**Guião 6**

Resumo introdutório

Número de *threads* usadas pelo *openMP*:

* Por omissão, = ao número de cores virtuais;
* Controlada pela flag: *--cpus-per-tasks* na linha de compilação;
* Controlada pela linha de comandos da bash: *export OMP\_NUM\_THREADS=N*
* Codificado na aplicação;

Compilar para OpenMP:

* Carregar módulo gcc que suporta openMP: *module load gcc/7.2.0*
* Compilar programa com suporte openMP: *gcc -O2 -fopenmp -lm sample.c*
* Executar programa no cluster com especificação de número de threads:
  + *srun --partition=cpar --cpus-per-task=2 ./a.out*
* Executar o comando para obter estatísticas de execução:
  + *srun --partition=cpar --cpus-per-task=2 perf stat ./a.out*

**Exercício 1:**

Variáveis em regiões paralelas:

* declaradas dentro de cada *thread*, são locais;
* declaradas fora, são partilhadas
  1. private (varlist): cada variável torna-se privada para *thread*. Valor inicial não especificado;
  2. firstprivate (varlist): igual ao *private*, mas as variáveis são inicializadas com o valor de fora da região paralela;
  3. lastprivate (varlist): igual ao *private*, mas o valor é o da última iteração do ciclo;
  4. reduction (op:var): igual ao *lastprivate,* mas o valor final é o resultado da redução dos valores privados calculados pela aplicação do op.

1. private, as variáveis são locais às threads, logo o w é inicializado dentro de cada uma das threads com o valor 0;

firstprivate, w é inicializado dentro da região paralela com valor de fora da região paralela, 10;

lastprivate, como w é inicializado dentro das threads começa como valor 0, o no fim, o lastprivate fica com o valor da última iteração executada, logo termina com 49.

redution, o valor inicial dentro de cada uma das threads é 10, mas no final é 110.

1. Sim, varia. Quando se passa da diretiva private para a diretiva firstprivate, lastprivate, ou redution, pois o valor de w é inicializado com 0 no caso private e em todos os outros casos é inicializado com o valor tomado antes da região paralela.
2. No caso do uso da private W= 10 e firstprivate W = 10 no caso de lastprivate w = 50 e reduction w = 110. Agora, no uso sem as primitivas poderia poderíamos perder valores pois as duas treads poderiam ler quase em simultâneo (e.g. w=10) e depois escrever ambas escreverem w=11 perdendo-se assim uma iteração (perguntar ao prof melhor isto).

**Exercício 2:**

1. Sim, varia. Devido aos problemas de dependência de dados partilhados e às leituras antes das escritas.
2. Sim, quantos mais threads, mais concorrência e consequentemente mais problemas de dependência de dados. Na verdade, isto só acontece quando as threads são executadas em processadores diferentes (e se cria paralelismo), pois se forem executadas no mesmo, em cada instante há apenas uma em execução e como tal não existe concorrência.
3. Ver melhor a aula anterior:

#pragma omp parallel

#pragma omp for

for(int i=0;i<size; i++) {

#pragma omp critical

{

dot += a[i]\*b[i];

}

}

1. Com o reduction (justificar melhor depois de fazer a alínea anterior);

**Exercício 3:**

Versão sequencial (executado em apenas uma thread):

Resultados:

pi = 3.1415925980

Performance counter stats for './a.out':

4,934.79 msec task-clock # 1.000 CPUs utilized

4 context-switches # 0.001 K/sec

0 cpu-migrations # 0.000 K/sec

175 page-faults # 0.035 K/sec

14,030,707,334 cycles # 2.843 GHz

10,018,434,576 stalled-cycles-frontend # 71.40% frontend cycles idle

12,022,856,089 instructions # 0.86 insn per cycle

# 0.83 stalled cycles per insn

1,004,330,267 branches # 203.520 M/sec

114,597 branch-misses # 0.01% of all branches

4.935614057 seconds time elapsed

4.934186000 seconds user

0.001000000 seconds sys

Extrato do Código paralelizado:

#pragma omp parallel for reduction (+ : mypi)

for(int i=0; i<n; i++) {

#pragma omp critical

{

mypi = mypi + f(i\*h);

}

}

Execução da versão paralela com recurso a 2 threads:

pi = 3.1415925980

Performance counter stats for './a.out':

54,509.39 msec task-clock # 1.998 CPUs utilized

5 context-switches # 0.000 K/sec

1 cpu-migrations # 0.000 K/sec

273 page-faults # 0.005 K/sec

157,794,124,493 cycles # 2.895 GHz

115,085,234,499 stalled-cycles-frontend # 72.93% frontend cycles idle

49,529,836,319 instructions # 0.31 insn per cycle

# 2.32 stalled cycles per insn

15,199,428,570 branches # 278.841 M/sec

779,188,357 branch-misses # 5.13% of all branches

27.280472845 seconds time elapsed

54.507937000 seconds user

0.002000000 seconds sys

Versão com 16 threads

pi = 3.1415925980

Performance counter stats for './a.out':

3,147,120.83 msec task-clock # 14.875 CPUs utilized

903 context-switches # 0.000 K/sec

27 cpu-migrations # 0.000 K/sec

679 page-faults # 0.000 K/sec

9,125,969,340,643 cycles # 2.900 GHz

8,861,224,266,388 stalled-cycles-frontend # 97.10% frontend cycles idle

309,420,562,008 instructions # 0.03 insn per cycle

# 28.64 stalled cycles per insn

91,544,221,934 branches # 29.088 M/sec

4,336,009,402 branch-misses # 4.74% of all branches

211.565009479 seconds time elapsed

3147.125909000 seconds user

0.000000000 seconds sys