

Intermediate OpenMP

Pangfeng Liu
National Taiwan University

April 14, 2022

Schedule Clause

- A schedule clause specifies a scheduling policy for distributing loop iterations to threads for a `parallel` for pragma.

```
1 schedule (policy, chunk)
```

Schedule Clause

policy The scheduling policy for distributing iterations to threads.

chunk The basic unit number of iterations for scheduling.

Scheduling Policy

OpenMP has the following scheduling policies for a `parallel for` pragma.

- static** A static round-robin policy.
- dynamic** When a thread finishes it asks for more, and the chunk size is fixed.
- guided** When a thread finishes it asks for more, and the chunk size decreases exponentially, and will not be smaller than the chunk size.
- runtime** The policy is determined by an environment variable `OMP_SCHEDULE` or calling a function `omp_set_policy`.
- auto** This policy selects a scheduling automatically.

Chunk Size

- The chunk corresponds to the *granularity* in the introductory part of this course.
- If the granularity is large, the scheduling overhead becomes smaller, but it is more likely to have uneven workload distribution.

Dicsussion

- Find out the default policy and chunk size based on the observation fromt the previous “Basic OpenMP Programming” lecture.

Static Policy

- The static policy distributes the iterations in a round-robin fashion.
- This is the default policy.
- A thread is given a chunk size of iterations, then the next threads is given the same amount of iterations.
- Repeat this process until all iterations are distributed.

Static Policy Example

- In the following example we distribute n iterations among t threads under `schedule(static, 4)` policy.
- The n and t are given as command line arguments.
- We put a `sleep(i)` to simulate the amount of work, where i is the loop index.
- We use a variable `elapsedTime` to keep track (roughly) the amount of time each thread runs.

Example 1: (for-static-chunk-4.c) Set chunk size to 4

```
2  #include <omp.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <unistd.h>                /* for sleep */
```

```
8  int main(int argc, char *argv[])
9  {
10     assert(argc == 3);
11     omp_set_num_threads(atoi(argv[1]));
12     int n = atoi(argv[2]);
13     printf("# of proc = %d\n", omp_get_num_procs());
14     printf("# of loop iterations = %d\n", n);
15     int elapsedTime = 0;
16     #pragma omp parallel for firstprivate(elapsedTime) \
17     schedule(static, 4)
18     for (int i = 0; i < n; i++) {
19         sleep(i);
20         elapsedTime += i;
21         printf("thread %d i %d elapsed time %d.\n",
22               omp_get_thread_num(), i, elapsedTime);
23     }
24     return 0;
25 }
```

Demonstration

- Run the `for-static-chunk-4-omp` program with 16 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work evenly distributed? Observe the `elapsedTime` and give your conclusion.
- How to improve the workload distribution?

Static Policy Example

- In the previous example, the threads with higher indices do more work, and the workload is unbalanced among threads.
- One simple way to reduce work imbalance is to reduce the chunk size and evenly distribute the heavy iterations among threads.
- We try chunk size one and observe the results.

Example 2: (for-static-chunk-1.c) Set chunk size to 1

```
15 #pragma omp parallel for firstprivate(elapsedTime) \  
16 schedule(static, 1)  
17     for (int i = 0; i < n; i++) {  
18         sleep(i);  
19         elapsedTime += i;  
20         printf("thread %d i %d elapsed time %d.\n",  
21             omp_get_thread_num(), i, elapsedTime);  
22     }
```

Demonstration

- Run the `for-static-chunk-1-omp` program with 16 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work more evenly distributed than the previous example? Observe the `elapsedTime` and give your conclusion.
- How to further improve the workload distribution?

Workload Imbalance

- In the previous example, the threads with higher indices do more work, and the workload is imbalanced among threads.
- Even though we reduce the chunk size to 1, the last thread still has a much heavier workload than the first one.

Dynamic Policy

- The dynamic policy dictates that when a thread finishes its work it asks for more, and the chunk size is fixed.
- The work is in a *work pool* – the work is put into a pool and any idle worker can take work from it.

Example 3: (for-dynamic-chunk-1.c) Set scheduling to dynamic

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(dynamic, 1)  
18     for (int i = 0; i < n; i++) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```

Demonstration

- Run the `for-dynamic-chunk-1-omp` program with 16 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work more evenly distributed than the previous example? Observe the `elapsedTime` and give your conclusion.
- How to further improve the workload distribution?

