Embedded System Software Design

Project 1

Objective:

Since multi-core architectures are widely developed in most of hardware platforms, feasible task management mechanisms for multi-core systems are also proposed. In this project, we will implement Global scheduling and Partition scheduling by using a serious of system calls.

Problem Definition:

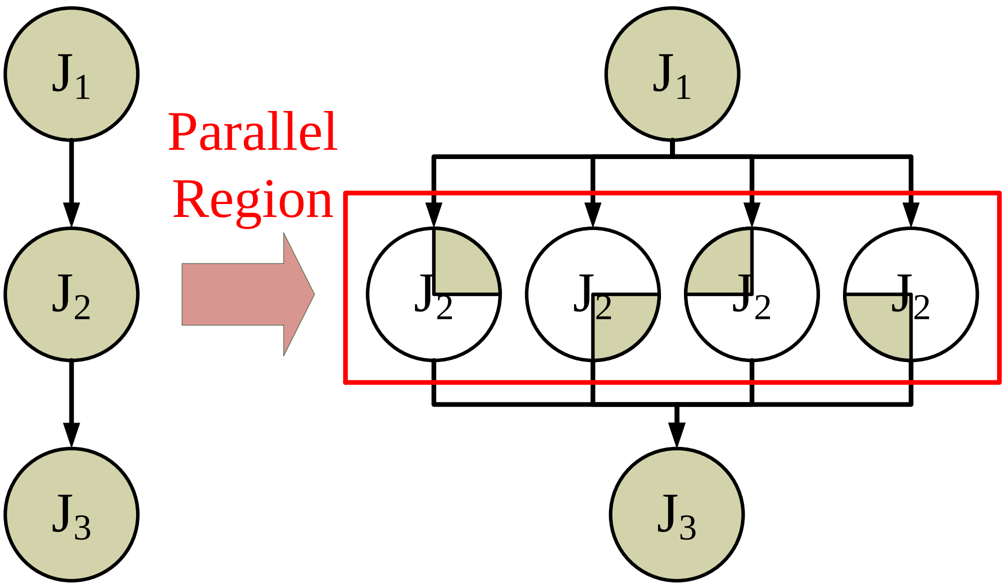
To input a real user application, and runs on a multi-core system. By parallelizing the some regions in program, we can reduce the execution duration, but make higher scheduling complexity. We compare Global scheduling and Partition scheduling on Linux system.

Experimental Environment:

- ✓ PC: i7 (8 cores) or 4 cores
- ✓ OS: Ubuntu 15.14
- ✓ Compiler: GCC 5.2.1

Parallelizing user program:

OpenMP is an API that supports multi-core platforms. After parallelizing some regions in user program, we can represent a task as a task graph:



where J_2 is separated in four threads, and who are executed on four cores. However, how to efficiently manage these threads to make a better system benefit is an important issue on multi-core systems.

We can add the command of OpenMP before the loops to averagely assign computations into threads. The format of command is:

#programa omp directive [clause]

Directive type (From: http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx):

Directive	Description
atomic	Specifies that a memory location that will be updated atomically.
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.
critical	Specifies that code is only executed on one thread at a time.
flush (OpenMP)	Specifies that all threads have the same view of memory for all shared objects.
for (OpenMP)	Causes the work done in a for loop inside a parallel region to be divided among threads.
master	Specifies that only the master thread should execute a section of the program.
ordered (OpenMP Directives)	Specifies that code under a parallelized for loop should be executed like a sequential loop.
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel.
sections (OpenMP)	Identifies code sections to be divided among all threads.
single	Lets you specify that a section of code should be executed on a single thread, not necessarily the master thread.

threadprivate	Specifies that a variable is private to a thread.

