Project Report
CS 341: Computer Architecture Lab

Guna Prasaad 110050082  C Yeshwanth 110050083  Vipul Venkataraman 110050084  Mohit Gupta 110050085  Divyam Bansal 110050086
Introduction

Project Specifications

For this project, we learnt about how multi-core processors were different from other parallel computing frameworks. We learnt about the differences between multi core processors, GPUs, multi threaded and parallel computing architectures using multiple processors as well. We spent some time learning about the types of performance gains one can expect from multi core architectures owing to their exploitation of thread level parallelism. We also analyzed the problems that can occur with multicore architectures like the consistency of cache across different cores and how these can take a hit on the performance of the system. We also looked at different benchmark programs to see how parallelism could improve the run time performance of the program. For example, the problem of computing the sum of an array of numbers can be sped up by a factor of n where n is the number of cores in an asymptotic situation.

We also looked into other programs that can benefit from parallelism like Matrix Multiplication and Monte Carlo Simulation. Parallelism is possible here because the computations going on in computing some sub product of the matrix are independent of the other and the Monte Carlo Simulation involves a large number of parallel samples from an identical distribution which makes it easily 'parallelizable' in a multi core system.

We also looked at theoretical bounds for the performance of multi core systems like Amdahl’s law and Gustafson’s Law both of which analyze the performance bounds from different perspectives. As far as Amdahl’s Law is concerned, it fixes the size of the problem to be solved while Gustafson’s Law holds the time taken by the parallel program as fixed. We also looked at how to estimate the 'parallelizability' of a given program using analysis techniques after the program has been executed using the Karp Flatt metric.

We also looked into practical aspects of how to run our experiments using C++ code. The use of threads in C++ code allow the parallelization of our code into multiple modules which can be run concurrently on multiple cores. The threads in C++ do not inherently have any support for exploiting multi threaded or multi core architectures but rely on the OS to perform the necessary parallelization. We learnt how to restrict the number of cores used by the linux kernel to conduct our experiments. After this, we plan to run our experiments on multiple programs with different levels of parallelization that they permit through the use of shared variables, etc, to judge the impact that they have on performance. We will document the performance gains that we obtained through multicore processors using the usual metrics like speed up and comparing different programs based on the Karp Flatt metric for ‘parallelizability’ of programs. We will also go through some other benchmarks for testing the performance of multi core processors.

References

Parallel Sum

Introduction

In this section, we have consider the problem of finding the sum of elements of an array. This problem of finding sum of array elements occurs quite frequently in practice and we wanted to analyze how we can improve the performance in terms of time by using multiple cores.

Algorithm

Here we give a naive explanation to our algorithm

- The user has to input the number of threads to be used, call it \( n \)
- The size of the array was set to be \( 10^9 \)
- We divide the array into \( n \) parts, depending on the number of threads
- We compute the sum of each part parallely
- We then wait for all the threads to complete
- We then add the individual sums obtained through the parallelism introduced

Plot

We include the performance plot obtained through our experiments

![Figure 1: Number of threads vs Time in nanoseconds](image)
Inferences and Results

We enumerate the observations made from the plots as follows

- We observe that the running time decreases as expected initially
- But it doesn’t continue to do so
- After certain number of threads, the running time increases
- Since our processor is 4 cores, we won’t have increased parallelism when the number of threads goes beyond 4
- We feel the thread creation overhead will outweigh the performance improvement due to increased number of threads after a certain number
- This is because the waiting time to complete all the threads will then increase

References

Merge Sort

Introduction

In this section, we have considered the problem of sorting elements of an array in increasing order. This problem is very common in practice. So we have used parallel programming to speed up the process.

Algorithm

Here we give a naive explanation to our algorithm

- The user specifies the number of elements to be sorted say $n$
- We then generate a random permutation of numbers between 0 and $n-1$
- The user has to input the number of threads to be used, call it $t$
- We divide the array into $t$ parts, depending on the number of threads
- All the subarrays are sorted by individual threads
- We then wait for all the threads to complete
- We do a $t$-way merge of the arrays thus obtained
- For the merge step, we have used a minheap to get every element in $O(\log t)$ time

Plot

We include the performance plot obtained through our experiments

Figure 2: Number of threads vs Time in nanoseconds
Inferences and Results

We enumerate the observations made from the plots as follows

- We observe that the running time decreases as expected initially
- But it doesn’t continue to do so
- After certain number of threads, the running time increases
- Since our processor is 4 cores, we won’t have increased parallelism when the number of threads goes beyond 4
- We feel the thread creation overhead will outweigh the performance improvement due to increased number of threads after a certain number
- This is because the waiting time to complete all the threads will then increase

References

Matrix Multiplication

Introduction
In this section, we have consider the problem of multiplying two matrices. This problem occurs quite frequently in practice and we wanted to analyze how we can improve the performance in terms of time by using multiple cores.