Workload Imbalance

- In the previous example, we run the iterations in ascending workload order; that is, we run the heaviest workload *last*.
- That means the last work may cause all threads to wait for the last thread to finish.

Descending Workload Order

- In order to avoid this problem we will run the iterations in *descending* workload order, so that the dynamic policy might have a better change to balance the workload.
- We can do this because this is a *parallel for* – the semantic explicitly demands that all iterations are independent.

Example 4: (for-dynamic-chunk-1-reverse.c) Descending workload order

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(dynamic, 1)  
18     for (int i = n - 1; i >= 0; i--) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```


Demonstration

- Run the `for-dynamic-chunk-1-reverse-omp` program with 16 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work more evenly distributed than the previous example? Observe the `elapsedTime` and give your conclusion.

Guided Policy

- The *guided policy* dictates that when a thread finishes its work it asks for more, and the chunk size *decreases* exponentially, and will not be smaller than the chunk size.

Rationals

- Initial large chunk size can reduce the overheads in scheduling.
- Later, a smaller chunk size can reduce the workload imbalance.
- In a sense, this is similar to the descending workload order – the larger tasks run first because of the large chunk size, and the smaller tasks run later.

Comparison

- We compare two policies – (dynamic, 1) and (guided, 1) on ascending workload order.
- The idea is that by combining the small workload at the beginning, we will have a better distribution.

Example 5: (for-dynamic-chunk-1.c) Dynamic policy for ascending workload order

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(dynamic, 1)  
18     for (int i = 0; i < n; i++) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```

Example 6: (for-guided-chunk-1.c) Guided policy for ascending workload order

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(guided, 1)  
18     for (int i = 0; i < n; i++) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```

Demonstration

- Run the `for-dynamic-chunk-1-omp` program with 32 iterations on 4 threads.
- Run the `for-guided-chunk-1-omp` program with 32 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work more evenly distributed than the previous example? Observe the `elapsedTime` and give your conclusion.

Worsen the Case

- To make the matter worse, we decide to run the guided policy on a descending workload with chunk size 1.
- We compare two policies – (dynamic, 1) and (guided, 1) on ascending workload order.

Example 7: (for-dynamic-chunk-1-reverse.c) Dynamic policy for descending workload order

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(dynamic, 1)  
18     for (int i = n - 1; i >= 0; i--) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```

Example 8: (for-guided-chunk-4-reverse.c) Guided policy for descending workload order

```
16 #pragma omp parallel for firstprivate(elapsedTime) \  
17 schedule(guided, 4)  
18     for (int i = n - 1; i >= 0; i--) {  
19         sleep(i);  
20         elapsedTime += i;  
21         printf("thread %d i %d elapsed time %d.\n",  
22             omp_get_thread_num(), i, elapsedTime);  
23     }
```

Demonstration

- Run the `for-dynamic-chunk-1-reverse-omp` program with 32 iterations on 4 threads.
- Run the `for-guided-chunk-1-reverse-omp` program with 32 iterations on 4 threads.

Discussion

- Describe how the iterations are distributed among the threads.
- Is the work more evenly distributed than the previous example? Observe the `elapsedTime` and give your conclusion.

Runtime Policy

- Call the function `omp_set_schedule` to determine scheduling policy.
- We try static, dynamic, guided and auto.

Flexibility

- We will read the policy as an integer as the fourth command line argument.
- We will read the chunk size as an integer as the fifth command line argument.

Prototype 9:

```
1 void omp_set_schedule(omp_sched_t kind, int modifier);
```

`kind` for the scheduling policy.

`modifier` for the chunk size.

- `omp_sched_static` (integer 1) for static
- `omp_sched_dynamic` (integer 2) for dynamic
- `omp_sched_guided` (integer 3) for guided
- `omp_sched_auto` (integer 4) for auto

Example 10: (for-runtime.c) Runtime policy for ascending workload order

```
8  int main(int argc, char *argv[])
9  {
10     assert(argc == 5);
11     omp_set_num_threads(atoi(argv[1]));
12     int n = atoi(argv[2]);
13     printf("# of proc = %d\n", omp_get_num_procs());
14     printf("# of loop iterations = %d\n", n);
15
16     int policy = atoi(argv[3]);
17     int chunk = atoi(argv[4]);
18     omp_set_schedule(policy, chunk);
```

```
20     int elapsedTime = 0;
21 #pragma omp parallel for firstprivate(elapsedTime) \
22 schedule(runtime)
23     for (int i = 0; i < n; i++) {
24         sleep(i);
25         elapsedTime += i;
26         printf("thread %d i %d elapsed time %d.\n",
27             omp_get_thread_num(), i, elapsedTime);
28     }
29     return 0;
30 }
```

Demonstration

- Run the `for-runtime-omp` program with 16 iterations on 4 threads, under various combinations of policy and chunk sizes.

