VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Parallel and Distributed Computing
CSE4001

Theory Assignment 1

Slot : E2

Name : Kulvir Singh

Register Number : 19BCE2074

Question 1 a

(a) Develop an execution schedule for the given code fragment for adding a list of four numbers using a two-way superscalar processor. [5]

Load R1, @2000

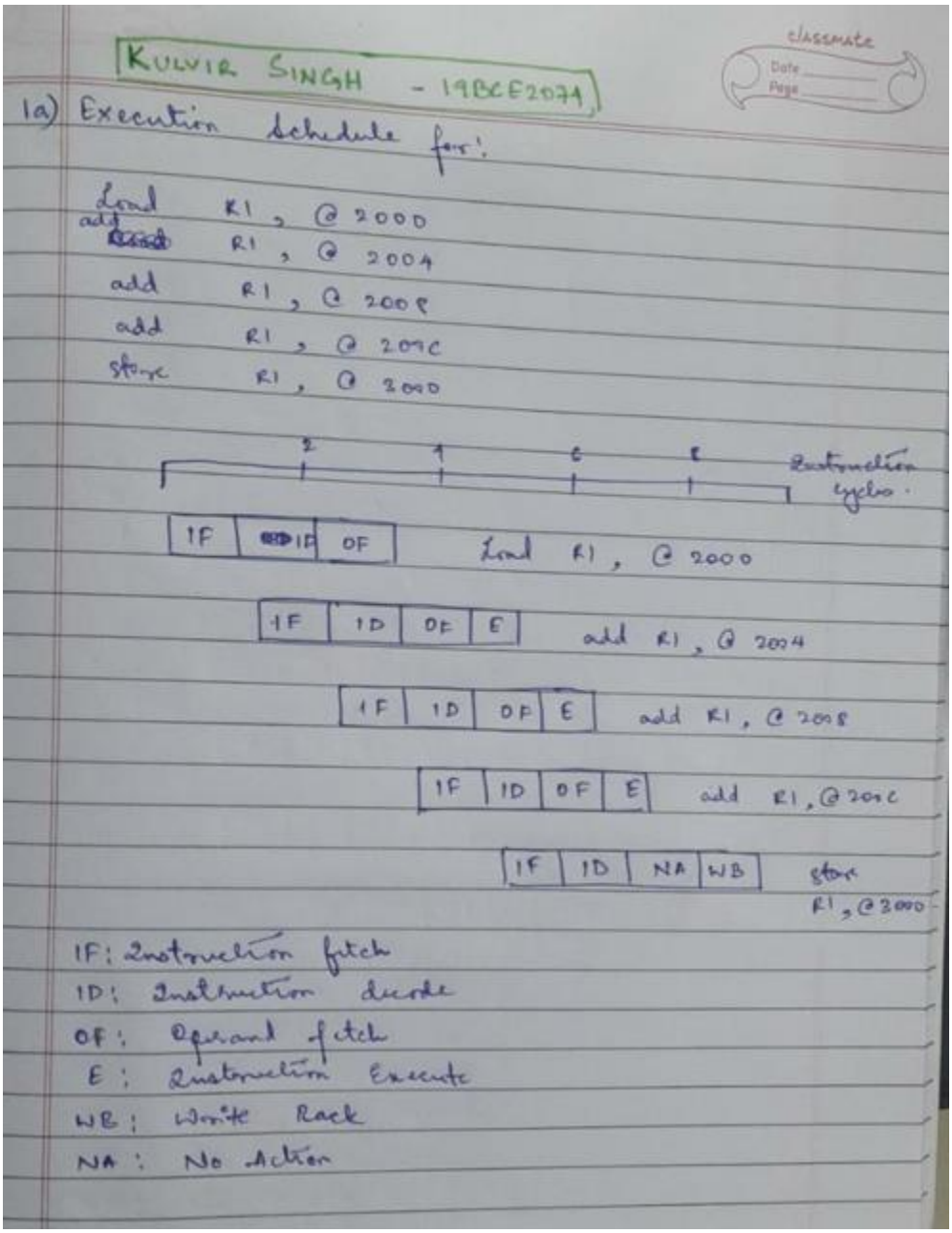
add R1, @2004

add R1, @2008

add R1, @200C

store R1, @3000

Solution :



Question 1 b

Write a C-program using Open MP to find the value of PI?

Solution :

For estimating the value of PI using parallel concepts we will make use of the MONTE CARLO ALGORTIHM.

