CHAPTER 5
MULTITHREADING BASED DATA SEARCHING AND SORTING

OpenMP has an advantage in synchronization over hand-threading uses more expensive system calls than present in OpenMP or the code efficient versions of synchronization primitives. As a shared-memory programming paradigm, it is suitable for parallelizing applications on simultaneous multithreaded and multicore processors. It is an API (application program interface) used for explicitly direct multi-threaded, shared memory parallelism to standardize programming extensions for shared memory machines is shown in Figure 5.1.

OpenMP has two key concepts namely;

(i) **Sequential equivalence**: Executes using one thread or many threads.
(ii) **Incremental parallelism**: A programming that evolves incrementally from a sequential program to a parallel program.

![Figure: 5.1 Model for OpenMP Program using threading](image)

The thread scheduling based model with kernel and user space is shown in Figure 5.2. OpenMP applications can efficiently exploit the execution contexts of multithreading processors. The multi-threading models are;
(i) Master-Slave model,
(ii) Worker-Crew model and
(iii) Pipeline model

Figure: 5.2 Multithreading processors using Kernel and User space

5.1 OPENMP ISSUES WITH MULTITHREADING APPROACH

OpenMP specification includes critical, atomic, flush and barrier directives for synchronization purposes as shown in Table 5.1

<table>
<thead>
<tr>
<th>Functions</th>
<th>Purposes</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-inserted locks</td>
<td>Analogous to mutex locks in threads library.</td>
</tr>
<tr>
<td>Critical directive</td>
<td>Associates with a name and all unnamed critical sections map to the same name.</td>
</tr>
<tr>
<td>Flushes directive</td>
<td>Avoid false sharing and placed on different cache lines contiguously.</td>
</tr>
<tr>
<td>Ordered directive</td>
<td>Perform I/O in a sequential order and a parallel loop is avoided in threads.</td>
</tr>
</tbody>
</table>
Effects of OpenMP for Multithreading Process

The effects of OpenMP for multithreading process are listed in Table 5.2.

Table 5.2 Effects of OpenMP

<table>
<thead>
<tr>
<th>Context</th>
<th>Utilization/Accomplished Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaling of OpenMP on multithreading</td>
<td>Effortless by the effects of extensive resource sharing.</td>
</tr>
<tr>
<td>Multicore architectures with multithreading cores</td>
<td>Gain designs to achieve the balance between energy.</td>
</tr>
<tr>
<td>Execute threads within multithreading</td>
<td>Utilized a single level of parallelism.</td>
</tr>
<tr>
<td>Optimization criteria</td>
<td>Energy, die area and performance.</td>
</tr>
</tbody>
</table>

The multithreading is required a solution which is scalable in a number of dimensions and achieve speedups. An efficient parallel program usually limits the number of threads to the number of physical cores that create a large number of concurrent threads. It describes the low-level Linux kernel interface for threads and the programs are invoked by a fork system call which creates a process and followed by an exec system call and loads a program to starts execution. Threads typically end by executing an exit system call, which can kill one or all threads.

Explicit Multithreading using multithreads

The Explicit multithreading is more complex compared to OpenMP and dynamic applications need to be implemented effectively so as to allow user control on performance. The explicit multithreading based multithreads with C coding are shown in Figure 5.3

Scheduling for OpenMP

OpenMP supports loop level scheduling that defines how loop iterations are assigned to each participating thread. The scheduling types are listed in Table 5.3.
Explicit multithreading based coding in C

Figure 5.3 Explicit Multi threading

Table 5.3 Scheduling Types

<table>
<thead>
<tr>
<th>Scheduling Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Each thread is assigned a chunk of iterations in fixed fashion (round robin).</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Each thread is initialized with a chunk of threads, when each thread completes its iterations; it gets assigned the next set of iterations.</td>
</tr>
<tr>
<td>Runtime</td>
<td>Scheduling is deferred until run time. The schedule type and chunk size can be chosen by setting the environment variable OMP_SCHEDULE.</td>
</tr>
<tr>
<td>Guided</td>
<td>Iterations are divided into pieces that successively decrease exponentially, with chunk being the smallest size.</td>
</tr>
</tbody>
</table>
5.2. OPTIMIZING EXECUTION CONTEXTS ON MULTITHREADING PROCESS

The selection of the optimal number of execution contexts for the execution of each OpenMP application is not trivial on multithread based multiprocessors. Thus, a performance-driven, adaptive mechanism which dynamically activates and deactivates the additional execution contexts on multithreading processors to automatically approximate the execution time of the best static selection of execution contexts per processor. It used a mechanism than the exhaustive search, which avoids modifications to the OpenMP compiler and runtime and identifies whether the use of the second execution context of each processor is beneficial for performance and adapts the number of threads used for the execution of each parallel region. The algorithm targets identification of the best loop scheduling policy which is based on the annotation of the beginning and end of parallel regions with calls to runtime. The calls can be inserted automatically, by a simple pre processor. The run-time linking techniques such as dynamic interposition can be used to intercept the calls issued to the native OpenMP runtime at the boundaries of parallel regions and apply dynamic adaptation even to unmodified
application binaries. It modifies the semantics of the OpenMP threads environment variable, using it as a suggestion for the number of processors to be used instead of the number of threads.

