

CSCE 5160 Parallel Processing

Review

Levels of parallelism

hardware level (data paths)

Instruction level

thread level

processor level

Algorithm level

Pipelining, Vector processing

SIMD and MIMD

Shared memory vs Message Passing

UMA and NUMA shared memory processors

Data parallelism and Task parallelism

CSCE 5160 Parallel Processing

Let us try to parallelize j loop using OpenMP

```
for (i = 0; i < n; i++) {  
    #pragma omp parallel for  
    for (j=0; j < n; j++) {  
        C[i,j] = 0;  
        for (k = 0; k < n; k++)  
            C[i,j] = C[i,j] + A[i,k]*B[k,j];  
        }  
    }
```

In OpenMP, the `pragma omp parallel for` states the the loop that follows should be executed in parallel

That is, for each value `j`, the loop will be assigned to a different processor

We will talk about OpenMP later and see to to have individual copies of variable as well as share data

CSCE 5160 Parallel Processing

Consider executing this using task parallel (pseudo code)

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++) { Create_task_to_Compute(i,j); }
```

In this example, **Create_task_to_Compute(i,j)** will spawn n tasks, each computing a different element of the result matrix.

We will see how we can create the tasks on different processors to compute(i,j) using MPI.

We need to learn how to send and receive data among the different processors

MIMD vs Cluster Computing

Cluster Computing and Grid Computing

These are basically Distributed memory (or message passing) MIMD systems.

In traditional message passing systems, it is assumed that all nodes in the system already have the program (or code) resident in their local memories. Only data is exchanged via messages.

In cluster computing, we cannot make such an assumption. We need to communicate both code and data using messages.

This leads to several interesting issues regarding the granularity of a task that should be executed at each node

Understand overhead of transmitting code and data

May need multi-level scheduling issues

A group of task will be assigned a cluster of nodes then we need to schedule individual tasks on individual nodes

Memory Systems

Memory latencies limit the performance of a parallel processing system
See Example 2.2 on page 17

1GHz CPU (1 ns cycle time) but DRAM latency is 100ns (no cache)

The best performance we can expect is 10M FLOPS

if we assume that each computation needs access to memory

Cache memories can help in reducing the latencies

If in cache we can retrieve data at the same speed as CPU

Cache miss rate is the limiting factor **So we need to reduce cache misses!**

Caches reduce latencies but require higher bandwidths

Latency: Time to get the “first” bit of data

Bandwidth: Number of bits per second

Memory Systems

Consider the example 2.4 on page 18

If we can only get one word per access then the **bandwidth is 1 word per 100 cycles**
or 1 word / 100ns or 10 M words per second

latency is 100ns

If we can get 4 word per access we have

4 words /100ns or 40M words per second

but latency is still 100ns

Consider the dot-product (multiplying two vectors)

$$\text{sum} = \text{sum} + a[i]*b[i]$$

Get 4 a[i]s and 4 b[i]'s in 200 cycles – and perform 8 operations (4 multiplies and 4 adds)
8 operations in 200ns or get 40 MFLOPS

This can sometimes be related to cache misses

you need to get a new 4-word block once every 4 accesses
or we can say there is a 25% miss rate

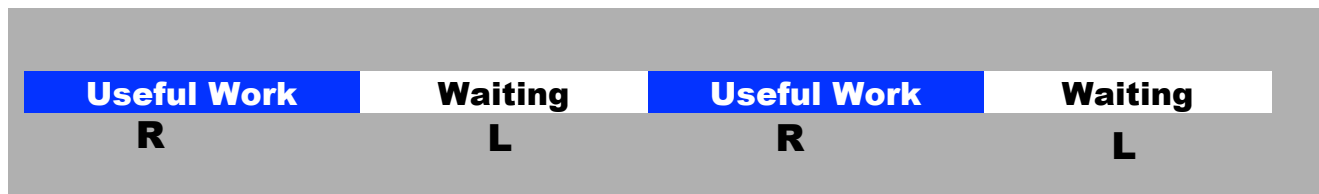
Reducing Latency

Other ways of reducing latency

Multithreading and pre-fetching data

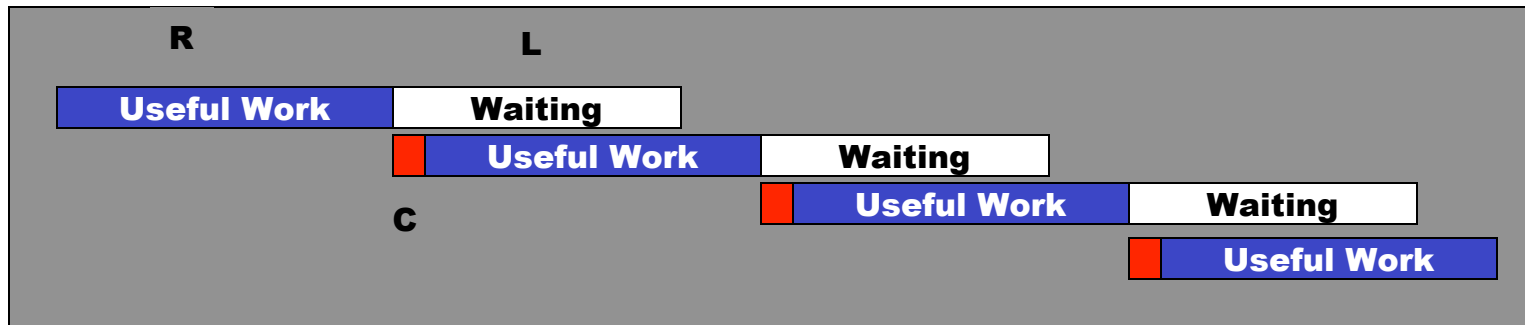
Let us start with a single thread

- Consider a single *CPU* and one thread
 - When a thread issues a “long latency” operation, the thread is blocked and *CPU* idles
- Let L be amount of time needed for the long latency operation, and let R be the average amount of time between such latency operations
 - Utilization without threads $U_1 = R/(R+L)$



Reducing Latency

- **Multithreading**
 - Consider a single *CPU* supporting multiple threads
 - When a thread issues a “long latency” operation, the thread is context switched and a new thread starts execution.
 - In addition to *R* and *L*, let us assume that *C* is the time needed to context switch



Reducing Latency

- **Multithreading**

- What is the maximum Utilization that is possible?
 - Maximum utilization with threads $U_{\max} = R/(R+C)$
 C is the context switching overhead
- How many threads are needed to reach U_{\max} ?
 - $N_{\text{saturation}} = (R+L)/(R+C)$
- If there are fewer threads?
 - Utilization with N threads $U_N = N*R/(R+L)$

Pre-fetching

Get data before it is actually needed.

Consider the example of fetching 4 words at a time from memory

This is prefetching 3 words before they are needed.

assuming those 3 words will be needed

Multithreading requires more hardware resources

more hardware “contexts”

need more cache memory – to accommodate the needs of all threads

need more memory bandwidth

Prefetching also needs more memory bandwidth

See example 2.9 on page 23

Pre-fetching

See example 2.9 on page 23

We are looking at using cache of 32KB for a single thread or
32 threads each thread using 1KB cache.

Miss rate if we use 32KB for a single thread is 10% (90% hit rate)
if we use 1KB per thread the miss rate is 75% (25% hit rate)

What are the bandwidth requirements if the computation makes a request every 1ns cycle?

Single thread– a miss every 1 in 10 requests. Since latency is 100ns we can assume that on average we need one word every 10 cycles. But since we get 4 words at a time
 $BW = 4 \text{ words}/10\text{ns} = 400\text{MB}$

Multithreaded. We have 3 misses or requests to memory in 4.
Or need 3 words every 4 cycles.
Again since we get 4 words every time
 $BW = 3\text{GBytes}$

Abstract Parallel Processing Model

For single processing system we normally use the RAM model

we need to get data from memory and process it

Complexity analyses are based on the abstract model

But if we have multiple processors sharing data we need to extend this model

Can all processors read and write at the same time?

Exclusive Read and Exclusive Write (EREW)

Very restrictive and sequential

Concurrent Read and Exclusive Write (CREW)

most practical in terms of implementation

Exclusive Read and Concurrent Write (ERCW)

doesn't make a lot of sense unless the writes are to different locations

Concurrent Read and Concurrent Write (CRCW)

most complex and potentially most parallel

PRAM model

Concurrent Write: We must assume one of the following

Common: only if all values to be written are the same

Arbitrary; only one value will actually be written

Priority: one value is chosen based on priority

Sum: the actual value to be written is the sum of all values

We can theoretically analyze any algorithm for its PRAM complexity using one of the various access models

For example Consider CRCW memory model for implementing matrix multiplication.

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++) {  
        C[i,j] = 0;  
        for (k = 0; k < n; k++) C[i,j] = C[i,j] + A[i,k]*B[k,j];  
    }
```