In this algorithm, :

INPUTS :

- N – the range of iterations
- K – number of threads to be created

FLOW :

- Initialize 3 variables x,y and d to store the X and Y coordinates of a random point
- We have 2 variables with values 0 to store the points lying inside circle of radius 0.5 and square of side length 1
- Now starts the parallel processing with OpenMp together with reduction()
- Finally we use this formula to get the approx. value
  Pi = 4.0 * ((double)pCircle / (double)(pSquare))

OUTPUT : Value of PI approximated

Code :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void monteCarlo(int N, int K)
{

        double x, y;
        double d;

        int pCircle = 0;
        int pSquare = 0;

        int i = 0;

#pragma omp parallel firstprivate(x, y, d, i) reduction(+ : pCircle, pSquare) num_threads(K)
```

```
	{
		srand48((int)time(NULL));

		for (i = 0; i < N; i++) {
			x = (double)drand48();
			y = (double)drand48();
			d = ((x * x) + (y * y));
			if (d <= 1) {
				pCircle++;
			}

			pSquare++;
		}
	}
	double pi = 4.0 * ((double)pCircle / (double)(pSquare));
	printf("Final Estimation of Pi = %f\n", pi);
}

int main()
{
	int N = 100000;
	int K = 6;
	monteCarlo(N, K);
}
```

Output Screenshot :

Question 2 a

Does Multithreading and prefetching solve all the problems related to memory system performance? Justify your answer with examples. [5]

Solution

No, multithreading and prefetching do not solve ALL the problems related to memory system performance. While the use of multithreading and prefetching do solve a majority of time complexity related issues in computing, they have a few drawbacks. We will first discuss the advantages of the use of multithreading and prefetching with respect to memory system performance.
Resource sharing, All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.
Responsiveness, Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time. Therefore we can say that it proves to be economical for the memory system.
Limitations of Memory System Performance • Memory system, and not processor speed, is often the bottleneck for many applications. • Memory system performance is largely captured by two parameters, latency and bandwidth. • Latency is the time from the issue of a memory request to the time the data is available at the processor. • Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Example :

Multithreading and prefetching are critically impacted by the memory bandwidth. Consider the following example:

Consider a computation running on a machine with a 1 GHz clock, 4-word cache line, single cycle access to the cache, and 100 ns latency to DRAM. The computation has a cache hit ratio at 1 KB of 25% and at 32 KB of 90%. Consider two cases: first, a single threaded execution in which the entire cache is available to the serial context, and second, a multithreaded execution with 32 threads where each thread has a cache residency of 1 KB.

• If the computation makes one data request in every cycle of 1 ns, you may notice that the first scenario requires 400MB/s of memory bandwidth and the second, 3GB/s. Bandwidth requirements of a multithreaded system may increase very significantly because of the smaller cache residency of each thread.

• Multithreaded systems become bandwidth bound instead of latency bound.

• Multithreading and prefetching only address the latency problem and may often exacerbate the bandwidth problem.

• Multithreading and prefetching also require significantly more hardware resources in the form of storage.

Question 2 b

Write a code using OpenMP to make the loop iterations independent to safely execute in any order without loop carried dependency. [5]

Solution :

The following code does not contain any loop dependency and the loop iterations are independent from each other. This has been achieved using OPENMP.

Code :

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main(int argc, char *argv[])
{
  int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];

  for (i = 0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
#pragma omp parallel shared(a, b, c, nthreads, chunk) private(i, tid)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n", tid);

#pragma omp for schedule(dynamic, chunk)
    for (i = 0; i < N; i++)
    {
      c[i] = a[i] + b[i];
      printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
    }
  } /* end of parallel section */
}
```

Output Screenshot :

```
kulvir06@ubuntu:~/Desktop/PDC$ gcc loop_dep.c -o loop_dep -fopenmp
kulvir06@ubuntu:~/Desktop/PDC$ ./loop_dep
Thread 1 starting...
Thread 1: c[0]= 0.000000
Thread 1: c[1]= 2.000000
Thread 1: c[2]= 4.000000
Thread 1: c[3]= 6.000000
Thread 1: c[4]= 8.000000
Thread 1: c[5]= 10.000000
Thread 1: c[6]= 12.000000
Thread 1: c[7]= 14.000000
Thread 1: c[8]= 16.000000
Thread 1: c[9]= 18.000000
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[20]= 40.000000
Thread 1: c[21]= 42.000000
Thread 1: c[22]= 44.000000
Thread 1: c[23]= 46.000000
Thread 1: c[24]= 48.000000
Thread 1: c[25]= 50.000000
Thread 1: c[26]= 52.000000
Thread 1: c[27]= 54.000000
Thread 1: c[28]= 56.000000
Thread 1: c[29]= 58.000000
Thread 1: c[30]= 60.000000
Thread 1: c[31]= 62.000000
Thread 1: c[32]= 64.000000
Thread 1: c[33]= 66.000000
Thread 1: c[34]= 68.000000
Thread 1: c[35]= 70.000000
Thread 1: c[36]= 72.000000
Thread 1: c[37]= 74.000000
Thread 1: c[38]= 76.000000
Thread 1: c[39]= 78.000000
Thread 1: c[40]= 80.000000
Thread 1: c[41]= 82.000000
Thread 1: c[42]= 84.000000
Thread 1: c[43]= 86.000000
Thread 1: c[44]= 88.000000
```