Introduction to Parallel Computing

Shared-memory programming with OpenMP

Table of Contents

Scheduling loops

- The $parallel\ for$ directive automatically applies **block portioning**: if the serial loop has n iterations, then thread 0 handles the first $n/thread\_count$ iterations, thread 1 handles the second $n/thread\_count$ iterations, and so on.
- This could be **less optimal**.
- For example, say we want to parallelize the serial loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
    sum += f(i);
```

  Where the time required to compute f is **proportional** to the size of the argument i.
  - Then, block portioning will assign much more work to the last thread, $thread\_count - 1$, than it will assign to thread 0
- It's better to use **cyclic partitioning** for better **work-balance** between threads: the iterations are assigned, one at a time, in a "**round-robin**" fashion to the threads.
- Suppose $t = thread\_count$. Then a **cyclic partitioning** will assign the iterations as follows

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

- Example:

```
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}  /* f */
```

  - Run the next three programs and compare the performance.

```
gcc-14 -o prog -fopenmp <filename.c>; ./prog
```

- The serial program

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double Sum(long n);
double f(long i);

int main(int argc, char* argv[]) {
   double  global_result;
   double  start, finish;
   long n = 100000;

   start = omp_get_wtime();
   global_result = Sum(n);
   finish = omp_get_wtime();

   printf("Result = %.14e\n", global_result);
   printf("Elapsed time = %f seconds\n", finish-start);

   return 0;
}

double f(long i) {
   long j;
   long start = i*(i+1)/2;
   long finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}

double Sum(long n) {
   double approx = 0.0;
   long i;

   for (i = 0; i <= n; i++) {
     approx += f(i);
   }
   return approx;
}
```

- The parallel program

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double Sum(long n, int thread_count);
double f(long i);

int main(int argc, char* argv[]) {
    double  global_result;
    double  start, finish;

    int thread_count = 2;
    long n = 100000;

    start = omp_get_wtime();
    global_result = Sum(n, thread_count);
    finish = omp_get_wtime();

    printf("Result = %.14e\n", global_result);
    printf("Elapsed time = %f seconds\n", finish-start);

    return 0;
}

double f(long i) {
    long j;
    long start = i*(i+1)/2;
    long finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

double Sum(long n, int thread_count) {
    double approx = 0.0;
    long i;

#  pragma omp parallel for num_threads(thread_count) \
reduction(+: approx)
    for (i = 0; i <= n; i++) {
        approx += f(i);
    }
    return approx;
}
```

- The **scheduled** parallel program

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

double Sum(long n, int thread_count);
double f(long i);

int main(int argc, char *argv[]) {
    double global_result;
    double start, finish;

    int thread_count = 2;
    long n = 100000;

    start = omp_get_wtime();
    global_result = Sum(n, thread_count);
    finish = omp_get_wtime();

    printf("Result = %.14e\n", global_result);
    printf("Elapsed time = %f seconds\n", finish - start);

    return 0;
}

double f(long i) {
    long j;
    long start = i * (i + 1) / 2;
    long finish = start + i;
    double return_val = 0.0;

    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}

double Sum(long n, int thread_count) {
    double approx = 0.0;
    long i;

#  pragma omp parallel for num_threads(thread_count) \
reduction(+: approx) schedule(dynamic)
    for (i = 0; i <= n; i++) {
        approx += f(i);
    }
    return approx;
}
```

- **Scheduling** in OpenMP: assigning iterations to threads.
- The ***schedule* clause** can be used to assign iterations to threads:

$$schedule(< type > [, chunk\ size])$$

- **Chunk**: a block of iterations that would be executed consecutively in the serial loop.
    - *chunk size* determines the **number of iterations in a block**. It's **optional**.

