Trabajo de Investigación sobre OpenMP

Patrick Xavier Márquez Choque
Univesidad Católica San Pablo
Arequipa, Perú
Email: patrick.marquez@ucsp.edu.pe

Abstract Este trabajo tiene como objetivo investigar la sección 5 del libro de Peter Pacheco que menciona el problema del productor-consumidor y realizar una implementación de este programa mencionado en el libro aplicando las directivas críticas, atómicas y los locks de OpenMP mencionadas en el libro.

Todas las pruebas de ejecución de los programas implementados se utilizan con un Hardware Promedio con una CPU Intel (R) Pentium(R) CPU G450 @ 3.50GHz con 2 Núcleos y 4 Procesadores lógicos con 12.0GB de Memoria RAM.

El link al repositorio donde se encuentra implementado los códigos mencionados se encuentra en el siguiente enlace: https://github.com/patrick03524/Programacion-Paralela-y-Distribuida/tree/main/SeptimoTrabajo

1 Problema del Productor-Consumidor

Este problema paralelo no apto en algunos casos de paralelización, el libro menciona varios elementos como:

- Queues: La 'Cola' es un tipo de dato abstracto basado en una lista donde se insertan nuevos elementos en la parte posterior de la cola y se remueven al frente de la misma, un concepto ya muy conocido en el mundo de la computación pero el problema paralelo del paradigma Productor-Consumidor de este concepto se puede entender con el siguiente ejemplo: Supongamos que tenemos un productor de threads y varios consumidores de estas threads. El productor quiere producir threads para los consumidores y las requerirá de un servidor preguntando si tiene este recurso disponible, como si en una tienda se pregunta por el stock de un producto y en caso no se tenga disponible el productor puede 'congelar' esta cola de objetos que necesitan los consumidores de tal manera que no se completaría el trabajo hasta que todos los consumidores terminen de usar las threads y los datos requeridos del productor.

doi:10.1017/xxxxxxx xxxx

- Sincronización: Llamados como envió y recibo de mensajes dentro de un medio donde la sincronización varia dependiendo del contexto ya que por ejemplo en los casos de las colas el envió de mensajes es crítico para esta sección ya que dependiendo de los detalles de la implementación de los productores es como preguntarán o dejaran disponible el recurso de las threads para los consumidores.

  En la implementación se logró la obtención de código mostrado en el libro con las dos principales directivas para medir la sincronización entre el productor y el consumidor que son la directiva atómica y la crítica utilizando locks.

- Directiva Atómica: Después de completar todos los envíos cada threads incrementa el envió antes de proceder al final de cada bucles. Esto permite que en secciones críticas se tenga una protección de OpenMP proporciona una directiva de rendimiento mucho más eficiente.

  Pragma omp atomic a diferencia de la directiva crítica, esta solo puede proteger secciones críticas que constan de una única asignación de memoria como asignaciones con operadores ++ o – para el desplazamiento o las operaciones binarios.

- Directiva Crítica: Esta directiva depende de un concepto más complejo que la atómica por ejemplo tenemos una cola que se ejecuta simultáneamente de tal manera que cada productor cuando libere una thread o la requiera una para sus consumidores entonces se bloqueará el acceso a los bloques de memoria de diferentes colas pero esto en ciertos casos esto puede traer insuficiencias así que para eso se implemento los locks, estos locks consiste de una estructura de datos con funciones que nos permiten dar una exclusión de parte de otros threads en una sección crítica así que de tal manera que es un lock simple pero que asegura la seguridad dentro de las llamadas que se hicieron.

2 Implementación

La implementación se realizó a partir del libro de cada una de las directivas mencionadas realizándose pruebas que luego serán analizadas en la siguiente sección.

2.1 Directiva Atómica

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 #include "queue.h"
5 #include "timer.h"
6
7 const int MAX_MSG = 10000;
8
```

