Lecture #13: Thursday, 16 may 2002
Lecturer: Mattan Erez
Scribe: Arjun Singh, Abishek Das
Reviewer: Mattan Erez

1 Two Fundamental Limits on Dataflow Multiprocessing

This paper observes two key problems in multiprocessing. First, the justification of extensive multithreading is based on an overly simplistic view of the storage hierarchy. Second, the local greedy scheduling policy embodied in dataflow is inadequate in many circumstances.

The synchronizing cost is a product of two things namely the number of global references and the amount of time spent synchronizing for a single reference. Assume that a physical processor hosts v virtual processors with at most one remote request each. Each virtual processor (VP) alternates between computation and communication. However, at the end of each local run the physical processor switches to another VP with a synchronization cost of S cycles which combines the cost of switching to another enabled VP and of enabling the requesting VP upon arrival of the response.

If v is sufficiently large, each request will complete before the other $v-1$ VPs have finished their turns at the processor.Thus, the processor is never idle and the communication latency is fully masked by the collective computation phases. We say the processor is saturated with available work. The utilization of the physical processor is given by

$$U_{sat} = \frac{R}{R+S} = \frac{1}{1+S\rho} \tag{1}$$

where $\rho = 1/R$, the number of remote references per instruction.

This demonstrates how synchronization cost determines the effectivenes in tolerating latency. If the synchronization cost is equal to the run length, the processor is half utilized in saturation.

The idle time of the processor is given by

$$Idle = L - (v-1)R - vS \tag{2}$$

where, L is the latency for remote operations, R is the number of useful cycles of work in between global references and S is the synchronization cost which includes the cost of

associating data, the scheduling overhead and the actual switching overhead. Note that they cound the synchronization time as useful work and not as idle time.

Scheduling must be done on a *thread* granularity and not at the instruction granularity. Also, it is far more efficient to schedule threads that share some state.

A *thread* is characterized by the following features:

- It is a group of instructions.
- It runs to completion.
- It shares registers and state.
- It ends on a remote reference.
- It is statically scheduled.

An *activation frame* is a collection of related threads. These activation frames can be employed to help scheduling threads with as much shared state as possible. Software can create an activation frame of data that is shared to help in scheduling. The idea is that when a new thread is required to be scheduled, it must be popped off the current activation frame and when all the threads in a frame are exhausted, the next activation frame must be used and so on.

1.1 Limitations of their scheduling

Consider the example illustrated in figure 1. Here two tasks are executed on each processor, one short and one long. The short task spawns off the two tasks executed by the next processor, while the long one just does some computation. There are plenty of independent tasks, but ineffective self-scheduling may fail to expose them. The short task should be scheduled first, so it can spawn work for the next processor. Execution of the long task is overlapped with the computation on succeeding processors, resulting in a tight piplelined execution (top right). However, if the long task is scheduled first only one processor will be busy at a time, resulting in very poor efficiency (bottom).

The solution is to allow fair, fine-grained dataflow scheduling so that neither task is delayed arbitrarily. Execution alternates between small fractions of the two tasks, so even if a long task is scheduled first, the short task is executed and the subsequent tasks are spawned before finishing the larger one.

However, fair scheduling can lead to the same problem. Figure 2 shows such a scenario. l independent tasks are executed on each processor. One task spawns the l tasks executed by the next processor at the end of its execution. This task should be executed first and to completion, resulting in a pipelined execution on all processors(top right). Fair scheduling would execute a small fraction of all tasks, then another fraction, and so on. Only when the last fraction of the first task is executed are the tasks spawned to the next processor,