5.3. OPENMP MEMORY MODULE TYPES

The memory module types include

(i) Copyin
(ii) Shared.c
(iii) Private.c
(iv) First Private.c
(v) Last Private.c
(vi) Default.c

The description of each is presented in this section. The value of the variable(s) before assigned to the core(s) and upon exit from the core(s) for respective cases is also illustrated

5.3.1. Copyin.c

In this thread private variables are not initialised unless using copyin to pass the value from the corresponding global variables. The value of the thread private variable is maintained throughout the execution of whole program and hence there is no need for copy out. ‘Copyin’ obtains the value of the variable declared under thread private. The value of variable for all threads will be same. In this copy source is the master thread. The following openMP API’s are used in ‘Copyin’

#pragma omp threadprivate(): This routine specifies that a variable is private to a thread
//omp_Set_num_threads(): It sets the number of threads in subsequent parallel regions
#pragma omp parallel copyin()private(): This API allows thread to access the master thread value for a thread private variable

#pragma omp section: It identifies code sections to be divided among all threads

The implementation results is given in Figure 5.4.

![Image of terminal output showing before and after thread incrementation results]

Figure: 5.4 Implemented results of ‘Copyin’

From the results there are two threads namely thread 1 and thread 2. In thread 1(I Core) and thread 2(II Core) the variable value is incremented by one. The thread private variable is initialized with the value of master thread before it exit the parallel region.

5.3.2. Shared.c

The data available inside a parallel region is shared. Here the data can be accessed by all threads simultaneously. The shared variable is available only at a particular memory location and thread can read and write to the particular location only. Loop iteration counter is the only one inside sharing region which is not being shared. The implementation result is shown in Figure 5.5.
Figure: 5.5 Implemented results of ‘Shared.c’

The OpenMP API used in the implementation are

```
#pragma omp parallel shared():  This API specifies that one or more variables
    should be shared among all threads
```

The result obtained shows that the value of the variable both inside and outside the Parallel region is same (shared).

5.3.3. Private.c

The data inside the parallel region is private to each thread. Each thread gets a copy of variable and can be used as a temporary variable. Here the variables are uninitialized in the beginning. Once the parallel ends the memory is freed and the temporary variables are not available. The openMP used in the implementation of ‘Private.c’ are

```
#pragma omp parallel private():  This API specifies that each thread should have
    its own instance of the variable
```

The implementation results are shown in Figure 5.6.
The result obtained shows that the copy of the initial variable is given to each thread and getting executed and once the parallel region is exit the original value is restored i.e \( i=8 \)

### 5.3.4. First private.c

Here the private clause is combined with initialization for a list of variables. First private specifies that each thread should have its own instance of a variable and that the variable should be initialized with the value of the original variable, because it exists before the parallel construct. The openMP API used in this ‘first private.c’ are

```c
#pragma omp parallel firstprivate()
```

It specifies that each thread should have its own instance of a variable and that the variable should be initialised with the value of the variable because its exists before the parallel construct

The obtained results are shown in Figure 5.7
The initial value will come out once the variable exists from the parallel region.

5.3.5. Lastprivate.c

It is similar to private except that original value is updated after the construct here last value of the variable is kept after the parallel region. The openMP API used in ‘lastprivate’ is

```c
#pragma omp parallel for lastprivate():
```

The API specifies that the enclosing context version of the variable is set equal to the private version of whichever thread executes the final iteration.

