

Threads

1. Introduction

Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system. Multithreading libraries tend to provide a function call to create a new thread, which takes a function as a parameter. A concurrent thread is then created which starts running the passed function and ends when the function returns. The thread libraries also offer data synchronization functions.

2. Problem

2.1. Matrix Multiplication

Time complexity of matrix multiplication is $O(n^3)$ using normal matrix multiplication. Multi-threading can be done to improve it. In multi-threading, instead of utilizing a single core of your processor, we utilize all or more core to solve the problem.

2.1.1. Matrix Multiplication by row

We compute each row of the resulting matrix on a thread. Initialize an array of threads with size equal to the number of rows of the first matrix then create threads that compute one row at a time.

2.1.2. Matrix Multiplication by element

We compute each element of the resulting matrix on a thread. Initialize an array of threads with size equal to the number of elements of the resulting matrix then create threads that compute an element at a time.

2.2. Merge Sort

Merge Sort is a popular sorting technique which divides an array or list into two halves and then starts merging them when sufficient depth is reached. Time complexity of merge sort is $O(n \log n)$. Multi-threading can be done to improve it. In multi-threading, instead of utilizing a single core of your processor, we utilize all or more core to solve the problem.

3. Solution

3.1. Matrix Multiplication

3.1.1. Implementation

The following code is for the matrix multiplication by row function. The said function computes the matrix multiplication row by row according to the step (core) value.

```
void* multi_by_row(void* arg)
{
    int step = step_i++;
    for (int i = step; i < (step + 1); i++)
    {
        for (int j = 0; j < k; j++)
        {
            for (int l = 0; l < m; l++)
            {
                *(matC+(i*k+j)) += (*(matA+(i*m+l))) * (*(matB+(l*k+j)));
            }
        }
    }
}
```

Then, the second approach is performed – thread for each element –, as seen in the code snippet, the element position is calculated from the given step, i.e. the 4th element in a 2x2 matrix would be in row $3/2 = 1$ and col $3\%2 = 1$ then the desired element lies in row 1 col 1, after that its value is computed from matA and matB corresponding rows and cols.

```
void* multi_by_element(void* arg)
{
    int step = step_i++;
    int row = step/k; // rows
    int col = step%k; // cols
    for (int i = 0; i < m; i++)
    {
        *(matC+(step)) += (*(matA+(row*m+i))) * (*(matB+(i*k+col)));
    }
}
```

One array is made in the driver code with size equal to the number of elements of the resulting matrix, then firstly the multiply by row threads are assigned to the few first locations in the array and executed and time elapsed is calculated, then the second process overwrites the array threads filling it with its threads, executing and calculating the time elapsed, then all of that is written to the output file. Driver code:

```
int main(){
    // Read file values in matA and matB
    read_input("input.txt");
    // Initialize output matrix
    matC = init_mat(n, k);

    // declaring threads
    int MAX_THREADS = n*k;
    pthread_t threads[MAX_THREADS];

    // creating threads
    clock_t begin = clock();
    for (int i = 0; i < n; i++){
        int* p;
        if(pthread_create(&threads[i], NULL, multi_by_row, (void*)(p)) == 0){}
        else{
            puts("ERROR IN THREADS");
        }
    }
    // joining and waiting for all threads to complete
    for (int i = 0; i < n; i++)
        pthread_join(threads[i], NULL);
    clock_t end = clock();
    t_1 = calculate_time(begin, end);

    // reset
    step_i = 0;
    matC = init_mat(n, k);

    // creating threads
    begin = clock();
    for (int i = 0; i < MAX_THREADS; i++){
        int* p;
        if(pthread_create(&threads[i], NULL, multi_by_element, (void*)(p)) == 0){}
        else{
            puts("ERROR IN THREADS");
        }
    }
    // joining and waiting for all threads to complete
    for (int i = 0; i < MAX_THREADS; i++)
        pthread_join(threads[i], NULL);
    end = clock();
    t_2 = calculate_time(begin ,end);

    output_to_file();
    return 0;
}
```

3.1.2. Output

```

/home/omarxadel/Desktop/matrix_multiplication/output_matrix.txt - Mousepad
File Edit Search View Document Help
Multiplication of A and B
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
END1 [elapsed time in procedure 1 is 0.000151]
Multiplication of A and B
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
END2 [elapsed time in procedure 2 is 0.000677]

```

As you can notice, the time taken by the thread per row procedure is less than that of the thread per element procedure, and that's because in the rows procedure, the CPU works with manageable number of threads relative to the CPU cores, in the element procedure, the CPU handles and manages multiple threads at the same time, making it costly to switch between them rather than doing the actual process.

