Universidad Rovira i Virgili

Escuela Técnica Superior de Ingeniería

ARQUITECTURA DE COMPUTADORES

PRÁCTICA 3: Procesadores MultiThread-Multicore:

Aplicación multithread

AUTORES:

YANI AICI LOUNIS

IVAN MORILLAS GÓMEZ

DOCENTE:

CARLES ALIAGAS CASTELL

14/01/2024

2023-2024

ÍNDICE

LEY DE AMDAHL

La ley de Amdahl establece que la mejora obtenida en el rendimiento de un sistema en debido a la alteración de uno de sus componentes, está limitada por la fracción de tiempo que utiliza dicho componente. El rendimiento de los algoritmos se puede determinar por el tiempo de ejecución. En los algoritmos secuenciales este tiempo de ejecución depende del tamaño de datos de entrada y la velocidad de computación. En cambio, el tiempo de ejecución a un algoritmo paralelo depende del tamaño de los datos, la velocidad de computación, el número de procesos, la comunicación entre procesos y una combinación entre el algoritmo y la arquitectura de la que se ejecuta. La medida que tenemos para el rendimiento y la ejecución de un código paralelo es el Speed-Up. El Speed-Up podemos definirlo como la aceleración que produce un algoritmo paralelo respecto a su versión secuencial. Las fórmulas que tenemos para calcular el Speed-Up son las siguientes:

$$Sn = \frac{1}{F + \frac{(1-F)}{n}} \qquad S(n) = \frac{T1}{Tn}$$

T1: tiempo de ejecución secuencial.

Tn: tiempo de ejecución paralelizado.

F: fracción no paralelizable.

n: número de threads.

Para la realización de esta práctica, hemos utilizado dos procesadores con las siguientes características:

Intel Xeon E5-2690 (teen)

Frecuencia de ejecución multicore: 2.9 GHz Cores: 8 x 2 = 16

Threads: (8 x 2) x 2 = 32

AMD Ryzen Threadripper PRO 3995WX (orca)

Frecuencia de ejecución multithread: 2.7 GHz Cores: 64

Threads: 64 x 2 = 128

PARALELIZACIÓN DEL CÓDIGO

En primer lugar, hemos implementado dos estructuras adicionales para poder pasar los argumentos necesarios de las funciones *QuickSort* y *Merge*:

```c
//Struct para los argumentos del QuickSort
struct qs_args{
  int *val;
  int ne;
};
```

```c
//Struct para los argumentos del Merge
struct merge2_args{
  int *val;
  int n;
  int *vo;
};
```

También hemos implementado dos nuevas funciones con *QuickSort* y *Merge* para poder trabajar con *threads* solo cambiando la cabecera y la declaración de sus variables:

```c
//Funcion QuickSort con threads
void *qs_thread(void *qs_args){
  struct qs_args *args = qs_args;
  int *val = args->val;
  int ne = args->ne;
```

```c
//Funcion Merge con threads
void *merge2_thread(void *merge2_args){
  struct merge2_args *args = merge2_args;
  int *val = args->val;
  int n = args->n;
  int *vo = args->vo;
```

Donde tenemos de parámetros los *structs* mencionados anteriormente, declaramos la variable **args* como un *struct* e inicializamos las variables necesarias para cada función.

Dentro del main tenemos que declarar los *threads* que vamos a utilizar y los *structs* con los argumentos necesarios para ambas funciones:

```c
pthread_t threads[parts]; //Declarar el thread
struct qs_args args_qs[parts]; //Declarar el struct de los argumentos del QuickSort
struct merge2_args args_merge2[parts]; //Declarar el struct de los argumentos del Merge
```

Después, con un bucle, vamos creando los *threads* paralelamente donde hay el identificador, la función y sus argumentos correspondientes y, previamente, inicializar los argumentos del *struct*. Y para evitar que un *thread* no empiece antes de que acabe otro, usaremos el *pthread_join()* para que los *threads* acaben su ejecución:

```c
//Bucle para crear threads del QuickSort
for (i=0;i<parts;i++){
  //printf("de %d a %d\n",i*ndades/parts,(i+1)*(ndades/parts));
  args_qs[i].val = &valors[i*(ndades/parts)];
  args_qs[i].ne = ndades/parts;
  pthread_create(&threads[i], NULL, qs_thread, &args_qs[i]); //Crear los threads
}

//Bucle de espera de finalizacion de todos los threads
for(i = 0; i < parts; i++){
  pthread_join(threads[i], NULL);
}
```

