ready Nodes:

1. Thread 1 assigns and executes n6
2. Thread 2’s deque is empty, so it steals n8 from Thread 1’s deque
Example of Work Stealing Using Ready Nodes:

1. Thread 1 assigns and executes n7
2. Thread 2 assigns and executes n9
Why use nodes that represent a single instruction, instead of a stream of instructions?

- Multiprogrammed environment requires two levels of scheduling:
  1. The work stealing algorithm exists in user level, and is responsible simply for mapping nodes to a thread
  2. The kernel is responsible for mapping threads to processors.

- The kernel is treated as an adversary, as there is no guarantee to what type of schedule the kernel will choose for your program’s threads, nor the amount of processors the kernel will assign to your program.

- If instruction streams were used, we could end up with the original problem of not fully utilizing all processors.
Reasoning – Benign Adversary

- Studies were performed using different types of kernels (adversaries), which could change the way tasks were executed within the kernel schedule.

- Benign Adversary
  - A benign adversary is only able to choose the number of tasks, not which tasks it can execute
  - In this case, it is not necessary to call yield from the non-blocking work stealer
Reasoning – Oblivious Adversary

• Oblivious Adversary
  • An oblivious adversary is able to choose the number of tasks and which tasks it can execute, but it must do so before ever coming on-line.
  • In other words, its schedule is pre-determined before it begins to execute any tasks.
  • In this case, it is necessary for a thread that is attempting to steal work to call yieldTo(victim), which means that the current thread cannot be scheduled again until the victim has been scheduled.
• Adaptive Adversary

  • An adaptive adversary is able to choose the number of tasks and which tasks it can execute, and it can do so while it is online.

  • In other words, its schedule does not have to be pre-determined before it begins to execute any tasks.

  • In this case, it is necessary for a thread that is attempting to steal work to call `yieldToAll()`, which means that the current thread cannot be scheduled again until the kernel has scheduled every other thread first.
Conclusions

• The proposed thread scheduling algorithm proves to be quite effective in a multiprogrammed environment, with speedups that are close to PA-fold (PA is the number of processors allocated by the kernel for a given program).

• Very little overhead is created to implement the dags and deques, which does not burden executions in dedicated (non-multiprogrammed) parallel environments.
Future Work

- At the time of publication, this algorithm was implemented into a C++ threads library on UNIX.
- Future targets for integration include Java, Cilk, and a POSIX threads library.
- Cilk currently indicates it uses a work-stealing algorithm for its scheduler, but it is uncertain if it is using this exact algorithm.
QUESTIONS OR COMMENTS?