Clause type (From http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx):

Clause	Description
copyin	Allows threads to access the master thread's value, for a threadprivate variable.
copyprivate	Specifies that one or more variables should be shared among all threads.
default (OpenMP)	Specifies the behavior of unscoped variables in a parallel region.
firstprivate	Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct.
if (OpenMP)	Specifies whether a loop should be executed in parallel or in serial.
lastprivate	Specifies that the enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration (for-loop construct) or last section (#pragma sections).
nowait	Overrides the barrier implicit in a directive.
num_threads	Sets the number of threads in a thread team.
ordered (OpenMP Clauses)	Required on a parallel for (OpenMP) statement if an ordered (OpenMP Directives)directive is to be used in the loop.
private (OpenMP)	Specifies that each thread should have its own instance of a variable.

reduction	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
schedule	Applies to the for (OpenMP) directive.
shared (OpenMP)	Specifies that one or more variables should be shared among all threads.

Example:

```
#pragma omp parallel for num_threads(4)
for (i=0; i < num_steps; i++)
{
        MultiplyMatrix();
}
```

We parallel the following for loop, and set the loop will be separately executed in four threads. Note that the default setting of the number of threads refers to the hardware architecture where the program running on.

Implement global scheduling

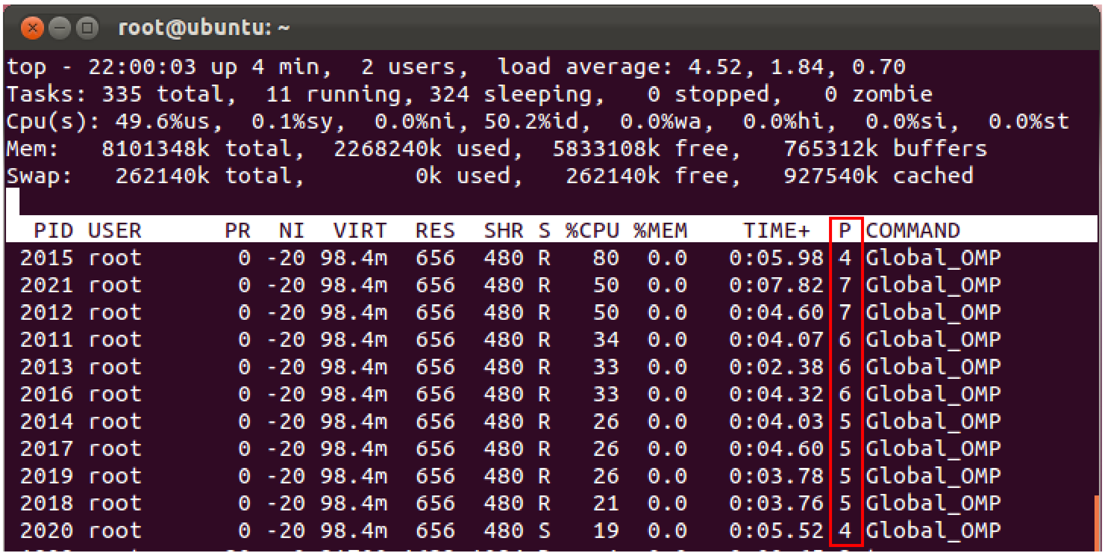
Linux provides global scheduling and run-time load balance mechanism. Therefore, we have to add some system calls to limit the cores that threads run on. We can use function "CPU_ZERO(&set) "and "CPU_SET(i, &set)"to set the state of cores in unable or enable.

Example:

```
#define _GNU_SOURCE

cpu_set_t    set;
CPU_ZERO(&set);                    //disable all CPUs
CPU_SET(1, &set);                  //enable CPU#1
CPU_SET(2, &set);                  //enable CPU#2
```

Assuming we execute a program on a PC with 8 cores, and only allow the program runs on CPU 4-7. We can insert the following codes in the program to implement that.

```
#define _GNU_SOURCE
#include <sched.h>

cpu_set_t    set;
int ret;

CPU_ZERO(&set);
for(i=7; i >3; i--) CPU_SET(i, &set);
```



The above figure shows the scheduling state with 12 threads and running on Core 4-7. The threads are dynamically migrated to the idle core. It makes the threads can be executed as soon as possible. We cannot predict which core the thread is running on.