With n processors, we will assign different j loops to different processors

We can perform all memory accesses in parallel

PRAM model

What is the complexity?

The number of steps to complete the innermost loop computation
= 6 (3 read accesses, 1 write, 1 multiplication and 1 addition)

Complexity is $n*n*6 = O(n^2)$

If we have EREW PRAM model, none of the memory accesses can be completed in parallel -- that means, the read access for A and B in the innermost loop must be serialized.

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++) {  
        C[i,j] = 0;  
        for (k = 0; k < n; k++) C[i,j] = C[i,j] + A[i,k]*B[k,j];  
    }
```

So, even if we have n processors, we require n steps to read A[i,k] -- however B[k,j] can proceed in parallel since j is the processor number. The innermost computation requires n+5 steps (n steps to read A[i,k]).

Complexity is $n*n*(n+5) = O(n^3)$ -- no parallelism

Interconnection Networks

Processor - Memory and Processor-Processor networks

We may have networks connecting processors directly to memory units (shared memory)

Or

We may have networks connecting processors to processors (message passing)

Completely connected: every memory (or processor) is connected to every processing element (Figure 2.14, page 39)

Star Connected – Figure 2.14 page 39

Cross-bar networks. We need $p*b$ switching elements and each element will have $p+m$ wires. See page 35, Figure 2.8