Brian Ackley

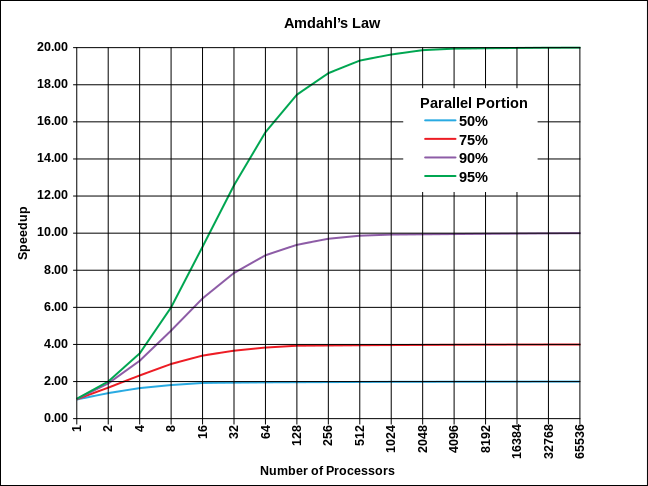
CSE 565

Spring 2014

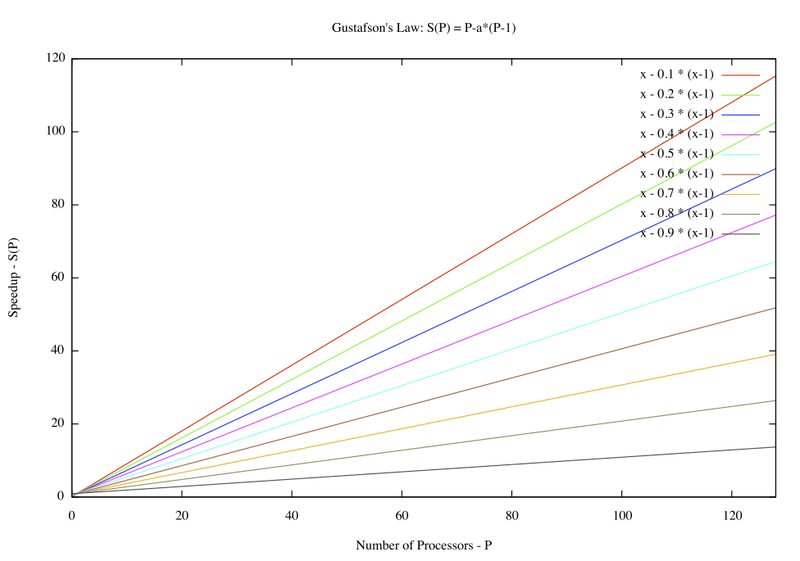
Midterm

* 1. The processor can take in two different types of information, instructions and data. These can then be separated into streams; Stream means a sequence of either instructions or data operated by the computer. Flynn’s taxonomy classifies machines according to whether they have one or more streams of each type. There are four combinations, SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data streams), MISD (multiple instruction streams, single data stream), and MIMD (multiple instruction streams, multiple data streams).
     1. The MISD is little used in practical machines, what’s used more often is MIMD and SIMD.
     2. The MIMD would be used for that of a multi-core machine such as an X86-64.
     3. The SIMD would be used for that of a GPU computer processor.

1.2. Amdahl's law is a model for the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized.



Gustafson’s law says that computations involving arbitrarily large data sets can be efficiently parallelized.



\*both graphs were taking from Wikipedia pages on the topic of amdahl’s law and Gustafson’s law\*

1.2.1. Amdahl’s law does have a mathematical proof. With one processor the algorithm will take n times units. With N processors the (1-f)n parallelizable operations will take (1-f)n/N time and the remaining f n non parallelizable operations will take f n time. We end up with f n + ((1-f)n)/N. the speedup of S will be n/(f n + ((1-f)n)/N) = 1/(f + (1-f)/N)

1.2.2. Neither of Amdahl’s law or Gustafson’s law was demonstrated from direct observation of actual executions from that of a program. They were both done through a mathematical solution through the problem. Once these were figured out they could then be implemented in some form of code and tested.

1.2.3. Gustafson's law relates to that of the machines we have today, with multi-processors, GPUs and supercomputing. This shows that with more performance we can do more task and jobs at a giving time then would could with less processing power.

2.1. Parallel formulation refers to a parallelization of a serial algorithm while parallel algorithm may represent an entirely different algorithm that the one used serially. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time.

2.2. The depth-first-search algorithm that Awerbuch discusses talks about a search and find algorithm that helps speed up searching through a tree. It does this by using parallelization and using task to speed up the process.

2.2.1. The task in the algorithm consist of DISCOVER, RETURN, VISITED, and ACK. The discover task is the first task that must be done before any other task. The reasoning for this is that the algorithm needs to have some type of data to start with. The next task could be visited; in the algorithm it tells any other task that the node is already aware of that data. Next could be either return or ack; return basically will return the value of which the node has already visited before. Ack is basically confirmation that the node got to the visited message ok. One could use data decomposition to show how this works. Going through a tree one node at a time can be very time consuming and by speeding this up by parallization increases the speedup the most. The set would be unique in this case since the tasks are the set itself so everything is part of this set that would be included in the task.

2.2.2. discover would go to visited, visted would go to ack, ack would go to return and return would go back to discover.

