PD_10_Kushagra Suryawanshi

HPC LAB ASSIGNMENT 2

Title:
Write a C program and convert it into parallel using Open MP directive.

Aim:
Design and implement a serial program written in C and convert it into its parallel equivalent. Select an appropriate program for serial to parallel conversion. (Divide and Conquer strategies are most suited and preferred or Matrix Multiplication (possibility of parallelism)).

Objective:
1. To analyse the existing algorithm
2. To use Open MP directive in accordance with the need of parallel construct in the code.

Theory:

Q1. Write about the algorithm selected for serial to parallel conversion.

A1. There are a number of bottlenecks typically encountered in the transition from serial processing to parallel processing. One of the most pernicious is that of **mindset**: people who have been in the computing business for a long time are understandably reluctant to have to learn a new way of designing their codes, and an efficient parallel algorithm often has little similarity to an efficient serial algorithm. The very first task in the conversion effort is to step way back from the existing serial application, and re-examine the **intent** it was written to serve: *can this task be effectively and efficiently performed in parallel, and, if so, how best can that be accomplished?*

Very often existing serial code has to be almost completely ignored, and the parallel version written virtually from scratch. This can be a major commitment of resources, and for some dusty-deck codes the projected return from such an investment is often considered to be insufficient to warrant the effort.

However, once the decision has been made to move from serial to parallel, the real nitty-gritty work of code conversion can very often be helped along by application of the growing number of automatic tools, well-seasoned by the manual use of hard-learned rules of thumb.

Q2. Write about the Open MP constructs used.

A2. The core elements of Open MP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

The syntax of the parallel construct is as follows:

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
  structured-block
```

where clause is one of the following:

```
if([parallel :] scalar-expression)
num_threads(integer-expression)
default(shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction([reduction-modifier ,] reduction-identifier : list)
proc_bind(master | close | spread)
allocate([allocator :] list)
```

Compute Total cost and Efficiency as: -

Total Cost = Time complexity × Number of processors used
Efficiency = Execution time of sequential algorithm/Execution time of the parallel algorithm

Consider data points as follows: -

Sr. No.	Data points/Data values	Time Taken for serial program	Time Taken for parallel program
1	500*500	**0.719s**	**1.04s**
2	1000*1000	**4.6s**	**8.4s**
3	2000*2000	**41s**	**25s**

Input: Unsorted array of data points/values.
Output: Sorted Array of data points/values.
Platform: Ubuntu (give latest version) or Windows
Conclusion: Thus, successfully studied, analysed serial to parallel conversion.

FAQs:

Q1. What is an Open MP directive? Give syntax of parallel for directive.

A1. Open MP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. Open MP identifies parallel regions as blocks of code that may run in parallel.

Example:

#pragma omp parallel

#pragma omp for, etc.

Syntax of parallel for

#pragma omp parallel for

for (int i=1; i<100;i++)

{…

…}

Q2. Give an example from your daily activities where you can incorporate parallelism.
A2. A parallel program to play chess might look at all the possible first moves it could make. Each different first move could be explored by a different processor, to see how the game would continue from that point. At the end, these results have to be combined to figure out which is the best first move. In parallel processing, we take in multiple different forms of information at the same time. This is especially important in vision. For example, when **you see a bus coming towards you, you see its colour, shape, depth, and motion all at once**. If you had to assess those things one at a time, it would take far too long.

CODE:

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

void fill(long long int **mat, int N) {
    *mat = (long long int *)malloc(N * sizeof(long long int));
    for (int i = 0; i < N; i++) {
        (*mat)[i] = rand();
    }
}

void print(long long int *mat, int a, int b) {
    for (int i = 0; i < a; ++i) {
        for (int j = 0; j < b; ++j) {
            printf("%lld ", mat[i * b + j]);
```

```
        }
        printf("\n");
    }
    printf("\n\n");
}

void matMul(const long long int *A, const long long int *B, long long int *C,
int L, int M, int N) {

    clock_t start, end;
    start = clock();
#pragma omp parallel for default(shared) schedule(dynamic)
    for (int i = 0; i < L; ++i) {
        for (int k = 0; k < N; ++k) {
            for (int j = 0; j < M; ++j) {
                C[i * M + j] += A[i * N + k] * B[k * M + j];
            }
        }
    }
    end = clock();
    printf("\nTime for threaded: %lfs\n", (double) (end - start) /
CLOCKS_PER_SEC);
}

void matMulopt(const long long int *A, const long long int *B, long long int
*C, int L, int M, int N) {

    clock_t start, end;
    start = clock();

    for (int i = 0; i < L; ++i) {
        for (int k = 0; k < N; ++k) {
            for (int j = 0; j < M; ++j) {
                C[i * M + j] += A[i * N + k] * B[k * M + j];
            }
        }
    }
    end = clock();
    printf("\nTime normal: %lfs\n", (double) (end - start) / CLOCKS_PER_SEC);
}


int main() {

    int L = 5000, M = 5000, N = 2000;
    printf("Matrix of size = %d x %d", M,N);
    long long int *A, *B, *C, *E;

    fill(&A, L * N);
```

```
    fill(&B, N * M);

    C = (long long int *) calloc(L * M, sizeof(long long int));
    E = (long long int *) calloc(L * M, sizeof(long long int));


    matMul(A, B, C, L, M, N);
    matMulopt(A, B, E, L, M, N);

    for (int i = 0; i < L * M; ++i) {
        if (C[i] != E[i]) {
            printf("Problem with %d\n", i);
            break;
        }
    }

    printf("number of threads: %d\n", omp_get_num_threads());

    free(A), free(B), free(C), free(E);
    return 0;
}
```

OUTPUT:

```
kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ gcc -fopenmp -o hpc_matrix hpc_matrix_2.c

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./hpc_matrix.exe
Matrix of size = 500 x 500
Time for threaded: 1.046000s

Time normal: 0.719000s
number of threads: 1

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ gcc -fopenmp -o hpc_matrix hpc_matrix_2.c

kush123@LAPTOP-BHV5R0EU /cygdrive/f/Trimester 8/HPC
$ ./hpc_matrix.exe
Matrix of size = 1000 x 1000
Time for threaded: 8.468000s

Time normal: 4.657000s
number of threads: 1
```