The implemented result is shown in Figure 5.8

The result shows whatever the last value inside the parallel region will come out as the final value. Here the last value inside the parallel region is 50, the same value is retained at exit of the parallel region.
5.3.6. Default.c

It define the default scope. This particular clause allows the user to state that the default data scoping inside the parallel region may be shared, first private, private or none. For example Default private, Default shared etc. The openMP API used in this clause is

```c
#pragma omp parallel default():
```

It specifies the behaviour of un scoped variables in a parallel region

5.4 MULTIQUEUE IMPLEMENTATION USING SCHEDULER

In this implementation the scheduler inputs the value to the respective queue depending on the key request; during no request according to a first fill queue the data is stored. The scheduler behavior is shown in Table 5.4.
<table>
<thead>
<tr>
<th>Key request</th>
<th>Queue selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>‘l’ or Q₁</td>
</tr>
<tr>
<td>2</td>
<td>‘m’ or Q₂</td>
</tr>
<tr>
<td>3</td>
<td>‘n’ or Q₃</td>
</tr>
<tr>
<td>4</td>
<td>‘o’ or Q₄</td>
</tr>
<tr>
<td>None</td>
<td>‘l’ or Q₁ continues till fill, then ‘m’ or Q₂ continues and so on</td>
</tr>
</tbody>
</table>

**Scheduler Diagram**

Figure 5.9 shows scheduler diagram that shows initial priorities of the queues and how the priority level changes after each queue is filled. This suits systolic array based implementation.

Note:- Priority Pᵢ > Pⱼ for i < j

The various elements allotted to different queues (for illustration) is given in Table 5.5
Table 5.5 Allotment of elements in queue

<table>
<thead>
<tr>
<th>Queues</th>
<th>Q₁</th>
<th>Q₂</th>
<th>Q₃</th>
<th>Q₄</th>
</tr>
</thead>
</table>

State Diagram

The Scheduler state diagram for scheduler for multiple queue implementation is shown in Figure 5.10.

Figure: 5.10 State diagram for scheduler in multiple queue
Implementation Results

Scheduler for multiple queue is implemented in python software and the obtained results are presented in following figures. When key one is pressed then queue ‘l’ is in running mode and other queues are in suspended mode. When there is no key request then queue ‘l’ continues in running mode. When key request two is made then queue ‘m’ is in running mode and other queues are in suspended mode. Similarly queue ‘n’ starts running when key request three is made as illustrated in Figure 5.11.

![Diagram showing implementation results of multiple queue]

**Figure: 5.11 Implementation results of multiple queue**
Each time a queue is selected by the scheduler, one element is filled in the queue. Once all the elements are filled in the queue, then the particular queue will be full, and the highest priority allotted to the next queue as shown in Figure 5.12.

Figure: 5.12 Implementation results of multiple queue (Contd)
When the first queue is full, the elements will be filled in second (m) queue in multiples of three until all the elements are filled and the same process is repeated for third queue (n) in multiples of four and fourth queue (o) in multiples of five elements and so on. The Figure 5.13 shows once the elements are filled in ‘m’ queue, the highest priority assigned to ‘n’ queue.

Figure: 5.13 Implementation results of multiple queue (contd)
Once the queues are filled with data elements data transfer process completed successfully. The Figure 5.14 shows the queues ‘l’, ‘m’, ‘n’, ‘o’ are full.

Figure 5.14 Completion of data transfer in Multiple Queue

5.5. IMPLEMENTATION USING SEMAPHORES

Semaphore is used to represent the status of a resource. To use a shared resource such as I/O port, hardware device or global variable, there is a need to request the semaphore from OS and release the semaphore after access to avoid deadlock. As an example, set of two tasks is taken, but the discussion is applicable for multiple tasks. The tasks within a set, share a variable, but access to it is guarded by a semaphore. Each task can attempt to obtain the semaphore. On obtaining a semaphore, a check is done to ensure that the guarded variable has an expected value. The variable is then cleared before incrementing it back up to the expected value. Between each increment, the variable is checked to ensure that it contains the value to which it was just set to make sure it has not been altered. When the starting value is again reached, the task releases the semaphore to the other task.

Resource sharing/locking can be achieved using binary and counting semaphore. When there are ‘n’ resources and those resources has to be shared among tasks, then if n = 1, binary
A semaphore is used and if n > 1 counting semaphore is used. The real-time system reads temperature values from CAN1 and CAN2 channels and displays its average. To implement these three tasks, task1 (highest priority), task2 (mid priority) and task3 (low priority) are created. The purpose is

(i) One binary semaphore is created to guard the use of the buffer by task1 and task2, and
(ii) counting semaphore to guard the elements in the buffer between task1, task2 and task3.

Task1 and Task2 waits for the availability of binary semaphore, so that one task should not write the data to the buffer when other task is writing the data. Also, Task1 and Task2 waits for the counting semaphore, so that it makes sure that there is some empty space in the buffer to write the data. Otherwise task1 and task2 has to wait until task3 reads some data from the buffer and frees some part of buffer. In task3 critical section is included to avoid task1 and task2 to write data in the buffer when task3 is reading and moving the data positions in the buffer, and prevent data corrupt.