- Types of scheduling:

static

- The iterations are assigned to the threads before the loop is executed

dynamic/guided

- The iterations are assigned to the threads while the loop is executing
- After a thread completes its current set of iterations, it requests more from the run-time system

auto

- The compiler and/or the run-time system determine the schedule
- *Does not specify chunk size*

runtime

- The schedule is determined at run-time based on an environment variable

The $static$ schedule type

- The system assigns **chunks** of $\boldsymbol{chunksize}$ **iterations** to each thread in a round-robin fashion.
- Example: suppose we have 12 iterations, 0, 1,...,11, and three threads.
    - For $schedule(static, 1)$, the iterations will be assigned as

Thread 0 :	0, 3, 6, 9
Thread 1 :	1, 4, 7, 10
Thread 2 :	2, 5, 8, 11

    - For $schedule(static, 2)$, the iterations will be assigned as

Thread 0 :	0, 1, 6, 7
Thread 1 :	2, 3, 8, 9
Thread 2 :	4, 5, 10, 11

    - For $schdule(static, 4)$, the iterations will be assigned as

Thread 0 :	0, 1, 2, 3
Thread 1 :	4, 5, 6, 7
Thread 2 :	8, 9, 10, 11

    - The **default** schedule is defined by a particular implementation of OpenMP, but in most cases it is equivalent to the clause:

$$schedule(\ \boldsymbol{static}\ , total\_iterations\ /\ thread\_count)$$

- The $static$ schedule is a good choice when each loop iteration takes roughly the **same amount of time** to compute.

The $dynamic$ and $guided$ schedule type

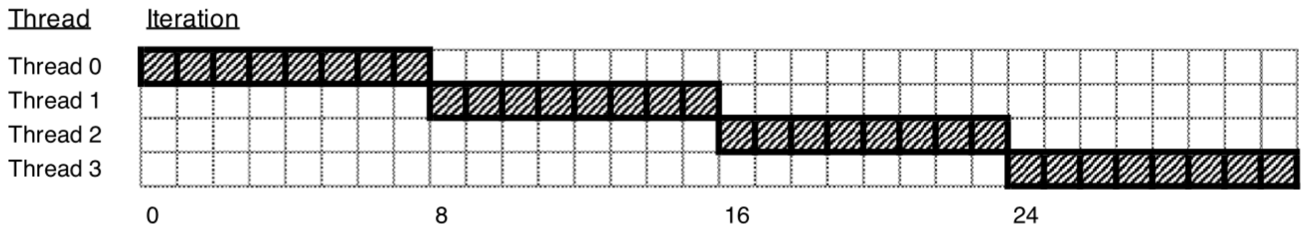
- $dynamic$ schedule: the iterations are also **broken up into chunks** of $chunksize$ consecutive iterations.
    - Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- The **default $chunksize$** is 1.
- The **difference** between $static$ and $dynamic$ schedules:
    - $dynamic$ schedule assigns ranges to threads on a **first-come, first-served** basis.
        - Good choice if loop iterations **do not take a uniform** amount of time to compute.
        - There is some **overhead** associated with assigning them dynamically at run-time.
    - $static$ schedule assigns a range of threads **fixedly** at the start of the loop
        - Used when loop iterations **take a uniform** amount of time.
- With **larger** chunk sizes, **fewer** dynamic assignments will be made.
- In a ***guided* schedule**, as chunks are completed, the size of the new chunks **decreases**.
- If no $chunksize$ is specified, the size of the chunks decreases down to **1**. If $chunksize$ is specified, it decreases down to $\boldsymbol{chunksize}$.