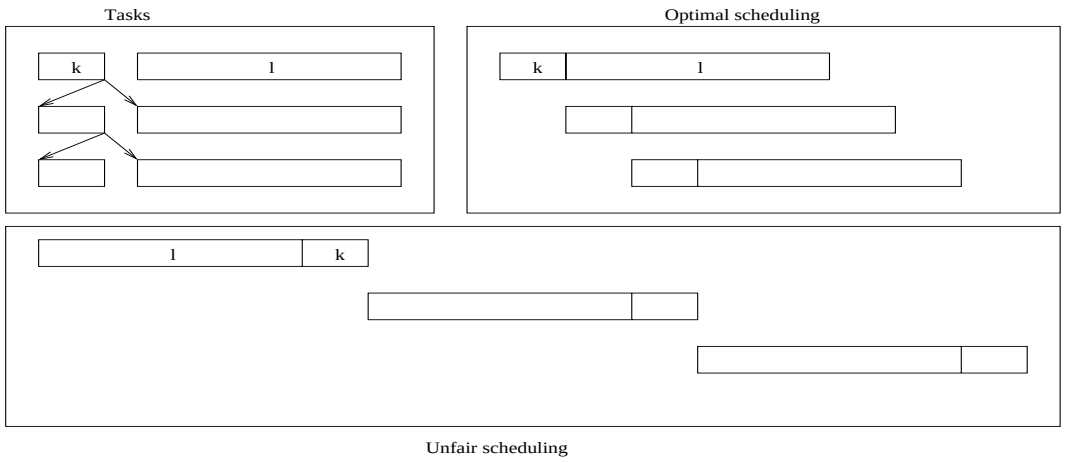

Figure 1: Scheduling limitation 1.

so only one processor is busy most of the time. Even simple unfair scheduling is likely to be beter than fair scheduling: if tasks are chosen at random, unfair scheduling is better than fair scheduling by a factor of two on average.

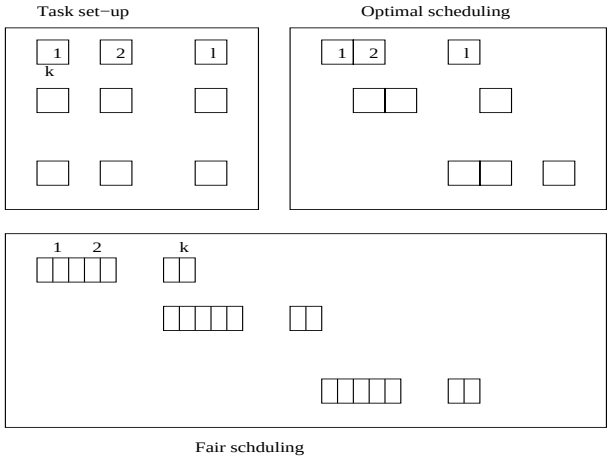

Figure 2: Scheduling limitation 2.

2 Exploting Fine-grain Thread Level Parallelism on the MIT Multi-ALU Processor

This paper discusses the use of fine-grain threads to fill the parallelism gap between the two extremes:

- Instruction-level: grain size of a single instruction
- Thread-level: coarse threads with grain size of a thousands of instructions

This parallelism is orthogonal and complimentary to coarse-thread parallelism and ILP. The authors also show that this approach enables exploiting parallelism in tasks with run lengths as small as 20 cycles. They also describe the communication and synchronization meachanisms implemented in the Multi-ALU processor(MAP) chip, which includes a thread creation instruction, register communication and a hardware barrier. With a three-processor implementation of the MAP, they claim to achieve fine-grain speedups of 1.2-2.1 on a suite of aplications.

The motivation for their work has been expressed in terms of the limitations in the prevalent methods of exploting parallelism:

- ILP in applications is restricted by control flow
- Hardware in superscalar designs is not scalable: both the instruction scheduling logic and the register file of a superscalar grow quadratically as the number of execution units is increased.
- There is limited coarse thread parallelism at small problem sizes in many applications.

Loop execution is a good example to illustrate the first and last limitations. Programs can be accelerated using coarse threads to extract parallelism from outer loops, fine threads to extract parallelism from inner loops and ILP to extract parallelism from subexpressions.

Fine-threads are appropriate for execution across multiple processors at a single node of a parallel computer where the interaction laencies are on the order of a few cycles. However, the cost to initiate a task, pass its argumnets, synchronize with its completion, and return results must be small compared to the work accomplished by the task. Hence, low overhead mechanisms for communication and synchronization is required.

The MAP chip provides three on-chip processors and methods for quickly communicating and synchronizing among them. The reason to choose three processors is not clear, but space restrictions seem to be most likely reason. Also, there is no binary advantage in their communication and synchronization mechanisms. Each processor is termed as a cluster, each of which is a 64-bit, three-issue, pipelined processor consisting of two integer ALUs, a floating-point ALU, associated register files, and a 4KB instruction cache. Each integer register file has 14 registers per thread, while the floating-point register file has 15 registers per thread. Each thread also has 16 condition code (CC) registers to hold boolean values. The execution units are 6-way multithreaded with the register files and pipeline registers of the top stages of the pipeline replicated. A synchronization pipeline stage holds instructions from each thread until they are ready to issue and decides on a cycle-by-cycle basis which thread will use the execution unit.