To monitor the behavior that the system migrates threads to idle CPUs, we can use system call “sched_getcpu() ”to get CPU’s ID that the specific thread runs on. We insert a piece of code in for loop to compare the previous CPU and current CPU which the thread runs on. When the thread is running, if the thread is migrated to the other idle CPU by system, it will show the information and record the migration times as shown in following figure.

```
root@ubuntu: /home/procolor/MyProject
sys     0m0.004s
root@ubuntu:/home/procolor/MyProject# time ./Global_OMP
The thread 9 is moved from CPU7 to CPU5
The thread 4 is moved from CPU7 to CPU5
The thread 8 is moved from CPU4 to CPU6
The thread 10 is moved from CPU4 to CPU6
The thread 7 is moved from CPU5 to CPU4
The thread 7 is moved from CPU4 to CPU5
The thread 6 is moved from CPU7 to CPU6
The thread 2 is moved from CPU4 to CPU7
The thread 9 is moved from CPU5 to CPU4
The thread 8 is moved from CPU6 to CPU4
The thread 6 is moved from CPU6 to CPU7
The thread 7 is moved from CPU5 to CPU4
The thread 7 is moved from CPU4 to CPU5
The thread 3 is moved from CPU5 to CPU6
The thread 2 is moved from CPU7 to CPU6
The thread 9 is moved from CPU4 to CPU6
The thread 7 is moved from CPU5 to CPU7
Total number of migration is 17
real    0m19.447s
user    1m5.064s
sys     0m0.012s
```

Example:

```
#include <sched.h>

int cpu_alloc[set_num_threads];
int migration_counter=0;

if(sched_getcpu()!=cpu_alloc[omp_get_thread_num()])
{
        printf("The thread %d is moved from CPU%d to CPU%d \n",
omp_get_thread_num(), cpu_alloc[omp_get_thread_num()], sched_getcpu());
        cpu_alloc[omp_get_thread_num()]= sched_getcpu();
        migration_counter++;
}
```

Implement partition scheduling

In global scheduling, we only define the region that the threads can run on. If we want to assign a specific thread to a specific core, we can use the function "`omp_get_thread_num()`" and "`getpid()`" to derive the real thread ID, then according its ID to set its CPU affinity.

Due to the created thread ID is fixed in a program, we can program the for loop without partitioning the threads:

```
#pragma omp parallel for num_threads(set_num_threads) private(PID, ret, set)
for (i=0; i < omp_get_num_threads(); i++)
{
    PID=getpid()+omp_get_thread_num();
    CPU_ZERO(&set);
    CPU_SET(4+i%4, &set);
    ret=sched_setaffinity(PID, sizeof(cpu_set_t), &set);
}
```