5.5.1. Experimental setup

The real-time setup consists of two LPC 1768 boards. The transmitters and receivers of two CAN boards are connected to one another as cross coupled i.e. Tx1 to Rx2 and Rx1 to Tx2. The serial port of the RTOS board is connected mutually to the computer's serial port. The experimental setup for implementing the semaphore is shown in Figure 5.15.
One of the boards is loaded with CAN program and the other board is loaded with Semaphore program. The semaphore board is connected with hyper terminal (115200 baud rate) to view the output.

5.5.2. State Diagram (Semaphore)

The state diagram for Task 1, Task 2 and Task 3 are shown in Figure 5.16 and Figure 5.17.
Figure: 5.16 State diagram for task 1 (semaphore)
Figure: 5.17 State diagrams for Task 2 and Task 3

5.5.3. Development cycle for semaphore

The development cycle starts with the design phase where in, the algorithms for task1, task2 and task3 are designed.
Algorithm for Task 1:

(i) Delay for 3 sec
(ii) Wait for CAN1 data
(iii) Read CAN1 data
(iv) Print Data on Hyperterminal
(v) Wait for Buffer_lock binary semaphore
(vi) Wait for Buffer_Count counting semaphore
(vii) Store the CAN1 Data in Buffer
(viii) Release the Buffer lock semaphore
(ix) Go back to step (ii)

Algorithm for Task 2:

(i) Delay for 2 sec
(ii) Wait for CAN2 data
(iii) Read CAN2 data
(iv) Print Data on Hyperterminal
(v) Wait for Buffer_lock binary semaphore
(vi) Wait for Buffer_Count counting semaphore
(vii) Store the CAN2 Data in Buffer
(viii) Release the Buffer lock semaphore
(ix) Go back to step (ii)

Algorithm for Task 3:

(i) Delay for 1 second
(ii)  check if any data in buffer

(iii) if data is present, then disable interrupts

(iv)  Read CAN data from buffer

(v)  Release Buffer_Count Semaphore

(vi)  Calculate Average of CAN data

(vii) Release the Buffer_Lock Binary Semaphore

(viii) if data not present in step3 then delay for 2 seconds

(ix)  Go back to Step (ii)

The above algorithms are implemented using RTOS functions App_Task1Create(), App_Task2Create() and App_Task3Create(). Stack of sufficient size is created for the task to run properly. This is done by passing address of the stack in OSTaskCreate API. The priority of the task is also passed to the API. Two semaphores are created one for binary and another for counting in Sem_init() function. In task3() OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() macros are used to disable and enable interrupts respectively.

5.5.4. Code Implementation in C

Task Creation And Semaphore Init Code

```c
void App_Task1Create (void)
{
    CPU_INT08U os_err;

    os_err = os_err; /* prevent warning... */
    UART0_SendString("Creating Task 1 with priority 56...\r\n");
    os_err = OSTaskCreate((void (*)(void *)) uctsk_Task1,
                          (void *) 0,
```
(OS_STK *) &App_Task1Stk[APP_TASK_STK_SIZE - 1],
(INT8U ) APP_TASK1_Prio);

#if OS_TASK_NAME_EN > 0

Task 1

static void uctsk_Task1(void *pdata)
{

    char *ptr;
    INT8U err=0;
    OS_SEM_DATA sem_data;
    float Temp;
    ptr=(char *)&Temp;
    UART0_SendString("Task1 is Created\r\n");

    OSTimeDlyHMSM(0,0,3,0);

Task 2

static void uctsk_Task2()
{

    INT8U err=0;
    OS_SEM_DATA sem_data;
    char *ptr;
    float Temp;
    ptr=(char *)&Temp;

    UART0_SendString("Task2 is Created \r\n");
Task 3

static void uctsk_Task3()
{
    if (OS_CRITICAL_METHOD == 3) /* Allocate storage for CPU status register */
    OS_CPU_SR cpu_sr;
    #endif

    int i=0;
    OS_SEM_DATA sem_data;
    float temp;
    UART0_SendString("Task3 is Created \n\n");
    OSTimeDlyHMSM(0,0,1,0);

    OSTimeDlyHMSM(0,0,2,0);

    The waiting queue is emptied by the reset operation by both binary and counting
semaphores and the wait state is terminated when semaphore is not signaled within the allotted
time thereby the resource is not locked permanently by a slow priority process. The
fundamental requirement of a multitasking system is to detect occurrence of an event and then
to synchronize a task with that event (synchronous or asynchronous). The requirement is
satisfied by semaphore.

5.6. CHAPTER SUMMARY

    In this chapter a brief overview about the openMP tools is presented. Discussion on
openMP memory module types is made with the implemented results of various modules like
copyin, shared.c etc. Multiple queue implementation using scheduler is presented for four
queues and finally Semaphore concept is realized in the LPC 1768 and µC/OS-II environment.