3.2. Merge Sort

3.2.1. Implementation

Merge sort is implemented with two main functions, the recursive `merge_sort()` and `merge()` which completes the merge of the divided arrays when the sorting is done.

```

void merge_sort(int *arg){
    int *n1 = (int *)malloc(2 * sizeof(int));
    int *n2 = (int *)malloc(2 * sizeof(int));
    pthread_t pthread, pthread1;
    int mid = (arg[0] + arg[1]) / 2;
    n1[0] = arg[0];
    n1[1] = mid;
    n2[0] = mid + 1;
    n2[1] = arg[1];
    if (arg[0] < arg[1]){
        pthread_create(&pthread, NULL, merge_sort, n1);
        pthread_create(&pthread1, NULL, merge_sort, n2);
        pthread_join(pthread, NULL);
        pthread_join(pthread1, NULL);
        merge(arg[0], mid, arg[1]);    }}

```

```
void merge(int l, int m, int h){
    int *left = malloc((m - l + 1) * sizeof(int));
    int *right = malloc((h - m) * sizeof(int));
    int n1 = m - l + 1, n2 = h - m, i, j;
    for (i = 0; i < n1; i++)
        left[i] = a[i + l];
    for (i = 0; i < n2; i++)
        right[i] = a[i + m + 1];
    int k = l;
    i = j = 0;
    while (i < n1 && j < n2){
        if (left[i] <= right[j])
            a[k++] = left[i++];
        else
            a[k++] = right[j++];
    }

    while (i < n1){
        a[k++] = left[i++];
    }

    while (j < n2){
        a[k++] = right[j++];
    }
}
```

The driver code calls the merge sort function with initial arguments – zero and the array size – to start the merge sort process. The merge sort process starts a new thread for each division of the array.

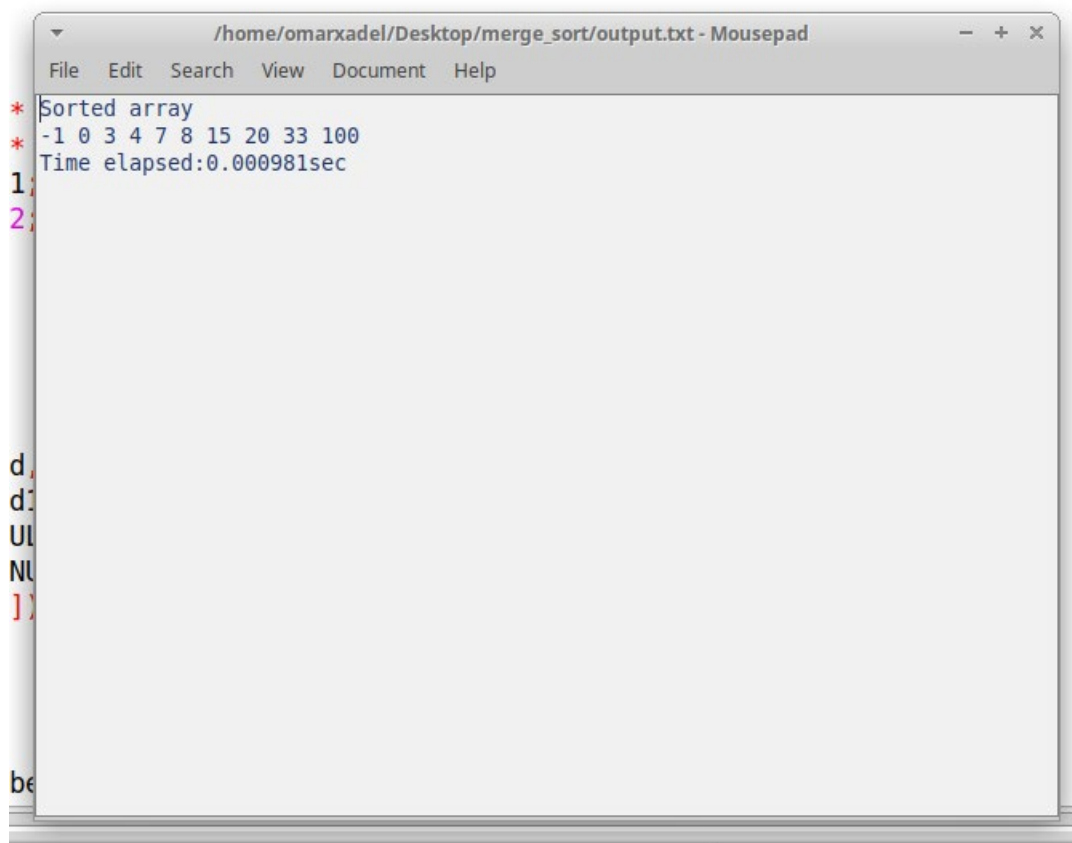
```
int main(){
    // read input file
    read_input("input.txt");

    // specify initial args
    int *pos = (int *)malloc(2 * sizeof(int));
    pos[0] = 0; //lo
    pos[1] = N - 1; //hi

    clock_t begin = clock();
    merge_sort(pos);
    clock_t end = clock();
    t = calculate_time(begin, end);

    output_to_file();
    return 0;
}
```

3.2.2. Output



The screenshot shows a window titled "/home/omarxadel/Desktop/merge_sort/output.txt - Mousepad". The window contains the following text:

```
* Sorted array
* -1 0 3 4 7 8 15 20 33 100
* Time elapsed:0.000981sec
1;
2;
```

4. Conclusion

Multithreading enhances the performance of most of the programs but to a limited extent and it also depends vastly on the CPU that's executing the processes.