```
9  void Usage(char* prog_name);
10 void Send_msg(struct queue_s* msg_queues[], int my_rank,
11       int thread_count, int msg_number);
12 void Try_receive(struct queue_s* q_p, int my_rank);
13 int Done(struct queue_s* q_p, int done_sending, int thread_count);
14
15 /*--------------------------------------------------------------------
       */
16 int main(int argc, char* argv[]) {
17    int thread_count;
18    int send_max;
19    struct queue_s** msg_queues;
20    int done_sending = 0;
21    double start, finish, elapsed;
22
23    GET_TIME(start);
24
25    if (argc != 3) Usage(argv[0]);
26    thread_count = strtol(argv[1], NULL, 10);
27    send_max = strtol(argv[2], NULL, 10);
28    if (thread_count <= 0 || send_max < 0) Usage(argv[0]);
29    msg_queues = malloc(thread_count*sizeof(struct queue_node_s*));
30 #  pragma omp parallel num_threads(thread_count) \
31       default(none) shared(thread_count, send_max, msg_queues,
      done_sending)
32    {
33       int my_rank = omp_get_thread_num();
34       int msg_number;
35       srandom(my_rank);
36       msg_queues[my_rank] = Allocate_queue();
37
38 #     pragma omp barrier /* Don't let any threads send messages  */
39                          /* until all queues are constructed     */
40       for (msg_number = 0; msg_number < send_max; msg_number++) {
41          Send_msg(msg_queues, my_rank, thread_count, msg_number);
42          Try_receive(msg_queues[my_rank], my_rank);
43       }
44 #     pragma omp atomic
45       done_sending++;
46       while (!Done(msg_queues[my_rank], done_sending, thread_count))
47          Try_receive(msg_queues[my_rank], my_rank);
48       Free_queue(msg_queues[my_rank]);
49       free(msg_queues[my_rank]);
50    }  /* omp parallel */
51
52    free(msg_queues);
53    GET_TIME(finish);
54    elapsed = finish - start;
55    printf("The elapsed time is %e seconds\n", elapsed);
56    return 0;
57 }  /* main */
58
59 void Usage(char *prog_name) {
60    fprintf(stderr, "usage: %s <number of threads> <number of messages
       >\n",
```

```
61          prog_name);
62    fprintf(stderr, "   number of messages = number sent by each thread
       \n");
63    exit(0);
64 }  /* Usage */
65
66 void Send_msg(struct queue_s* msg_queues[], int my_rank,
67       int thread_count, int msg_number) {
68 // int mesg = random() % MAX_MSG;
69    int mesg = -msg_number;
70    int dest = random() % thread_count;
71 #  pragma omp critical
72    Enqueue(msg_queues[dest], my_rank, mesg);
73 }  /* Send_msg */
74
75 void Try_receive(struct queue_s* q_p, int my_rank) {
76    int src, mesg;
77    int queue_size = q_p->enqueued - q_p->dequeued;
78
79    if (queue_size == 0) return;
80    else if (queue_size == 1)
81 #     pragma omp critical
82       Dequeue(q_p, &src, &mesg);
83    else
84       Dequeue(q_p, &src, &mesg);
85    //printf("Thread %d > received %d from %d\n", my_rank, mesg, src);
86 }   /* Try_receive */
87
88 int Done(struct queue_s* q_p, int done_sending, int thread_count) {
89    int queue_size = q_p->enqueued - q_p->dequeued;
90    if (queue_size == 0 && done_sending == thread_count)
91       return 1;
92    else
93       return 0;
94 }   /* Done */
```

Listing 1: Directiva Atómica

2.2 Directiva Crítica

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4  #include "queue_lk.h"
5  #include "timer.h"
6
7  const int MAX_MSG = 10000;
8
9  void Usage(char* prog_name);
10 void Send_msg(struct queue_s* msg_queues[], int my_rank,
11       int thread_count, int msg_number);
12 void Try_receive(struct queue_s* q_p, int my_rank);
13 int Done(struct queue_s* q_p, int done_sending, int thread_count);
```

```
14
15 /*--------------------------------------------------------------------
       */
16 int main(int argc, char* argv[]) {
17    int thread_count;
18    int send_max;
19    struct queue_s** msg_queues;
20    int done_sending = 0;
21    double start, finish, elapsed;
22
23    GET_TIME(start);
24
25    if (argc != 3) Usage(argv[0]);
26    thread_count = strtol(argv[1], NULL, 10);
27    send_max = strtol(argv[2], NULL, 10);
28    if (thread_count <= 0 || send_max < 0) Usage(argv[0]);
29
30    msg_queues = malloc(thread_count*sizeof(struct queue_node_s*));
31
32 #  pragma omp parallel num_threads(thread_count) \
33       default(none) shared(thread_count, send_max, msg_queues,
       done_sending)
34    {
35       int my_rank = omp_get_thread_num();
36       int msg_number;
37       srandom(my_rank);
38       msg_queues[my_rank] = Allocate_queue();
39
40 #     pragma omp barrier /* Don't let any threads send messages  */
41                          /* until all queues are constructed     */
42
43       for (msg_number = 0; msg_number < send_max; msg_number++) {
44          Send_msg(msg_queues, my_rank, thread_count, msg_number);
45          Try_receive(msg_queues[my_rank], my_rank);
46       }
47 #     pragma omp atomic
48       done_sending++;
49
50       while (!Done(msg_queues[my_rank], done_sending, thread_count))
51          Try_receive(msg_queues[my_rank], my_rank);
52
53       /* My queue is empty, and everyone is done sending
       */
54       /* So my queue won't be accessed again, and it's OK to free it
       */
55       Free_queue(msg_queues[my_rank]);
56       free(msg_queues[my_rank]);
57    }  /* omp parallel */
58
59    free(msg_queues);
60    GET_TIME(finish);
61    elapsed = finish - start;
62    printf("The elapsed time is %e seconds\n", elapsed);
63    return 0;
64 }  /* main */
```

```
65
66 void Usage(char *prog_name) {
67    fprintf(stderr, "usage: %s <number of threads> <number of messages
       >\n",
68          prog_name);
69    fprintf(stderr, "   number of messages = number sent by each thread
       \n");
70    exit(0);
71 }  /* Usage */
72
73 void Send_msg(struct queue_s* msg_queues[], int my_rank,
74       int thread_count, int msg_number) {
75 // int mesg = random() % MAX_MSG;
76    int mesg = -msg_number;
77    int dest = random() % thread_count;
78    struct queue_s* q_p = msg_queues[dest];
79    omp_set_lock(&q_p->lock);
80    Enqueue(q_p, my_rank, mesg);
81    omp_unset_lock(&q_p->lock);
82 }  /* Send_msg */
83
84 void Try_receive(struct queue_s* q_p, int my_rank) {
85    int src, mesg;
86    int queue_size = q_p->enqueued - q_p->dequeued;
87
88    if (queue_size == 0) return;
89    else if (queue_size == 1) {
90       omp_set_lock(&q_p->lock);
91       Dequeue(q_p, &src, &mesg);
92       omp_unset_lock(&q_p->lock);
93    } else
94       Dequeue(q_p, &src, &mesg);
95    //printf("Thread %d > received %d from %d\n", my_rank, mesg, src);
96 }   /* Try_receive */
97
98 int Done(struct queue_s* q_p, int done_sending, int thread_count) {
99    int queue_size = q_p->enqueued - q_p->dequeued;
100   if (queue_size == 0 && done_sending == thread_count)
101      return 1;
102   else
103      return 0;
104 }   /* Done */
```