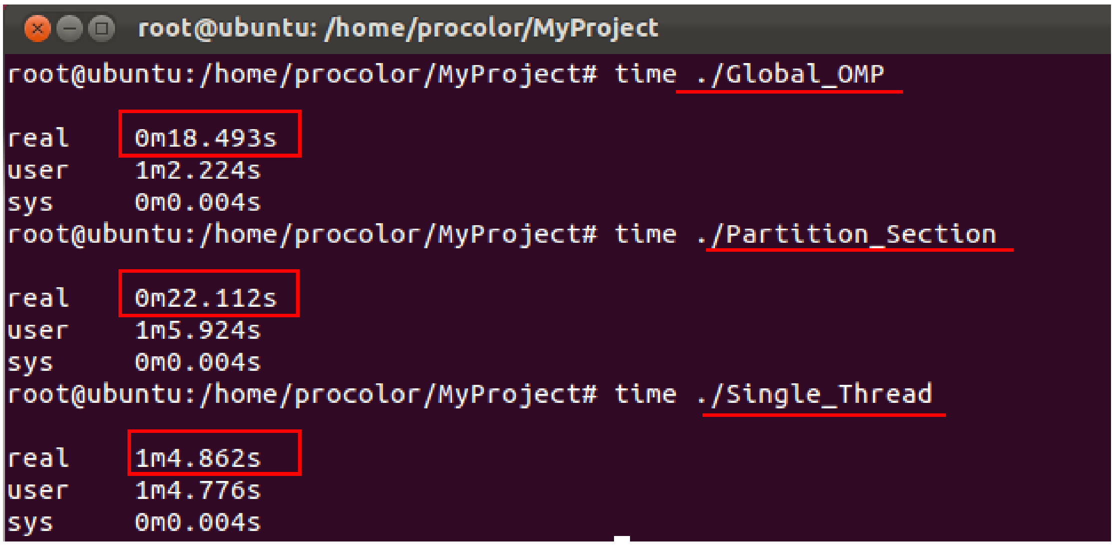
Because the experiment platform is a PC with 8 cores in this example, we assign the threads to Core 4-7.

```
root@ubuntu: ~
top - 22:52:58 up 57 min,  2 users,  load average: 0.86, 1.02, 0.57
Tasks: 332 total,  14 running, 318 sleeping,   0 stopped,   0 zombie
Cpu(s): 22.7%us,  0.2%sy,  0.0%ni, 77.2%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8101348k total,  2278244k used,  5823104k free,   782308k buffers
Swap:   262140k total,        0k used,   262140k free,   960688k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  P COMMAND
 2625 root       0 -20 98.4m  652  480 R   37  0.0   0:01.12 7 Partition_Secti
 2626 root       0 -20 98.4m  652  480 R   37  0.0   0:01.12 4 Partition_Secti
 2627 root       0 -20 98.4m  652  480 R   15  0.0   0:00.44 5 Partition_Secti
 2631 root       0 -20 98.4m  652  480 R   15  0.0   0:00.44 5 Partition_Secti
 2632 root       0 -20 98.4m  652  480 R   15  0.0   0:00.44 6 Partition_Secti
 2623 root       0 -20 98.4m  652  480 R   14  0.0   0:00.43 5 Partition_Secti
 2624 root       0 -20 98.4m  652  480 R   14  0.0   0:00.43 6 Partition_Secti
 2628 root       0 -20 98.4m  652  480 R   14  0.0   0:00.43 6 Partition_Secti
 2622 root       0 -20 98.4m  652  480 R    3  0.0   0:00.09 4 Partition_Secti
 2629 root       0 -20 98.4m  652  480 R    3  0.0   0:00.09 7 Partition_Secti
 2630 root       0 -20 98.4m  652  480 R    3  0.0   0:00.09 4 Partition_Secti
 2633 root       0 -20 98.4m  652  480 R    3  0.0   0:00.08 7 Partition_Secti
```

The above figure shows the scheduling state with 12 threads and running on Core 4-7. The threads are executed on dedicated cores. It is predictable that the thread will be finished in which core. Without migrating to the idle core, it makes longer response time to finish a thread.

Comparison

Finally, we can compare global scheduling and partition scheduling with the program without parallelism. We can use command ``time“ to estimate the total execution time of the program.



In the result, we can figure out global scheduling has a better performance than partition scheduling. And after parallelizing the program, we strongly reduce the execution time.

Command Line:

Compile the program with OpenMP: gcc –fopenmp xxx.c –o xxx
Time evaluate: time ./xxx

Display top CPU process: top –H –p <PID>
(press f → j to show the allocated CPU)

Crediting :

The number of created threads must be twice as the number of CPUs, and the number of CPUs should be more than 3.

[Global Scheduling. 40%]

- Describe how to implement multithread by using OpenMP **15%**
- Describe how to estimate task migration **10%**
- Show the result of task migration and describe why task is migrated **15%**

[Partition Scheduling. 40%]

- Describe how to implement partition scheduling **20%**
- Show the scheduling states of tasks **20%**

[Result. 20%]

- Compare the response time of the program in three execution types (Serial, Global, Partition)

[Bonus. 10%]

- Analyze the performance of three execution types (Serial, Global, Partition) in threads level.

Project submit

Submit deadline: 09 :00, April. 21, 2017

Submission : Moodle

File name format : ESSD_Student ID_HW1.rar

※ Strictly prohibited copying !

ESSD _Student ID_HW1.rar must inculde the report (.pdf) and source code.

嚴禁抄襲，發生該類似情況者，一律以零分計算