ECSE 420-Parallel Computing

Assignment 1

Simon Foucher

260 223 197

**1-Describe briefly the following terms, expose their cause, and work-around the industry has undertaken to overcome their consequences:**

• **Memory wall**:

Refers to the latency encountered when a processer wants to communicate with memory, which limits the throughput of a memory access. It is primarily caused by the disparity between memory cycle time (MHz) and processor speed (GHz), but it is not an issues of bandwidth; it’s an issue of latency. Its negative effects are greatly felt when a CPU tries to perform little processing on a large quantity of data. Since the CPU is forced to wait for the next data, it can only perform the little work required on the current data it has then waists cycles until the new data arrives.

It is also relevant to note that CPU speeds have increased more significantly that memory speeds over the years, so the memory wall fundamental issues has increased (fortunately, some solutions have been developed)

**Work Around:**

1. Larger CPU caches which take advantage of locality. Besides increasing the cash size, making the cash more efficient can also help
2. The development of branch predictor algorithms to pre-fetch instructions
3. Harvard Architecture, which uses 2 separate busses; one for data and the other for instructions
4. Speeding up the memory clock and developing wider memory busses. Both of these approaches come with great complications; the memory clock causes stringent system timing requirement, and a wider bus implies more pins, more power consumption and more PCB layout complexity.
5. Berkley is currently working on Intelligent RAM (IRAM) which tries to integrate both a processor and memory onto the same die, or the same package.

**• Frequency wall**

Processor designs have reached the physical limits of achievable frequencies. Keeping increasing the frequencies above 3-4GHz have shown to bring diminishing returns on performance, because of signal skew and EMF interference between wires.

**Workarounds**:

The first approach developed to work around frequency limitations was to pipeline the data path. By parallelizing the instructions, we can effectively shorten the clock’s duty cycle and speedup the clock. Once pipelining reached its limits, people have been focusing their efforts on packing many processors on a single chip (dual/quad cores, etc…)

sources:

*http://en.wikipedia.org/wiki/Von\_Neumann\_architecture*

*http://c2.com/cgi/wiki?VonNeumannBottleneck*

*http://www.acm.org/crossroads/xrds5-3/pmgap.html*

*http://www.hpcaconf.org/hpca11/slides/Cell\_Public\_Hofstee.pdf*

**Explain the following terms and produce example for each of them:**

**1. SISD:** This represents a single instruction per clock cycle working on a single piece of data. These architectures only have a single processor, executing sequential single instructions on a distinct piece of data at a time.

Ex: A non pipelined MIPS architecture

**2. SIMD**: This implies a processor running a single instruction working on multiple pieces of data at the same time.

Example: In BCD addition, if a packed BCD only takes 1 byte and we are working with 32 bit registers, we can store 4 packed BCD per register, so this would allow a single ADD instruction to perform 4 additions at the same time, Intel’s MMX

**3. MISD**: Found when many processors, each having a control unit are performing different operations on the same data via a shared memory unit. At every clock cycle, one datum is extracted from memory and all processors do different things to the same data.

Example: a pipelined architecture

**4. MIMD:** This is a good way to achieve parallelism. MIMD architectures refer to many processors working asynchronously and independently on discrete data sets. At any given time of the execution, processors can be executing different instructions on unrelated data.

Example: Hypercube interconnection network and NUMA machines

*Sources:*

*http://www.tommesani.com/ProgrammingModels.html*

*http://en.wikipedia.org/wiki/MIMD*

**2- You should extend Amdahl’s and Gustafson-Barsis bound and make it slightly more realistic. Assuming the fixed overhead cost in the communication and the setup of parallel processes, derive the expressions for both bounds that take the overhead into account.**

S = sequential fraction of work

P = parallel fraction of work

n = number of processors

Amhdal’s original law

SU = 1/(S+P/n)

Therefore, S + P = 1; S = 1 - P; P = 1 - S,

We need to find new S and P accounting for the overhead of communication (OH)

S’ = (S + OH)/(total work + OH) = (S + OH)/(S + P + OH)

P’ = P / (total work + OH) = P / (S + P + OH)

New speed up, following the original Law:

SU = 1/(S’ + p’/n)

SU = (S + P + OH) / (S + OH + P/n)

Original Gustafson’s law

SU = n – S\*(n-1)

S’ = same as above = (S + OH)/(total work + OH) = (S + OH)/(S + P + OH)

So new taking into account OH:

SU = n –S’\*(n-1)

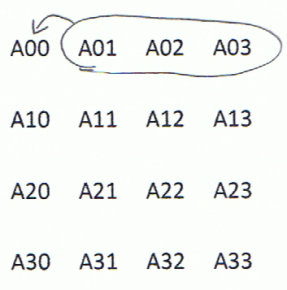
SU = n - (S + OH)\*(n-1)/(S + P + OH)

**3-Gaussian elimination**

**a) Draw a simple figure illustrating the dependences among matrix elements.**

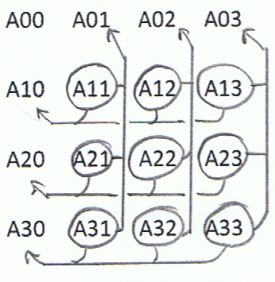
First inner for loop: for j = k+1 to n -1 do Akj = Akj/Akk

All row elements are dependent on the first non zero element

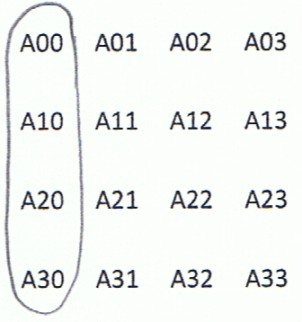


Second inner for loop: for i = k+1 to n – 1 do Aij = Aij – Aik\*Akj

Lower row elements are dependent on their corresponding column index element in the first row, as well as the first non zero element of their respective row.

