Departament d'Enginyeria Informàtica i Matemàtiques
[DEIM] Universitat Rovira i Virgili

Arquitectura de Computadores
AC
Curso 17/18
Primera Convocatoria
Pràctica 1: Simulador Superscalar *Simplescalar*

Arquitectura de Computadors

Procesadores MultiThread-Multicore:
Aplicación multithread

Aleix Mariné i Tena

Cristòfol Daudèn Esmel

DG: Biotecnologia + Enginyeria Informàtica

Índex

Introducció

En aquesta pràctica treballarem sobre dos màquines diferents:

- Màquina multiprocessador amb 4 processadors unicore 4x Opteron 844 a 1.8Ghz, que es comuniquen entre ells mitjançant links directes i accedeixen a una memòria de 4GB mitjançant.
- Màquina multithread amb 2 processadors, 8 cores per processador i 8 threads per core, en total pot executar 128 threads simultàniament. La màquina es tracta d'un Sun Enterprise T5240 a 1.4 Ghz.

Sobre aquestes màquines executarem un codi determinat i n'obtindrem uns resultats que posteriorment analitzarem. Aquest codi donat, un cop paral·lelitzat, l'executarem amb x nombre de threads i sobre un vector que pot tenir x elements. Aquest codi serà una versió més senzilla del quickSort.

Mesurarem el temps d'execució i el SpeedUp (Temps d'execució seqüencial / Temps d'execució paral·lelitzat).

Paral·lelització de l'Algorisme

Algorisme:

- Es dividirà el vector en parts i és farà una ordenació local d'aquests segments del vector.
- Després és cridarà al merge, aquest el que farà és anar comparant 2 a 2 els elements més petits de cada segment i els anirà ordenant, de forma que s'haurà d'executar $\log_2$(Parts en les que s'ha dividit el vector).

Per separar la fase de sort i la de merge necessitem una barrera, de la mateixa forma que en les diverses etapes de merge → mutex

Hem agut d'implementar dues estructures addicionals per poder passar els arguments necessaris als fils d'execució de la funció *quickSort* i *merge*:

```
typedef struct{
    int *val;
    int ne;
}arg_qs_struct;
```

```
typedef struct{
    int *val;
    int n;
    int *vo;
}arg_merge_struct;
```

A més, hem hagut de implementar una nova funció per al quickSort i *merge* perquè es pugin executar des de diversos fils (modificació de la capçalera i declaració de les variables passades a través de les estructures implementades), tot i que mantindrem la funció *quickSort* ja donada per les crides recursives dins del mateix fil:

```
void * qs_thread(void * arguments)
 {
  //Redefinim les variables
  arg_qs_struct *args = arguments;
  int ne = (*args).ne;
  int *val = (*args).val;
  int i,f,j;
  int pivot,vtmp,vfi;
```

```
void * merge_thread(void * arguments)
 {
  //Redefinim les variables
  arg_merge_struct *args = arguments;
  int n = (*args).n;
  int *val = (*args).val;
  int *vo = (*args).vo;
  int vtmp;
  int i,j,posi,posj;
```

En el *main* hem de instanciar les dues estructures implementades i també una variable que ens apunti a l'identificador de cada fil:

```
//VAriables necessàries per crear els threads
pthread_t threads[parts];
arg_qs_struct args_qs[parts];
arg_merge_struct args_merge[parts];
```

A més, enlloc de realitzar diverses crides seqüencials a la funció *qs* i *merge2* crearem diversos fils que s'executaran paral·lelament, per evitar que no comenci el *merge* abans no acabi el *sort* i entre els diferents nivells de *merge*, farem ús de la funció pthread_join(), que s'espera fins que acabi l'execució dels fils.

```
// Quicksort a parts, Cada part l'hauria de fer un thread !!
  for (i=0;i<parts;i++)
  {
    //qs(&valors[i*ndades/parts],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]);
  }

  //Esperem a que finalitzen tots els threads
  for(i=0;i<parts;i++){
    pthread_join(threads[i], NULL);
  }
```

```
vin = valors;
vout = valors2;
for (m = 2*ndades/parts; m <= ndades; m *= 2)
{
  j=0;
  for (i = 0; i < ndades; i += m){
    //merge2(&vin[i],m,&vout[i]);
    args_merge[j].val = &vin[i];
    args_merge[j].n = m;
    args_merge[j].vo = &vout[i];
    pthread_create(&threads[j], NULL, merge_thread, &args_merge[j]);
    j++;

  }
  for (i = 0; i < j; i++){
    pthread_join(threads[i], NULL);
  }
  vtmp=vin;
  vin=vout;
  vout=vtmp;
}
```

Execució del codi

Primer hem de configurar el túnel per poder accedir al servidor zoo:

ssh -NfL 8055:zoo:22 47475627-Q@portal1-deim.urv.cat

Copiem les dades (codi .c) al servidor zoo:

scp qs_merge_ac.c 47475627-Q@zoo.lab.deim:.