Discussion

- Describe how the iterations are distributed among the threads when the policy is auto.
- Which policy distributes workload more evenly than others? Observe the `elapsedTime` and give your conclusion.

```
1 double omp_get_wtime();
```

- Return the current time.
- Can measure the length of a window by two consecutive calls.

Parallel Sections

- Want to run a set of computations in parallel, e.g., initializing two matrices a and b.
- Can have a thread to initialize a and another thread for b.
- Use parallel sections pragma.

Parallel Sections

```
1 #pragma omp parallel sections
2 {
3 #pragma omp section
4     /* section 1 */
5 #pragma omp section
6     /* section 2 */
7 }
```

Naive Parallelization

- A parallel program initializes two matrices a and b using `#pragma omp parallel` sections.
- The sizes of both matrices are 8192 by 8192.
- We use `omp_get_wtime` to measure time.
- Finally we check for correctness of the program by assertions.

Example 11: (2loops.c)

```
2  #include <stdio.h>
3  #include <assert.h>
4  #include "omp.h"
5
6  #define N 8192
7  int a[N][N], b[N][N];
8
9  int main()
10 {
11     int i, j;
12     double t;
13     printf("number of processor = %d\n",
14           omp_get_num_procs());
15     t = omp_get_wtime();
```

```
17 #pragma omp parallel sections
18 {
19 #pragma omp section
20 {
21     printf("thread %d for a\n",
22           omp_get_thread_num());
23     for (i = 0; i < N; i++)
24         for (j = 0; j < N; j++)
25             a[i][j] = i + j;
26 } /* section */
27 #pragma omp section
28 {
29     printf("thread %d for b\n",
30           omp_get_thread_num());
31     for (i = 0; i < N; i++)
32         for (j = 0; j < N; j++)
33             b[i][j] = i - j;
34 } /* section */
35 } /* parallel sections */
```

```
37  t = omp_get_wtime() - t;  
38  printf("time is %lf\n", t);  
39  {  
40      for (i = 0; i < N; i++)  
41          for (j = 0; j < N; j++)  
42              assert(a[i][j] == i + j);  
43  
44      for (i = 0; i < N; i++)  
45          for (j = 0; j < N; j++)  
46              assert(b[i][j] == i - j);  
47  }  
48  return 0;  
49 }
```

Demonstration

- Run the 2loops-omp program.

Discussion

- Does this parallel program correctly initialize the matrices?
What could be the reason if it does not?
- Can you conclude which thread will run which section?

Index Variables

- The problem with the previous program is that two threads running two sections share the index variables `i` and `j`.
- There are various ways to fix the problem, and we will use the most intuitive one – by declaring the index variables within the initialization part of for loop, as c99-style syntax.

Example 12: (2loops-stdc99.c)

```
2  #include <stdio.h>
3  #include <assert.h>
4  #include "omp.h"
5
6  #define N 8192
7  int a[N][N], b[N][N];
8
9  int main()
10 {
11     double t;
12     printf("number of processor = %d\n",
13           omp_get_num_procs());
14
15     t = omp_get_wtime();
```

```
17 #pragma omp parallel sections
18 {
19 #pragma omp section
20 {
21     printf("thread %d for a\n", omp_get_thread_num());
22     for (int i = 0; i < N; i++)
23         for (int j = 0; j < N; j++)
24             a[i][j] = i + j;
25 } /* section */
26 #pragma omp section
27 {
28     printf("thread %d for b\n", omp_get_thread_num());
29     for (int i = 0; i < N; i++)
30         for (int j = 0; j < N; j++)
31             b[i][j] = i - j;
32 } /* section */
33 } /* parallel sections */
34 t = omp_get_wtime() - t;
```

```
36 printf("time is %lf\n", t);  
37 {  
38     for (int i = 0; i < N; i++)  
39         for (int j = 0; j < N; j++)  
40             assert(a[i][j] == i + j);  
41  
42     for (int i = 0; i < N; i++)  
43         for (int j = 0; j < N; j++)  
44             assert(b[i][j] == i - j);  
45 }  
46 return 0;  
47 }
```

Demonstration

- Run the `2loops-stdc99-omp` program.