It has not been mentioned in the paper that threads have been virtualized as Horizontal and Vertical threads. While the former can share its context amongst all the

clusters, the later executes in only one cluster. Throughout the paper, they seemed to be running only one thread per cluster.

2.1 Communication

Unlike coarse grained multiprocessors, where comunication between threads is exposed to the application as memory references or messages, both of which require many cycles to transmit data from one chip to another, threads on seperate clusters in the MAP chip may communicate either through the shared memory, or through registers. Since the data need not leave the chip to be transferred from one thread to another, communication is fast, and thus well-suited to fine-grain threads. While memory communication takes place with loads and stores, register-register communication between clusters take place via the Cluster switch. The former takes a minimum of 10 cycles (both hits) and a maximum of 36 cycles (both miss), while the later is extremely fast, requiring only one more cycle to write to a remote register than to a local register. However, the register file sizes limit the storage for communicated values, and also require an additional synchronization between the consumer and producer to prevent values in the destination cluster from being overwritten.

The table 4 in the paper show the producer overhead and transfer latency during communication. The producer overhead includes initiating communication and address calculation (not required for registers). Both the producer overhead and transfer latency are also dependent on cache hit/miss for memory communication.

2.2 Synchronization

Synchronization is required to indicate when a task is to be started, when it is complete, or when two running threads must communicate. The MAP chip allows synchronization through memory, registers, and a hardware barrier instruction, each of which has different costs.

- Barrier: This is a global synchronization mechanism, implemented by the instruction *cbar*. A thread's execution is stalled until the threads on the other two clusters have reached a *cbar* instruction. Waiting threads do not spin or omsume any resources.

- Memory synchronization: Every memory location has a single synchronizaion bit that exists both in the off-chip DRAM and in the cache, enabling locking on a location-by-location basis. Load and store operations compare the synchronzation bit to a pre-condition and set it to a post-condition if successful. This allows synchronization on a word-by-word basis, but a consumer thread waiting for a producer will continue to make memory requests while spinning, which can slow down other threads trying to access the meory system. An alternatve to this is polling.

- Register: Fully/empty bits in a register scorecard is used to determine when values in registers are valid. When an operation issues, it marks the scorecard for its destination register invalid, and when the result is written, the destination register's scorecard is marked valid. To reduce the amount of interaction between physically distant clusters, an operation that writes to a remote cluster does not mark its destination register invalid. Instead, the onsumer must execute an explicit *empty* instruction to invalidate the destination register, prior to receiving any data.

2.3 Thread Creation

Three methods for starting a thread on a remote cluster have been examined in the paper. The cold start method uses a $hfork$ (horizontal fork) instruction, which starts the remote thread automatically with a single instruction. It writes a remote program counter and updates the thread control registers. Condition codes are used o denote success and failure. The other two methods are standby methods, in which there is a slave thread already running in the remote cluster and the master and slave communicate using either the memory or the registers. The *master call* overhead is in creating the thread, the *slave invoke* latency is from beginning the master call to the first slave instruction, the *slave return* latency is the time for the slave to signal the master and finally the *master return* latency is in master resynchronizing with the slave.

The $hfork$ instruction and the *standby register* methods are most efficient, with only one cycle of overhead for the master at call and return. *standby register* is a little faster overall as the slave incocation time is shorter. *standby memory* again suffers because of the memory spin loops to synchronize the master and slave.

2.4 Applications

The paper does a comparision study comparing the fine-grained applications, normalizing them to one cluser. They showed that their communication and synchronzation mechanisms, along with the standby thread creation mechanisms, helped to speed-up the applications. They have shown exploting inner and outer loop parallelism seperately. In inner-loop parallelism, they observe that load imbalance in the parallel versions is the main limitation, and not the communication overhead for FFT. Outer-loop parallelism results in shorter execution times than inner-loop, as more of the code is parallelized and the larger grain size requires less communication and synchronzation. FFT improves since the problem sizes in the parallel sections increase. Their design seemed to be over aggressive for the applications running on them.