Accedim al servidor:

ssh -X -p 8055 47475627-Q@localhost

Compilem el codi per al gat (Màquina multiprocessador), prèviament hem hagut d'afegir un exit(0) al final del main perquè l'execució sigues correcta:

cc -O qs_merge_ac.c -o qs_merge_ac –lpthread

Executem el codi:

srun -p gat time ./qs_merge_ac [mida array] [numero de threads]

En la següent imatge es mostra un exemple de l'execució del codi amb diferent nombre de therads, també podem observar que el valor de validació es manté constant en totes les execucions:

```
47475627-Q@zoo:~$ srun -p gat time ./qs_merge_ac 8000000 8
srun: job 61620 queued and waiting for resources
srun: job 61620 has been allocated resources
validacio 85906444506139
1.81user 0.08system 0:00.74elapsed 255%CPU (0avgtext+0avgdata 63124maxresident)k
0inputs+0outputs (0major+15820minor)pagefaults 0swaps
47475627-Q@zoo:~$ srun -p gat time ./qs_merge_ac 8000000 16
validacio 85906444506139
1.86user 0.05system 0:00.74elapsed 257%CPU (0avgtext+0avgdata 63128maxresident)k
0inputs+0outputs (0major+15845minor)pagefaults 0swaps
47475627-Q@zoo:~$ srun -p gat time ./qs_merge_ac 8000000 32
validacio 85906444506139
1.86user 0.07system 0:00.75elapsed 256%CPU (0avgtext+0avgdata 63136maxresident)k
0inputs+0outputs (0major+15902minor)pagefaults 0swaps
47475627-Q@zoo:~$ srun -p gat time ./qs_merge_ac 8000000 64
validacio 85906444506139
1.90user 0.08system 0:00.77elapsed 256%CPU (0avgtext+0avgdata 63160maxresident)k
0inputs+0outputs (0major+16024minor)pagefaults 0swaps
47475627-Q@zoo:~$ srun -p gat time ./qs_merge_ac 8000000 128
validacio 85906444506139
1.93user 0.10system 0:00.78elapsed 258%CPU (0avgtext+0avgdata 63312maxresident)k
0inputs+0outputs (0major+16278minor)pagefaults 0swaps
```

Compilem el codi per la màquina Cos:

srun -p tau excos gcc -O qs_merge_ac.c -o qs_merge_ac –lpthread

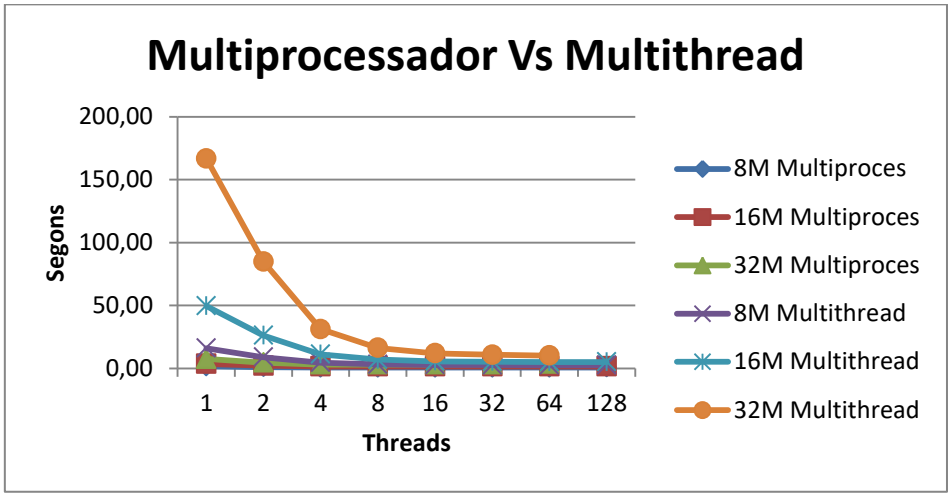
Executem el codi:

srun -p tau excos time ./qs_merge_ac 8000000 2

Anàlisi dels resultats

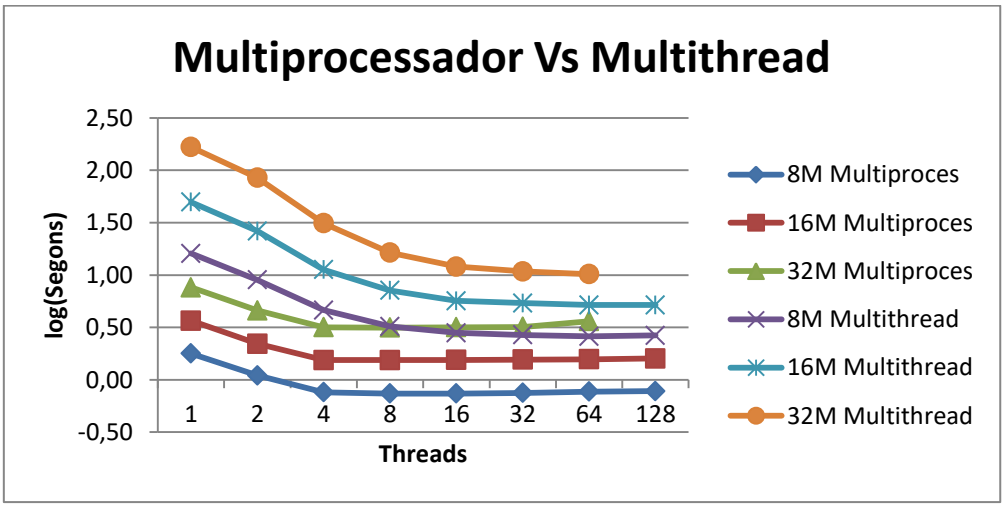
Com es observar en la gràfica, en la màquina multiprocés el codi s'executa molt més ràpid que en la màquina *multithread*, fins al punt de que l'ordenació d'un *array* de 32 M s'executa més ràpid que l'ordenació de *l'array* de 8 M.

Per veure millor els resultats hem representat en una altra gràfica el logaritme del temps que ha trigat en executar-se el codi en funció del nombre de fils que s'han creat.

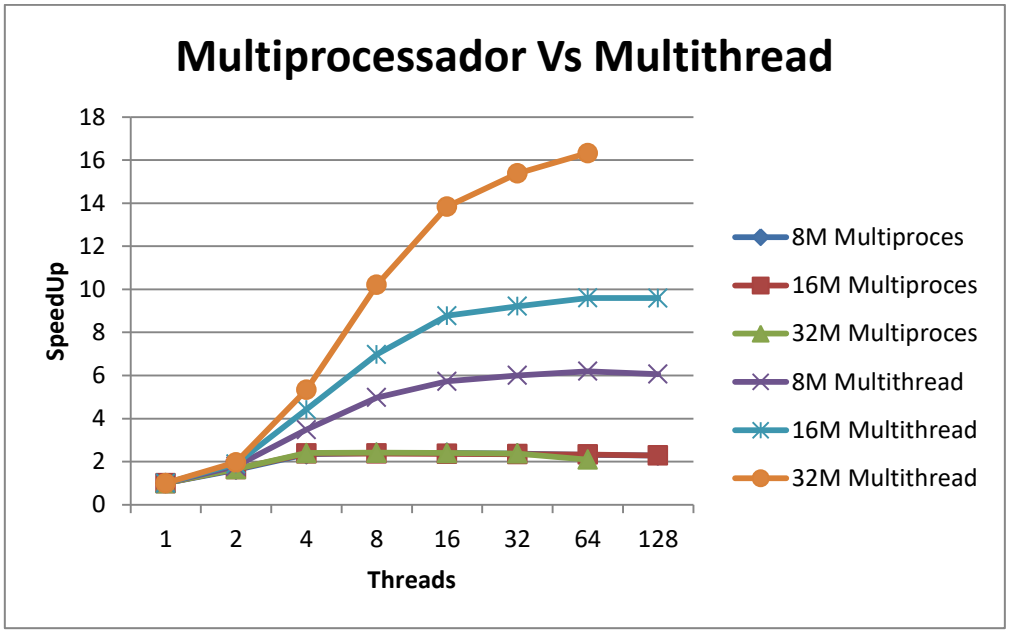


Com podem observar en la següent gràfica, en tots dos processadors, a mesura que augmentem el nombre de fils amb el que executem el codi disminueix el cost temporal del mateix fins arribar a un punt on s'estabilitza.

En tots els casos, encara que en multiprocessador no sigui observable, en quan partim el codi en molts de fils hi ha una petita penalització (l'execució tarda més que si el llancéssim amb menys fils). Això és degut a que la penalització de temps invertida en llançar els fils és més costosa que el benefici que proporciona la divisió del problema en aquests fils.



En aquesta gràfica podem observar la millora del rendiment obtinguda en executar el codi en diferents *threads*. Com ja esperàvem en un inici, la millora al executar el codi en diferents fils és molt més notòria en el processador multi fil que en el multiprocés.



Així que com a conclusió, podem dir que a pesar de que el codi s'executi més ràpid en el processador multiprocés que en el *multithread,* la millora en el rendiment obtinguda en executar-lo en diferents fils és molt més gran en el *multithread* que en el multiprocés.