The $runtime$ schedule type

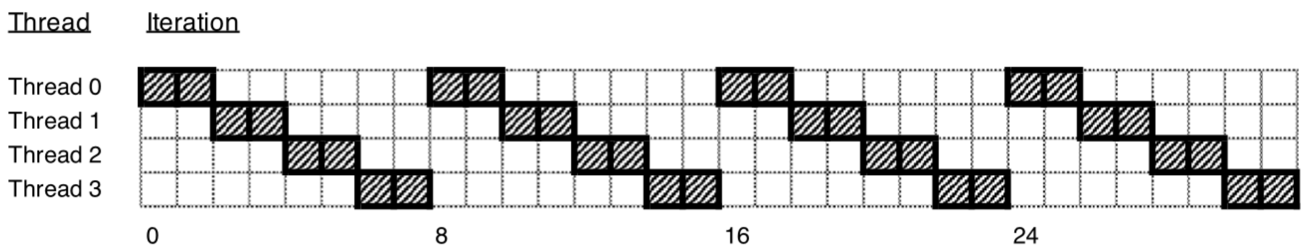
- When $schedule(runtime)$ is specified, the system uses the **environment variable `OMP_SCHEDULE`** to determine at run-time how to schedule the loop.
    - Environment variables are **named values** that can be accessed by a running program.
    - Examples: $PATH, HOME, SHELL$
- The **`OMP_SCHEDULE`** environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.
    - `$ export OMP_SCHEDULE="static ,1"`

- Scheduling **visualization** for the **static**, **dynamic**, and **guided** schedule types with **4 threads and 32 iterations**.

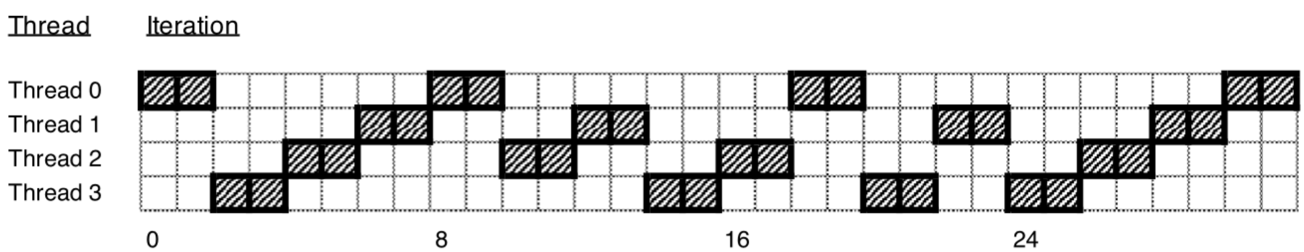
`schedule(static)`



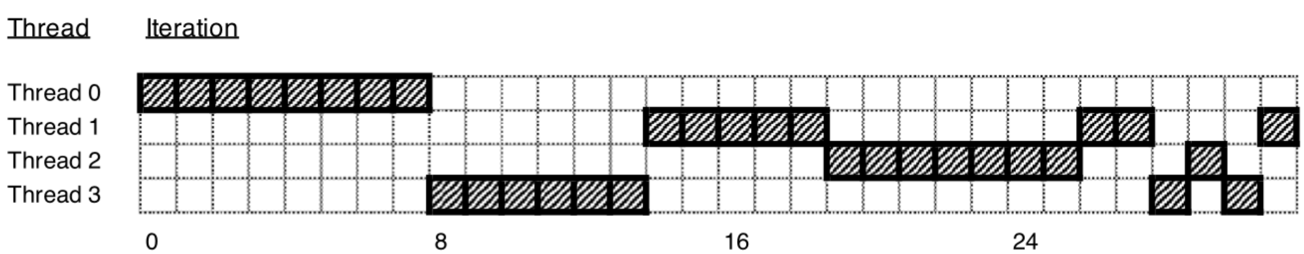
`schedule(static, 2)`



`schedule(dynamic, 2)`



`schedule(guided)`



Which schedule type to choose?

- It depends on the problem and the nature of the loop.
- There is some **overhead** associated with the use of a **schedule clause**.
  - $static$ overhead **less than** $dynamic$ overhead **less than** $guided$ overhead
- Practically, you should **try different options** until you hit a satisfactory performance.
  - Make use of the $schedule(runtime)$ to test different settings.