Discussion

- Does this parallel program correctly initialize the matrices?
How do you know?
- Can you conclude which thread will run which section?

parallel Revisited

- We have discussed `parallel sections` and `parallel for` directives, but in two separate contexts.
- Now we want to combine the idea and describe the true meaning of `parallel`, which is to “run the following statement with multiple threads”.
- If then we encounter a `sections`, then we have a `parallel sections`.
- If then we encounter a `for`, then we have a `parallel for`.

parallel

```
1 #pragma omp parallel
```

- Run the following statement with multiple threads.

parallel Examples

Example 13: (parallel-sections.c) An Example

```
1  #pragma omp parallel sections
2  {
3  #pragma section
4      code 1;
5  #pragma section
6      code 2;
7  }
8  /* The following is equivalent */
9  #pragma omp parallel
10 {
11 #pragma omp sections
12 {
13 #pragma omp section
14     code 1;
15 #pragma omp section
16     code 2;
17 }
18 }
```


parallel Examples

Example 14: (parallel-sections-wrong.c) A Wrong Example

```
1  #pragma omp parallel sections
2  {
3  #pragma section
4      code 1;
5  #pragma section
6      code 2;
7  }
8  /* the following will not compile */
9  #pragma omp parallel
10 #pragma omp sections
11 #pragma omp section
12     code 1;
13 #pragma omp section
14     code 2;
```

Compile Error

- The previous code will not compile because a `section` must be within a `sections`.
- Unfortunately the effect of an `sections` is only for the following statement, so we need to use compound statement to include *all section* statements.

parallel Examples

Example 15: (parallel-1for.c) An Examples

```
1 #pragma omp parallel
2 #pragma omp for
3   for (...)
4     code;
5 /* The following is equivalent */
6 #pragma omp parallel for
7   for (...)
8     code;
```

Multiply and Divide

- These two are equivalent.
- One can think of the `parallel` is to spawn multiple threads, and the `for` is to divide work.

parallel Examples

Example 16: (parallel-2for.c) An Examples

```
1 #pragma omp parallel for
2   for (...)
3 #pragma omp paralell for
4   for (...)
5
6 #pragma omp parallel
7   {
8 #pragma omp for
9   for (...)
10 #pragma omp for
11   for (...)
12 }
```

Multiply and Divide

- These two are also equivalent.
- One can think of the `parallel` is to spawn multiple threads, and the first `for` is to divide the work, and the second `for` is to divide the work *again*.

parallel Examples

Example 17: (parallel-2for-wrong.c) An Wrong Examples

```
1 #pragma omp parallel for
2   for (...)
3 #pragma omp paralell for
4   for (...)
5
6 #pragma omp parallel
7 #pragma omp for
8   for (...)
9 #pragma omp for
10  for (...)
```

Multiply and Divide

- These two are *not* equivalent.
- One can think of the `parallel` is to spawn multiple threads, and the first `for` is to divide the work.
- Since the `parallel` covers only one statement, now we switch back to single thread.
- The second `for` has no multiple threads to divide the work!
- Unlike the previous sections problem, this will *not* cause compile error and will simply degrade performance.

Both Section and For

- Want to use both sections and for in `2loops-std99.c`
- We already have a thread initializing `a` and another thread initializing `b`.
- Now we use `parallel for` to parallel the asserts at the end.

Example 18: (2loops-stdc99-both.c)

```
14 #pragma omp parallel
15 {
16 #pragma omp sections
17 {
18 #pragma omp section
19 {
20     printf("thread %d init a\n", omp_get_thread_num());
21     for (int i = 0; i < N; i++)
22         for (int j = 0; j < N; j++)
23             a[i][j] = i + j;
24 }
25 #pragma omp section
26 {
27     printf("thread %d init b\n", omp_get_thread_num());
28     for (int i = 0; i < N; i++)
29         for (int j = 0; j < N; j++)
30             b[i][j] = i - j;
31 }
32 }
33 }
```

```
35 #pragma omp parallel
36 {
37 #pragma omp for
38     for (int i = 0; i < N; i++) {
39         printf("thread %d check a\n",
40             omp_get_thread_num());
41         for (int j = 0; j < N; j++)
42             assert(a[i][j] == i + j);
43     }
44 #pragma omp for
45     for (int i = 0; i < N; i++) {
46         printf("thread %d check b\n",
47             omp_get_thread_num());
48         for (int j = 0; j < N; j++)
49             assert(b[i][j] == i - j);
50     }
51 } /* parallel */
```

Structure

- parallel
 - sections
 - section
 - section
- parallel
 - for
 - for

Meaning

- `parallel` Go multithreading
 - `sections` Divide the work
 - `section`
 - `section`
- `parallel` Go multithreading
 - `for` Divide the work
 - `for` Divide the work

Demonstration

- Run the 2loops-stdc99-both-omp program.

