OpenMP Scheduling

http://cs.umw.edu/~finlayson/class/fall16/cpsc425/notes/12-scheduling.html

Overview

We have seen that OpenMP will automatically partition the iterations of a for loop with the parallel for construct.

Now we will look at how we OpenMP schedules loop iterations, and how we can optimize the way loop iterations are divided.

Static Schedules

By default, OpenMP statically assigns loop iterations to threads. When the parallel for block is entered, it assigns each thread the set of loop iterations it is to execute.

The following program illustrates this is done by default:

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

#define THREADS 8
#define N 100

int main ( ) {
  int i;

  #pragma omp parallel for num_threads(THREADS)
  for (i = 0; i < N; i++) {
    printf("Thread %d is doing iteration %d.\n", omp_get_thread_num(
), i);
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

A static schedule can be non-optimal, however. This is the case when the different iterations take different amounts of time. This is true of the following program, in which each loop iteration sleeps for a number of seconds equal to the iteration number:

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 4
#define N 16

int main ( ) {
  int i;

  #pragma omp parallel for schedule(static) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);

    printf("Thread %d has completed iteration %d.\n",
omp_get_thread_num( ), i);
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

This program also specifies static scheduling, in the parallel for directive. This is the default on our systems, so is not needed.

How long does this program take?

How long could it take in the best case?

Dynamic Schedules

This program can be greatly improved with a *dynamic* schedule.
Here, OpenMP assigns one iteration to each thread. When the thread finishes, it will be assigned the next iteration that hasn't been executed yet.

Below is the program above modified to have a dynamic schedule:

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 4
#define N 16

int main ( ) {
  int i;

  #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);

    printf("Thread %d has completed iteration %d.\n",
omp_get_thread_num( ), i);
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

How much faster does this program run?

Dynamic Schedule Overhead

Dynamic scheduling is better when the iterations may take very different amounts of time. However, there is some overhead to dynamic scheduling.

After each iteration, the threads must stop and receive a new value of the loop variable to use for its next iteration.

The following program demonstrates this overhead:

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 16
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads dynamically.\n", N,
THREADS);
  #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

How long does this program take to execute?

If we specify static scheduling, the program will run faster:

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 16
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads statically.\n", N,
THREADS);
  #pragma omp parallel for schedule(static) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Chunk Sizes

We can split the difference between static and dynamic scheduling by using *chunks* in a dynamic schedule.

Here, each thread will take a set number of iterations, called a “chunk”, execute it, and then be assigned a new chunk when it is done.

By specifying a chink size of 100 in the program below, we markedly improve the performance:

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 16
#define N 100000000
#define CHUNK 100

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads dynamically.\n", N,
THREADS);
  #pragma omp parallel for schedule(dynamic, CHUNK)
num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }
  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Increasing the chunk size makes the scheduling more static, and decreasing it makes it more dynamic.

Guided Schedules

Instead of static, or dynamic, we can specify *guided* as the schedule.

This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.

How does the program above perform with a guided schedule?

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 16
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads guided.\n", N, THREADS);
  #pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

How does our program with iterations that take different amounts of time perform with guided scheduling?

```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>

#define THREADS 4
#define N 16

int main ( ) {
  int i;

  #pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);

    printf("Thread %d has completed iteration %d.\n",
omp_get_thread_num( ), i);
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Conclusion

OpenMP for automatically splits for loop iterations for us.

But, depending on our program, the default behavior may not be ideal.

For loops where each iteration takes roughly equal time, static schedules work best, as they have little overhead.

For loops where each iteration can take very different amounts of time, dynamic schedules, work best as the work will be split more evenly across threads.

Specifying chunks, or using a guided schedule provide a trade-off (uma alternativa) between the two.

Choosing the best schedule depends on understanding your loop.