Intermediate OpenMP

Pangfeng Liu
National Taiwan University

April 15, 2016

Schedule Clause

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

```
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_schedule`.

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

round robin

- 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

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <unistd.h>                /* for sleep */
```

```c
int main(int argc, char *argv[])
{
  assert(argc == 3);
  omp_set_num_threads(atoi(argv[1]));
  int n = atoi(argv[2]);
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of loop iterations = %d\n", n);
  int elapsedTime = 0;
#pragma omp parallel for firstprivate(elapsedTime) \
schedule(static, 4)
  for (int i = 0; i < n; i++) {
    sleep(i);
    elapsedTime += i;
    printf("thread %d i %d elapsed time %d.\n",
           omp_get_thread_num(), i, elapsedTime);
  }
  return 0;
}
```

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 not balanced among threads.
- One simple way to reduce work imbalance is to reduce the chunk size, so that the heavy iterations are more evenly distributed.
- We try chunk size 1 and observe the results.

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

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

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 not balanced 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

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

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 even if the workload is very balanced all the way to the end, 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

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

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 smaller chunk size can reduce the workload imbalance.
- In a sense this is similar to the descending workload order – the larger work is given out first because of large chunk size, and smaller work is given out 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

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

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

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

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

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

Example 8: (for-guided-chunk-1-reverse.c) Guided policy for descending workload order

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

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:

```
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

```
int main(int argc, char *argv[])
{
  assert(argc == 5);
  omp_set_num_threads(atoi(argv[1]));
  int n = atoi(argv[2]);
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of loop iterations = %d\n", n);

  int policy = atoi(argv[3]);
  int chunk = atoi(argv[4]);
  omp_set_schedule(policy, chunk);
```

```
  int elapsedTime = 0;
#pragma omp parallel for firstprivate(elapsedTime) \
schedule(runtime)
  for (int i = 0; i < n; i++) {
    sleep(i);
    elapsedTime += i;
    printf("thread %d i %d elapsed time %d.\n",
           omp_get_thread_num(), i, elapsedTime);
  }
  return 0;
}
```

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.

```c
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

```
#pragma omp parallel sections
{
#pragma omp section
  /* section 1 */
#pragma omp section
  /* section 2 */
}
```

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)

```
#include <stdio.h>
#include <assert.h>
#include "omp.h"

#define N 8192
int a[N][N], b[N][N];

int main()
{
  int i, j;
  double t;
  printf("number of processor = %d\n",
         omp_get_num_procs());
  t = omp_get_wtime();
```

```c
#pragma omp parallel sections
  {
#pragma omp section
    {
      printf("thread %d for a\n",
             omp_get_thread_num());
      for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
          a[i][j] = i + j;
    } /* section */
#pragma omp section
    {
      printf("thread %d for b\n",
             omp_get_thread_num());
      for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
          b[i][j] = i - j;
    } /* section */
  } /* parallel sectsions */
```

```
  t = omp_get_wtime() - t;
  printf("time is %lf\n", t);
  {
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        assert(a[i][j] == i + j);

    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        assert(b[i][j] == i - j);
  }
  return 0;
}
```

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?

fail??

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)

```c
#include <stdio.h>
#include <assert.h>
#include "omp.h"

#define N 8192
int a[N][N], b[N][N];

int main()
{
  double t;
  printf("number of processor = %d\n",
         omp_get_num_procs());

  t = omp_get_wtime();
```

```c
#pragma omp parallel sections
  {
#pragma omp section
    {
      printf("thread %d for a\n", omp_get_thread_num());
      for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
          a[i][j] = i + j;
    } /* section */
#pragma omp section
    {
      printf("thread %d for b\n", omp_get_thread_num());
      for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
          b[i][j] = i - j;
    } /* section */
  } /* parallel sectsions */
  t = omp_get_wtime() - t;
```

```
  printf("time is %lf\n", t);
  {
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        assert(a[i][j] == i + j);

    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        assert(b[i][j] == i - j);
  }
  return 0;
}
```

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

```
#pragma omp parallel
```

- Run the following statement with multiple threads.

parallel Examples

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

```
#pragma omp parallel sections
{
#pragma section
  code 1;
#pragma section
  code 2;
}
/* The following is equivlent */
#pragma omp parallel
{
#pragma omp sections
  {
#pragma omp section
    code 1;
#pragma omp section
    code 2;
  }
}
```

parallel Examples

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

```
#pragma omp parallel sections
{
#pragma section
  code 1;
#pragma section
  code 2;
}
/* the following will not compile */
#pragma omp parallel
#pragma omp sections
#pragma omp section
  code 1;
#pragma omp section
  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

```
#pragma omp parallel
#pragma omp for
  for (...)
    code;
/* The following is equivlent */
#pragma omp parallel for
  for (...)
    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

```
#pragma omp parallel for
  for (...)
#pragma omp paralell for
  for (...)

#pragma omp parallel
  {
#pragma omp for
    for (...)
#pragma omp for
    for (...)
  }
```

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

```
#pragma omp parallel for
  for (...)
#pragma omp paralell for
  for (...)

#pragma omp parallel
#pragma omp for
    for (...)
#pragma omp for
    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)

```
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
        printf("thread %d init a\n", omp_get_thread_num());
        for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
            a[i][j] = i + j;
      }
#pragma omp section
      {
        printf("thread %d init b\n", omp_get_thread_num());
        for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
            b[i][j] = i - j;
      }
    }
  }
```

```c
#pragma omp parallel
  {
#pragma omp for
    for (int i = 0; i < N; i++) {
      printf("thread %d check a\n",
        omp_get_thread_num());
      for (int j = 0; j < N; j++)
        assert(a[i][j] == i + j);
    }
#pragma omp for
    for (int i = 0; i < N; i++) {
      printf("thread %d check b\n",
             omp_get_thread_num());
      for (int j = 0; j < N; j++)
        assert(b[i][j] == i - j);
    }
  } /* 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)

```
#pragma omp parallel
#pragma omp sections
  {
#pragma omp section
    {
      printf("thread %d init a\n",
             omp_get_thread_num());
      for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
          a[i][j] = i + j;
    }
#pragma omp section
    {
      printf("thread %d init b\n",
             omp_get_thread_num());
      for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
          b[i][j] = i - j;
    }
  }
```

```
#pragma omp parallel
#pragma omp for
  for (int i = 0; i < N; i++) {
    printf("thread %d check a\n",
           omp_get_thread_num());
    for (int j = 0; j < N; j++)
      assert(a[i][j] == i + j);
  }
#pragma omp for
  for (int i = 0; i < N; i++) {
    printf("thread %d check b\n",
           omp_get_thread_num());
    for (int j = 0; j < N; j++)
      assert(b[i][j] == i - j);
  }
```

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)

```c
#pragma omp parallel
  {
#pragma omp sections
    {
#pragma omp section
      {
        printf("thread %d init a\n", omp_get_thread_num());
        for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
            a[i][j] = i + j;
      }
#pragma omp section
      {
        printf("thread %d init b\n", omp_get_thread_num());
        for (int i = 0; i < N; i++)
          for (int j = 0; j < N; j++)
            b[i][j] = i - j;
      }
    }
```

```
#pragma omp for
    for (int i = 0; i < N; i++) {
      printf("thread %d check a\n",
        omp_get_thread_num());
      for (int j = 0; j < N; j++)
        assert(a[i][j] == i + j);
    }
#pragma omp for
    for (int i = 0; i < N; i++) {
      printf("thread %d check b\n",
             omp_get_thread_num());
      for (int j = 0; j < N; j++)
        assert(b[i][j] == i - j);
    }
  } /* 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?