IMPLEMENTATION WITH MULTITHREADING PROCESS USING EMBEDDED SYSTEM

1K R JAYACHITRA, 2Dr. L C SIDDANNA GOWD

1Research Scholar, St.Peter’s University, Avadi, Chennai, India.
2Professor, ECE, GRT Institute of Engineering and Technology, Tiruttani, India
Email: jayamura1912000@gmail.com, gouda.lcs@gmail.com

ABSTRACT

In Embedded system based applications, the evolution of multicore architectures offers many performance enhancements like speed, concurrency, real-time implementation support etc. However, design issues like critical section handling, selecting optimal number of threads, racing condition avoidance, concurrent tasks handling etc. needs to be addressed. In this work, these issues are implemented for multicore architecture using openMP tool. Barrier region limitations are removed to exploit concurrency and demonstrated applications include are, (i) array filling multitasking (ii) sorting of number, and (iii) sorting of strings. In all the above examples, the performance of multicore is enhanced compared to single core.

Keyword: Multicore, Openmp, Racing, Multithreading, Embedded System.

1. INTRODUCTION

Parallel programming models are required to exploit multicore architecture and eliminate performance handicapped languages [2,7]. Typical examples of multicore processor are listed in Table 1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Reported cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel SCC</td>
<td>48</td>
</tr>
<tr>
<td>AMF ATI RV7000</td>
<td>10</td>
</tr>
<tr>
<td>NVIDIA Tesla C1060</td>
<td>30</td>
</tr>
<tr>
<td>Intel XEON</td>
<td>4</td>
</tr>
<tr>
<td>ARM MPCore</td>
<td>4</td>
</tr>
</tbody>
</table>

1.1 Power reduction in multiple cores

The power dissipation in single is related as 

\[ P_{\text{single}} = C V^2 f \]

where \( F \) number of times / second the circuit is oscillated [14,15]. In a multicore, the power dissipation is reduced [10] and is given by 

\[ P_{\text{max}} = 0.396 C V^2 f / \] which involves use of synchronization to protect data

1.2 Multithread Implementation

Threads are lightweight processes and share process state among multiple threads and reduce the cost of switch context [8]. Threads interact through reads/writes to a shared address space. The synchronization among threads requires a scheduler and shall determine when to run which threads [16]. The use of multithreads offer performance enhancement [11], but certain issues such as racing avoidance, critical section handling, barrier region, etc. needs to ensure yield correct results.

2. RACE CONDITION IN MULTITHREADING

In a shared address model, unintended sharing of data causes race condition and results in altering the program’s outcome (i.e. data loss) since, the threads are scheduled differently. Solving race conditions involves use of synchronization to protect data.
conflicts (or) change how data is accessed to minimize the excessive use of synchronization [1].

In this paper, fork-join parallelism model is used to transform a sequential algorithm into a parallel one. The code to create multithreads ‘N’ is listed below:

```c
omp_set_num_threads[n]
#pragma OMP parallel
{
    int ID=omp_get_thread_num());
}
```

The `omp_set_num_threads` and `omp_get_thread_num` are runtime function and used to request a certain number of threads [11,13] and returning a thread ID respectively. The first tread is int exe from ‘o’, and the last thread to N-1. Also, only N-1 threads are created since the Nth Parallel section can be invoked from the parent thread and a thread Pool exists to minimize cost of threads and a thread creation and destruction is eliminates for each parallel region. Barrier region refer to the waiting time due to some threads waiting for all the other threads to finish before proceeding.

3. IMPLEMENTATION OF THE WORK
3.1 Array Filling

In this case, an array is to be filled with elements both using single core single thread and multicore multithread [12,17].

The implementation code is listed in Figure 2.

```c
#pragma omp for nowait
for(i=0; i<value; i++)
{
    c[i]=i;
}
#pragma omp for nowait
For (i=value+1;i<value*2+1; i++)
{
    c[i]=1;
}
if (tid==0)
{
    St=omp_get_wtime();
}
```

The results are given in Figure 4 and Figure 5.

### Table 2 Different Shared Variable Application

<table>
<thead>
<tr>
<th>NR_THREADS</th>
<th>No. of multithreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>size of the array i.e. the No. of Multithreads array to be handled by one thread</td>
</tr>
<tr>
<td>tolarrsize</td>
<td>represents the total array size</td>
</tr>
<tr>
<td>C</td>
<td>array of size “tolarrsize”</td>
</tr>
<tr>
<td>Tid</td>
<td>Thread identifier</td>
</tr>
</tbody>
</table>

The meaning of different shared variable in this application is given in Table 2.

![Fig. 3 Threads Implementation](image)
3.2 Sorting of Numbers

In this application, numbers are to be sorted in either ascending or descending order using multithread implementation. For comparison, a non-multithread implementation is also considered. The obtained results are shown in Figure xx.

3.2.1 Multithread Implementation of Sorting Numbers in Ascending Order

```c
if (choice==1)
{
    ifp = fopen (readfile, "r");
    st=omp_get_wtime();
    #pragma omp shared(a,tsortno) private(i,count)
    {
        if ((ifp) == NULL)
        {
            printf("Error in logfile1.txt file 
");
        }
        fscanf (ifp, "%d", &i);
        while (!feof (ifp))
        {
            a[count]=i;
            count=count+1;
            //printf("count value is ",count) ;
            //printf ("%d
", i);
            fscanf (ifp,"%d", &i);
        }
        fclose (ifp);
    }
    ed=omp_get_wtime();
    printf("Total time taken for to ascend by Core: ",ed-st);
    fprintf(logfile_1,"Total time taken for to ascend by Core is : ");
    fprintf(logfile_1,"%f",ed-st);
    fprintf(logfile_1," for ");
    fprintf(logfile_1,"%d",tsortno);
    fprintf(logfile_1," input ");
}
```

3.2.2 Non-multithread Implementation of Sorting Numbers in Ascending Order

```c
if(choice==1)
{
    ifp = fopen (readfile, "r");
    st=omp_get_wtime();
    if ((ifp) == NULL)
    {
        printf("Error in logfile1.txt file 
");
    }
    fscanf (ifp, "%d", &i);
    while (!feof (ifp))
    {
        a[count]=i;
        count=count+1;
        fscanf (ifp,"%d", &i);
    }
    fclose (ifp);
    ed=omp_get_wtime();
    printf("Total time taken for to ascend by CPU: ",ed-st);
    fprintf(logfile_1,"Total time taken for to ascend by CPU is : ");
    fprintf(logfile_1,"%f",ed-st);
    fprintf(logfile_1," for ");
    fprintf(logfile_1,"%d",tsortno);
    fprintf(logfile_1," input ");
}
```

4. RESULTS AND DISCUSSION

In this work, results described the consumed times to sort numbers using OpenMP multithreading in ascending and descending order respectively as shown in Figure 6 and Figure 7.
5. CONCLUSION

This work is concentrated on openMP for multithreading for ascending and descending process to calculate time consumption to sort number and alphabets. The comparison study of ascending and descending order between number (using openMP) and alphabet (using multithreading) is shown the difference to increase speed and reduce power loss in network.

ACKNOWLEDGMENT

I acknowledge the help and facility offered by M/s MicroLogic Systems to carryout the multithreading process and embedded system studies with hardware environment.

REFERENCES