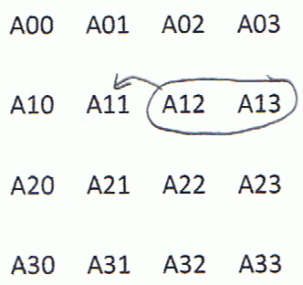


After which we can update the first non zero column with no data dependency (timing dependency; need to wait for first 2 inner loops to complete)



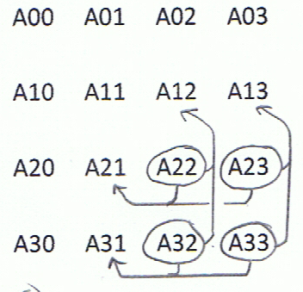
Then we get to the second iteration of the main loop. We find the same data dependencies, minus the first column which is already resolved. Similarly, we have the first inner loop on the second main sweep:

All row elements are dependent on the first non zero element

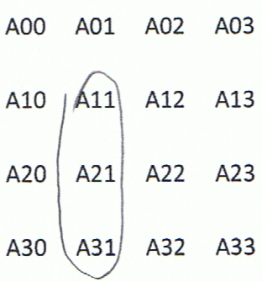


Then the second inner loop on the second main sweep:

Lower row elements are dependent on their corresponding column index element in the first row, as well as the first non zero element of their respective row.



Then the second column update can be done with no data dependency (timing dependency; need to wait for first 2 inner loops to complete)



And so on and so forth, until all the main sweeps are completed (all the columns are resolved)

**b) Assuming a decomposition into rows and an assignment into blocks of contiguous rows, write a shared address space parallel version using the primitives used for the equation solver.**

// Main loop

For(k = 0 to n - 1){

// Divide the first task amongst processes based on their pid. If there are more processors than // n-1, everything gets processed at the first iteration

If(pid = k % number of processors)

For( j = k + 1 to n - 1){

Akj = Akj/Akk;

}

Akk = 1;

// Wait for all the processors to complete their tasks

While((all Akk from k = 0 to k = n-1) != 1){}

// At this point (assuming n processor), all Matrix diagonal elements = 1

For( i = k+1 to n-1){

// Split the load amongst processors. Each processor will ‘complete’ a row task

if(pid = i % number of processors)

For( j = k+1 to n -1){

Aij = Aij - Aik\*Akj;

Aik = 0;

}

}

// Finally we want to pause until all the processors have completed their task

While((all Aik from k +1 to n-1) != 0){}

}

**4. Suppose we have a machine with the message start-up time of 20000 ns and the asymptotic peak bandwidth of 900 MB/s. The machine is sending the messages with n bytes. The start-up time includes all SW and HW overhead on the two processors, accessing the network interface and cross the network – it can be thought of as the time to send the zero-length message. At what message length is machine reaching the half of the peak bandwidth?**

Msg overhead: 20,000 ns

Peak Bandwidth: 900MB/s = 0.9B/ns

Size needed for half bandwidth (half-power point):

n1/2 = T0 \* B

n1/2 = 20,000 ns \* 0.9B/ns = 18,000B = 18KB

Check by computing transfer Time: 18,000B / (0.9B/ns) = 20,000ns of transfer time + 20,000ns of overhead = 40,000ns for the transfer, i.e. point of half-bandwidth

**5- We are going to find the average of elements in grid of (n\*n). Each element of this grid may be a mathematical expression. Therefore, every element of this grid requires computation. Based on the Amdhal’s law compute the speed up of using K processors in these two following situation:**

**a. Each processor has its own private value for holding the sum.**

Normal serial execution will take a single processor a first sweep to do local computations, then a second sweep to sum for the average, so a total time of 2 \* n2

If we can parallelize both the local calculations and the first round accumulations, here is the required time complexity:

- Sweep over n\*n and do computation (task shared between k processors): n2/k

- Accumulate private sum during sweep: n2/k

- Add all private sums into global sum: k

So the fastest parallel time is: n2/k + n2/k + k = 2n2/k + k

Giving a speedup of: 2n2/(2n2/k + k) = 2kn2/(2n2 + k2)

**b. The processors have to use one shared value to keep tracking of the sum. That is, every processor should sum its result to the shared variable of sum.**

Normal serial execution will still be of the order of: 2 \* n2

We can parallelize the first sweep suck that the k processor split the task of performing the local computations, reducing it to n2/k. Since the second phase requires the use of a single global variable, it has to be performed serially, so will still require n2 time.

So the speedup achieved is

2n2/(n2/k + n2) = 2/(1+1/k) = 2 at most with really large k.