Algorithm
Here we give a naive explanation to our algorithm

- The user has to input the number of threads to be used, call it n
- The size of the array is then obtained
- We divide the resultant into n parts, depending on the number of threads
- The division is made row wise
- We compute the result of each part parallely, though using the general algorithm for matrix multiplication for each value
- We then wait for all the threads to complete
- We store all the values obtained to the resultant matrix

Plot
We include the performance plot obtained through our experiments

![Graph](image)

Figure 3: Number of threads vs Time in nanoseconds
Inferences and Results

We enumerate the observations made from the plots as follows

- We observe that the running time decreases as expected initially
- But it doesn’t continue to do so
- After certain number of threads, the running time increases
- Since our processor is 4 cores, we won’t have increased parallelism when the number of threads goes beyond 4
- We feel the thread creation overhead will outweigh the performance improvement due to increased number of threads after a certain number
- This is because the waiting time to complete all the threads will then increase

References

2. [http://www.cas.mcmaster.ca/~qiao/courses/cs748/lecture05.html](http://www.cas.mcmaster.ca/~qiao/courses/cs748/lecture05.html)
Monte Carlo Simulation

Introduction
In this section, we have consider the problem of determining the value of \( \pi \) experimentally. We use the technique of Monte Carlo Simulation to do this. We perform repeated random samplings to obtain numerical results.

Algorithm
Here we give a naive explanation to our algorithm

- The user has to input the number of threads to be used, call it \( n \)
- The number of points to be samples is obtained
- We now generate points and find if they lie inside the circle parallel-ly
- We find the result for each thread independently
- We then wait for all the threads to complete
- We now determine the value of \( \pi \) from the ratio of the number of points that were found lying inside the circle

Plots
We include the performance plots obtained through our experiments

![Plot](image)

Figure 4: Number of threads vs Time in nanoseconds when \( n = 100 \)
Inferences and Results

We enumerate the observations made from the plots as follows:

- We observe that the running time decreases as expected initially.
- But it doesn’t continue to do so.
- After certain number of threads, the running time increases.
- Since our processor is 4 cores, we won’t have increased parallelism when the number of threads goes beyond 4.
- We feel the thread creation overhead will outweigh the performance improvement due to increased number of threads after a certain number.
- This is because the waiting time to complete all the threads will then increase.

References

Bellman Ford Shortest Path Algorithm

Introduction

The Bellman Ford Algorithm is an algorithm that computes the shortest paths from a given source node to every other node in a graph. It improves upon Dijkstra’s Algorithm by allowing negative edge weights in a graph as long as there are no negative edge weights in the graph. We consider the problem of trying to parallelize the Bellman Ford algorithm.

The Algorithm

The naive implementation of the Bellman Ford algorithm with pseudo-code is shown below.

```plaintext
procedure BellmanFord( list vertices, list edges, vertex source )
  // This implementation takes in a graph, represented as lists of vertices and edges, and fills two arrays (distance and predecessor) with shortest-path information
  // Step 1: initialize graph
  for each vertex v in vertices:
    if v is source then distance[v] := 0
    else distance[v] := infinity
    predecessor[v] := null

  // Step 2: relax edges repeatedly
  for i from 1 to size( vertices )−1:
    for each edge (u, v) with weight w in edges:
      if distance[u] + w < distance[v]:
        distance[v] := distance[u] + w
        predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
      error "Graph contains a negative-weight cycle"
```

This version of the Bellman Ford Algorithm is very slow and runs in $O(|v||e|)$ time. This is also difficult to parallelize as the weights at each node need to be synchronized after every iteration of the outer loop. The cost of synchronization can be very hurtful to the performance of the algorithm especially if the number of nodes in the graph is large. We therefore needed to implement an asynchronous version of the algorithm to reduce the cost of synchronization. We did this by observing that not all edges should be updated at once synchronously. An asynchronous update of edges is sufficient for the Algorithm. The second observation is that not all nodes need to be updated. We only need to update nodes whose neighbours have changed costs before but the changes have not propagated to the surrounding nodes. Taking a leaf out of Networking Protocols, we implemented a version of the Bellman Ford Algorithms which provides to each thread a set of start vertex weights to update and each of them proceeds to run a version of the asynchronous Bellman Ford algorithm on a shared graph. This reduces cost as each edge is not analysed in every iteration but rather only when a node at one the ends changes it’s weight. The asynchronous nature of the algorithm also reduces the synchronization overhead between threads.
Experimental Results

Figure 6: Number of threads vs Time in microseconds

Inferences and Results
As expected the running time decreases with the number of Threads being used for the computation. It decreases until the point of using 4 cores but increases hereafter because of the limitations of the machine we were testing on (We had 4 cores). After that point, another factor that comes into play is the cost of a singular update to a node itself. As we are using shared memory amongst threads, the likelihood of a conflict in the access to the nodes in the graph increases. This leads to a reduction in performance but is still better than Bellman Ford running on a single thread. This is a case where shred resources limit the performance of the algorithm but are required for the correctness of the algorithm.

References