ACK Discover Return

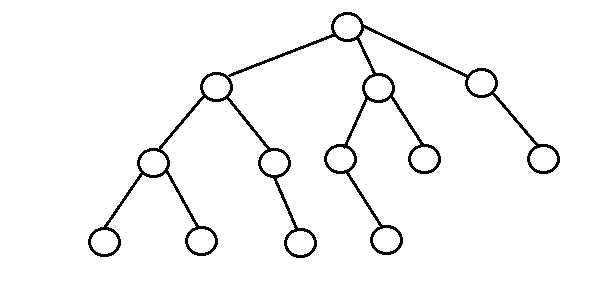
Visted

ACK

RETURN

ACK RETURN

DISCOVER



VISITED

The task dependency graph is a subgraph of the task interaction graph because there are interactions between different tasks even if the dependencies may not exist. This could occur for different reasons but usually happens because of shared data.

2.2.3. This can depend on how big the tree is and how many children nodes are on that branch of a parent node. The PE’s are the amount of children that exist on a single parent node. In a tree the big O is that of the size of the tree since one cannot search any larger than the size of the tree, the big O is that of O(log n) since we are now doing this in parallelization. If we were to do a search through the tree without parallelization then we would have to search each node individually and that would mean we would be searching through it at O(n) since each value must be searched.

2.3. one way to implement this on Cuda would be to parallelize all the children nodes on the tree. Discover would go to visited, which would go to a task to remove a node, which we could then have go to the ack. After the ack we could then go to the return task and from there we could go back up to discover. We would also need to create different functions within these tasks. We could setup everything but the discover and return task to be that of some type of device functions, and by doing that we could setup the discover task to be that of a host task for the entire thing.

3.1. multithreading is when a program can be used by more than one user at any time and even manage multiple request by those users without having to have multiple copies of the programming running on the computer. An example of this could be that of the Fibonacci numbers.

if (n < 2)

return n;

else

x = spawn Fib(n-1);

y = Fib(n-2);

sync;

return x+y;

mutex stands for mutual exclusion which ensures that two concurrent processes aren’t in a critical section at the same given time. Example of this:

mutex mx;

void print\_thread\_id (int id)

{

mx.lock();

cout << “thread id = “ << id << endl;

mx.unlock;

}

Int main()

{

thread th[5];

for (int I = 0; i < 5; ++i)

th [i] = thread(print\_thread\_id)

i++;

}

Memory leak happens when a program or the computer can’t manage its memory allocations correctly. This could come from an item that’s in memory but can’t be accessed by the code that is running. This can lead to many problems and leading to programs not running correctly and can even lead to crashing. A famous example of this is the Microsoft Windows Blue Screen. The most common issue that leads to memory leaks in object oriented languages is the lack of garbage control built into the code; in other words garbage in garbage out.

3.2. Threads are the smallest executable sequence of program instructions that an operating system scheduler manages. Threads are contained within a process and is a basic unit of the CPU utilization. On the Unix kernel, user threads are implemented on top of the kernel threads, which are implemented by allowing certain processes to share resources.

3.3. threads to a user process means that threads share parts of their state and allow multiple different threads read and write same memory. Each thread has its own registers and stack, but different threads can read and write the stack memory.

3.4. Posix threads allow concurrent process flow on multiprocessor systems to gain speed performance through parallel processing. Posix threads in Linux are now part of the interface standard. Boost threads allows the use of multiple threads to execute with shared data in C++. It provides classes and functions for managing all the threads and syncs data between the threads. This provides separate copies of the data specific to each thread individually. Zthreads is similar to that of boost threads and is available on nearly all platforms like that of boost. It includes several synchronization control objects and interruptible thread objects.

Posix thread implementation:

#include <pthread.h>

#include <stdio.h>

#define NUM\_THREADS 1

void \*PrintMessage (void \*threadid)

{

long third;

third = (long) threaded;

printf (“Computer Science 565!, thread #%ld!\n”, third;

pthread\_exit(NULL);

}

Int main (int argc, char \*argv[])

{

pthread\_t threads[NUM\_THREADS];

int rc;

long t;

for(t=0; t<NUM\_THREADS; t++){

printf("In main: creating thread %ld\n", t);

rc = pthread\_create(&threads[t], NULL, PrintMessage, (void \*)t);

if (rc){

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

pthread\_exit(NULL);

}

The main source of this code come from the computing.llnl.gov website under tutorials and written by Blaise Barney.

Boost Thread implementation:

#include <iostream>

#include <boost/thread.hpp>

#include <boost/date\_time.hpp>

boost::posix\_time::seconds workTime(3);

std::cout << "Worker: running" << std::endl;

// Pretend to do something useful...

boost::this\_thread::sleep(workTime);

std::cout << "Worker: finished" << std::endl;

}

int main(int argc, char\* argv[])

{

std::cout << "main: startup" << std::endl;

boost::thread workerThread(workerFunc);

std::cout << "main: waiting for thread" << std::endl;

workerThread.join();

std::cout << "main: done" << std::endl;

return 0;

}

This code comes from www.antonym.org website and is written by Gavin Baker

Zthread implementation:

#include "zthread/Thread.h"

#include <iostream>

using namespace ZThread;

class aRunnable : public Runnable {

void run() {

Thread::sleep(1000);

std::cout << "Hello from another thread" << std::endl;

}

};

int main() {

try {

// Implictly constructs a Task

Thread t(new aRunnable);

} catch(Synchronization\_Exception& e) {

std::cerr << e.what() << std::endl;

}

std::cout << "Hello from the main thread" << std::endl;

// Output:

// Hello from the main thread

// Hello from another thread

return 0;

}

This code comes from the zthread.sourceforge.net website and was written by Eric Crahen.