Discussion

- Observe the thread indices reported by `2loops-stdc99-both-omp` and confirm that multi-threading is active.
- Can you conclude which thread will run which section?
- Can you conclude which thread will run which loop?

One Statement after `parallel`

- What will happen if we do *not* place everything as a compound statement after `parallel`?

Example 19: (2loops-stdc99-both-wrong.c)

```
14 #pragma omp parallel
15 #pragma omp sections
16 {
17 #pragma omp section
18 {
19     printf("thread %d init a\n",
20           omp_get_thread_num());
21     for (int i = 0; i < N; i++)
22         for (int j = 0; j < N; j++)
23             a[i][j] = i + j;
24 }
25 #pragma omp section
26 {
27     printf("thread %d init b\n",
28           omp_get_thread_num());
29     for (int i = 0; i < N; i++)
30         for (int j = 0; j < N; j++)
31             b[i][j] = i - j;
32 }
33 }
```

```
35 #pragma omp parallel
36 #pragma omp for
37     for (int i = 0; i < N; i++) {
38         printf("thread %d check a\n",
39             omp_get_thread_num());
40         for (int j = 0; j < N; j++)
41             assert(a[i][j] == i + j);
42     }
43 #pragma omp for
44     for (int i = 0; i < N; i++) {
45         printf("thread %d check b\n",
46             omp_get_thread_num());
47         for (int j = 0; j < N; j++)
48             assert(b[i][j] == i - j);
49     }
```

Meaning

- `parallel` Go multithreading
 - `sections` Divide the work
 - `section`
 - `section`
- `parallel` Go multithreading
 - `for` Divide the work
- `for` Divide the work???

Problem

- Removing the compound statement of the first `parallel` will not cause any problem because the sections must use a compound statement to include two section. Otherwise it will not compile.
- Removing the compound statement of the first `parallel` cause performance problem because the second `parallel` does not cover the second `for`, so it will *not* go multithreading.

Demonstration

- Run the `2loops-stdc99-both-wrong-omp` program.

Discussion

- Observe the thread indices reported by `2loops-stdc99-both-wrong-omp` and confirm that multi-threading is *not active* in the last loop.
- Can you conclude which thread will run which section?
- Can you conclude which thread will run which loop?

One parallel

- Now we put everything into a single parallel.

Example 20: (2loops-stdc99-combined.c)

```
14 #pragma omp parallel
15 {
16 #pragma omp sections
17 {
18 #pragma omp section
19 {
20     printf("thread %d init a\n", omp_get_thread_num());
21     for (int i = 0; i < N; i++)
22         for (int j = 0; j < N; j++)
23             a[i][j] = i + j;
24 }
25 #pragma omp section
26 {
27     printf("thread %d init b\n", omp_get_thread_num());
28     for (int i = 0; i < N; i++)
29         for (int j = 0; j < N; j++)
30             b[i][j] = i - j;
31 }
32 }
```



```
34 #pragma omp for
35     for (int i = 0; i < N; i++) {
36         printf("thread %d check a\n",
37             omp_get_thread_num());
38         for (int j = 0; j < N; j++)
39             assert(a[i][j] == i + j);
40     }
41 #pragma omp for
42     for (int i = 0; i < N; i++) {
43         printf("thread %d check b\n",
44             omp_get_thread_num());
45         for (int j = 0; j < N; j++)
46             assert(b[i][j] == i - j);
47     }
48 } /* parallel */
```

Structure

- `parallel` Go multithreading
 - `sections` Divide the work
 - `section`
 - `section`
 - `for` Divide the work
 - `for` Divide the work

Efficiency

- The program will go multi-threading only *once*.
- This removes the overheads in creating and destroying the threads.

Demonstration

- Run the `2loops-stdc99-combine-omp` program.

Discussion

- Observe the thread indices reported by `2loops-stdc99-combined-omp` and confirm that multi-threading is active everywhere.
- Can you conclude which thread will run which section?
- Can you conclude which thread will run which loop?