OMP, for loop scheduling performance

Matteo Lugli, Carlo Uguzzoni

Introduzione

Sono stati studiati i tempi di esecuzione di un semplice *for loop* parallelizzato su più thread, sia dinamicamente che staticamente. Il codice completo si può trovare alla fine del documento. Seguono ora alcuni frammenti di codice commentati.

```
int N = 1; //size of chunk
int M = 100; //iterations
```

La variabile **N** denota la grandezza dei *chunk*, ossia il numero di iterazioni del loop che vengono assegnate per volta ad ogni thread che si libera. La variabile **M** invece rappresenta il numero di iterazioni del loop.

```
#pragma omp parallel for schedule(dynamic, N) num_threads(4)
/* #pragma omp parallel for schedule(static) */
for (int i = 0; i < M; ++i) {
  int randomnumber = (rand() % 6) + 1;
  usleep(mult*(randomnumber * 100));
}
```

La direttiva ***pragma*** definisce la modalità di schedulazione del loop, e nel caso dello scheduling dinamico viene passato come argomento la grandezza del *chunk*. Nel body del loop è stata usata un'istruzione di ***sleep*** per avere più controllo sulla durata delle singole iterazioni. In particolare, si è preferito non usare operazioni sulla memoria per evitare di aggiungere "rumore" dato dal comportamento a volte imprevedibile della cache (o comunque non argomento di questi esperimenti).

Tuttavia nel body del loop viene scelto casualmente un numero variabile $[0.1 - 0.6]$ di millisecondi in cui il thread viene fermato tramite l'istruzione di sleep. In questo modo si da' variabilità alla grandezza delle singole iterazioni: se si aspettasse lo stesso numero di millisecondi ad ogni iterazione, si osserverebbe sempre la miglior performance usando lo scheduling statico.

Esperimenti

Nei due esperimenti è stato fatto variare **M**, per mostrare in modo più chiaro la performance in relazione al numero di iterazioni. L'asse x denota semplicemente il numero della chiamata alla funzione ***foo()***. Per rendere consistenti i dati, *foo()* è stata chiamata 15 volte per ogni esperimento. Alla fine di ogni chiamata, è stato misurato il tempo di esecuzione complessivo, che viene mostrato sull'asse y.

Come si può notare all'aumentare del numero di iterazioni lo scheduling dinamico *fine-grain* diventa più performante, distaccandosi in modo più consistente dagli altri due tipi di scheduling. E' interessante notare come in Figura 1 lo scheduling statico sia mediamente migliore del dinamico: avendo poche iterazioni l'overhead dato dal costo di schedulazione è impattante.

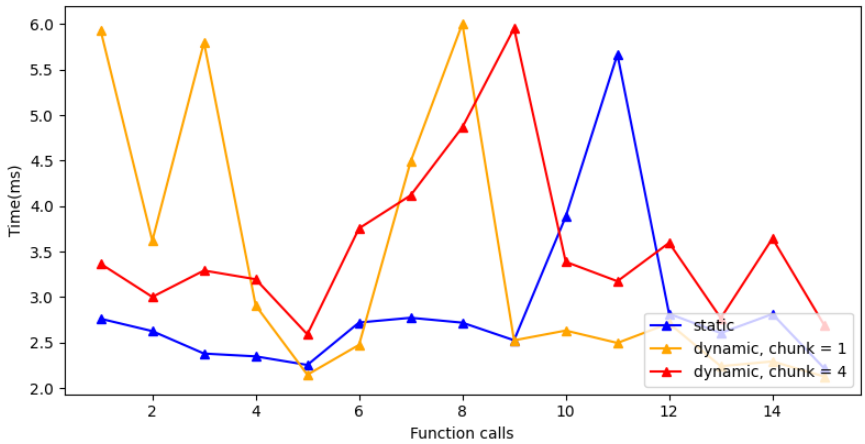

Figure 1: M (iterations) = 20;

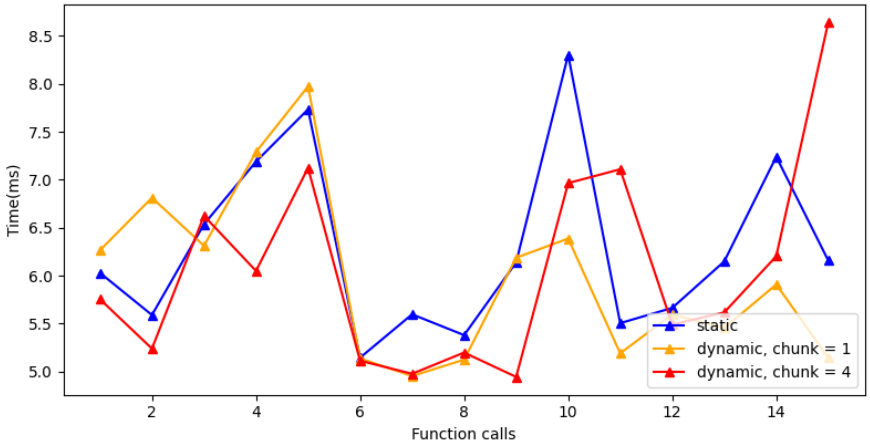

Figure 2: M (iterations) = 50;

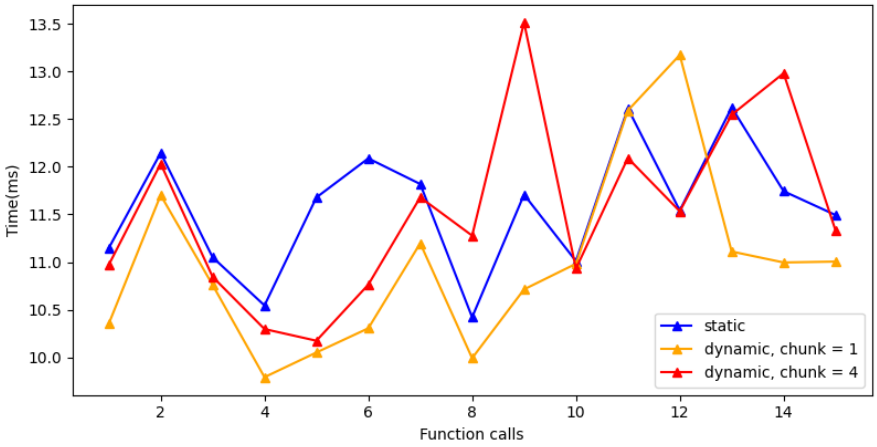

Figure 3: M (iterations) = 100;

Codice

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
void foo(FILE* fp) {
int N = 4; // chunk size
int M = 100;
struct timespec start, finish;
clock_gettime(CLOCK_REALTIME, &start);
#pragma omp parallel for schedule(dynamic, N) num_threads(4)
/* #pragma omp parallel for schedule(static) */
for (int i = 0; i < M; ++i) {
  int randomnumber = (rand() % 6) + 1;
  usleep(randomnumber * 100);
}
clock_gettime(CLOCK_REALTIME, &finish);
double elapsed_time = ((double) (finish.tv_nsec - start.tv_nsec))/((double) 1000000);
fprintf(fp, "%f\n",elapsed_time);
printf("Exec time: %f \n", elapsed_time);
return;
}
int main(){
  int num_ex = 15;
  FILE *fp = fopen("for_output_3.txt", "w");
  for(int i = 0; i < num_ex; i++){
foo(fp);
  }
  return 0;}
```