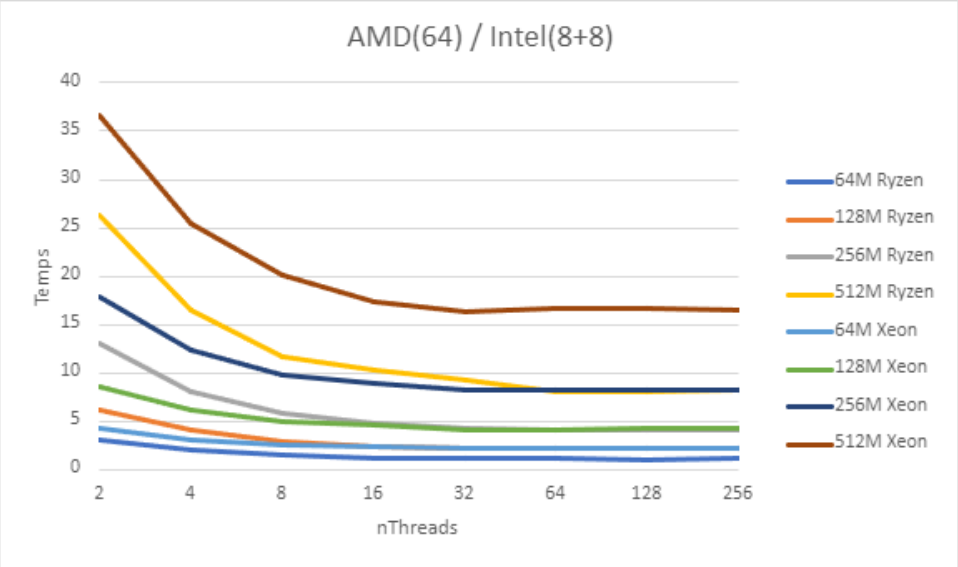
```
//Bucle para crear threads del Merge
for (m = 2*ndades/parts; m <= ndades; m *= 2){
  j=0;
  for (i = 0; i < ndades; i += m){
    args_merge2[j].val = &vin[i];
    args_merge2[j].n = m;
    args_merge2[j].vo = &vout[i];
    pthread_create(&threads[j], NULL, merge2_thread, &args_merge2[j]);
    j++;
  }
  //Bucle de espera de finalizacion de todos los threads
  for (i = 0; i < j; i++){
    pthread_join(threads[i], NULL);
  }
  vtmp=vin;
  vin=vout;
  vout=vtmp;
}
```

ANALISIS DE LOS RESULTADOS DE LA EJECUCIÓN

Para poder ejecutar el código, hemos tenido que entrar al servidor zoo y compilamos el codigo dandonos este resultado:

```
21162507-T@zoo:~$ srun -p teen -c 32 time ./qs_merge_paralel 64000000 2
srun: job 23910387 queued and waiting for resources
srun: job 23910387 has been allocated resources
validacio 687225066797337
Tiempo bucles Sort: 5.91526
6.92user 0.12system 0:04.30elapsed 163%CPU (0avgtext+0avgdata 504004maxresident)k
40inputs+0outputs (1major+595minor)pagefaults 0swaps
```

Donde nos fijamos en el *elapsed* para poder hacer el siguiente estudio:

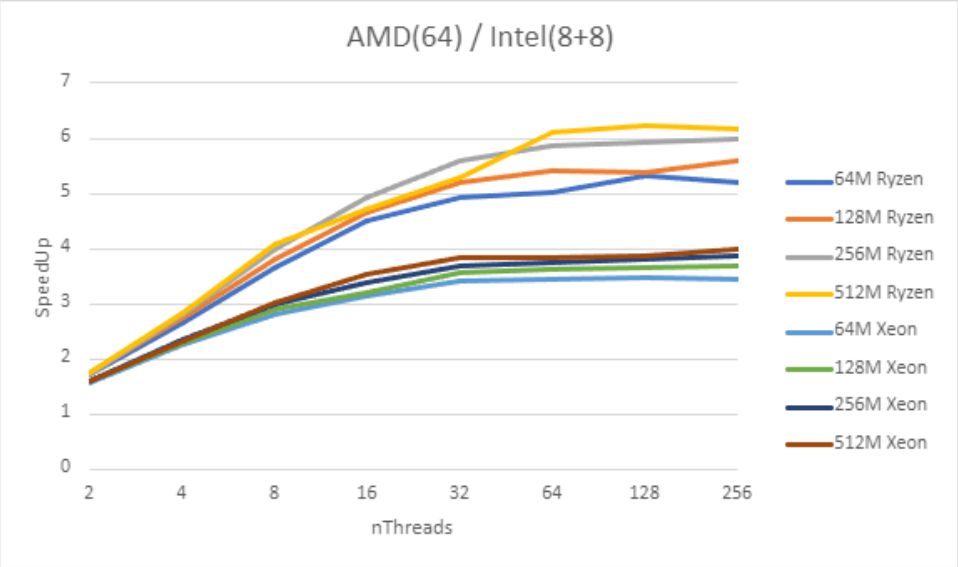


En esta gráfica podemos observar el tiempo de ejecución del código paralelizado en los servidores *teen* (Xeon) y *orca* (Ryzen) donde vemos que, a medida que aumentamos el número de *threads* con el que ejecutamos el código, el coste temporal va disminuyendo hasta llegar a un punto donde se estabiliza.

Para poder hacer el estudio del Speed-Up real, teníamos que obtener los resultados del código secuencial y del código paralelizado donde también miramos el tiempo *elapsed* y poder calcular el Speed-Up real mencionado anteriormente:

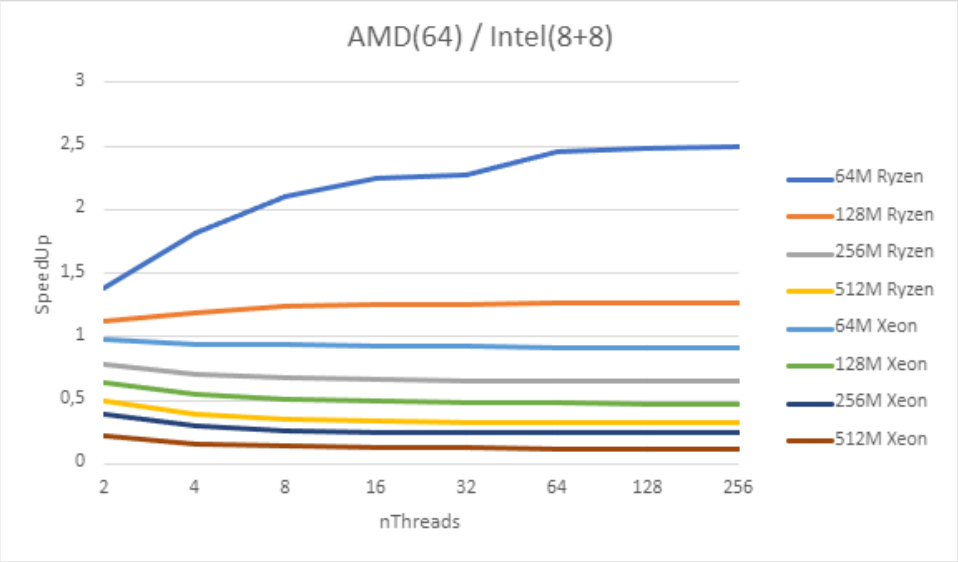
```
21162507-T@zoo:~$ srun -p orca -c 128 time ./qs_merge_ac 64000000 2
srun: job 23897834 queued and waiting for resources
srun: job 23897834 has been allocated resources
validacio 687225066797337
Temps bucle: 4.86961
5.23user 0.04system 0:05.28elapsed 99%CPU (0avgtext+0avgdata 503436maxresident)k
0inputs+0outputs (0major+376minor)pagefaults 0swaps
```

Y de resultado nos da esto:



En esta gráfica podemos observar la mejora del rendimiento obtenido al ejecutar el código en diferentes *threads*. En conclusión, podemos decir que cuanto más *threads* haya en la ejecución, mejor rendimiento tendrá y más rápido será y en comparación con ambos procesadores, podemos concluir que el multithread tiene mejor rendimiento que el multiprocesador.

Seguidamente, hemos hecho un estudio del Speed-Up teórico siguiendo la formula, explicada anteriormente, donde teníamos que saber la fracción no paralelizable, que lo hemos calculado restando el tiempo *elapsed* con el tiempo de los bucles de *Sort* medido con la función *clock(),* y los números de *threads* usados para la ejecución:



Como se puede observar, no vemos una clara mejoría, exceptuando la primera ejecución del servidor orca con un vector de 64M.