Listing 2: Directiva Crítica

2.3 Pruebas

En primer lugar las pruebas fueron realizadas considerando desde 1 hasta 32 threads para tener mejores observaciones sobre diferentes recepciones y envíos de mensajes, desde 2 hasta 100 mensajes en cada una de las threads para tener un mejor entendimiento de las conclusiones de este trabajo.

Directiva OMP Parallel							
Tiempo:	Mensajes	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
Queue Directiva Atómica	2	0.00226	0.00110	0.00291	0.37863	2.44260	14.00162
	4	0.00213	0.00142	0.00259	0.84447	7.12424	21.40150
	10	0.00674	0.00261	0.00517	0.89331	13.57999	34.50912
	50	0.03018	0.01108	0.02217	0.27440	15.48583	86.40705
	100	0.05203	0.10307	0.21825	0.21465	44.23025	87.45622
Queue Directiva Crítica con Locks	2	0.00003	0.00021	0.00040	0.03760	0.10661	0.24511
	4	0.00003	0.00022	0.00050	0.03468	0.13909	0.24641
	10	0.00003	0.00020	0.00059	0.03950	0.10549	0.27077
	50	0.00004	0.00025	0.00043	0.03734	0.10886	1.88799
	100	0.00004	0.00026	0.00055	0.03809	0.10699	3.64950

Figure 1: Cuadro comparativo de las Directivas

- Lo que podemos apreciar a primer avista es la gran diferencia de tiempos que se tienen entre cada una de las directivas, entonces lo que se puede concluir es el caso estudiado de la primer Directiva atómica ya que está demora mucho más en caso que se creen muchas más threads y por lo tanto durante el envió de mensajes es la que tendrá peor rendimiento a comparación de la directiva crítica ya que está implementada con locks, esto permite que aún pese a la gran complejidad de la directiva crítica esta se puede tener un mejor rendimiento y es gracias a las implementaciones de ambos programas.

- Debido al paradigma de la implementación de la cola como una estructura para el problema de productor-consumidor es que se realizó la implementación de la cola paralela que se encuentra en el libro de Peter Pacheco, es por eso que cuando se exceden con más threads es que se tienen peores tiempos y por lo tanto peor rendimiento.

- Esta implementación de los locks en el caso de la segunda directiva fue realizada ya que se implementó con locks en la cola modificada, esto permite un mejor rendimiento es necesario. Todos los códigos e implementaciones están en el link de github.