Parallel Computing

Academic year – 2020/21, spring semester

Computer science

Lecture 4

Lecturer, instructor:

Balakshin Pavel Valerievich

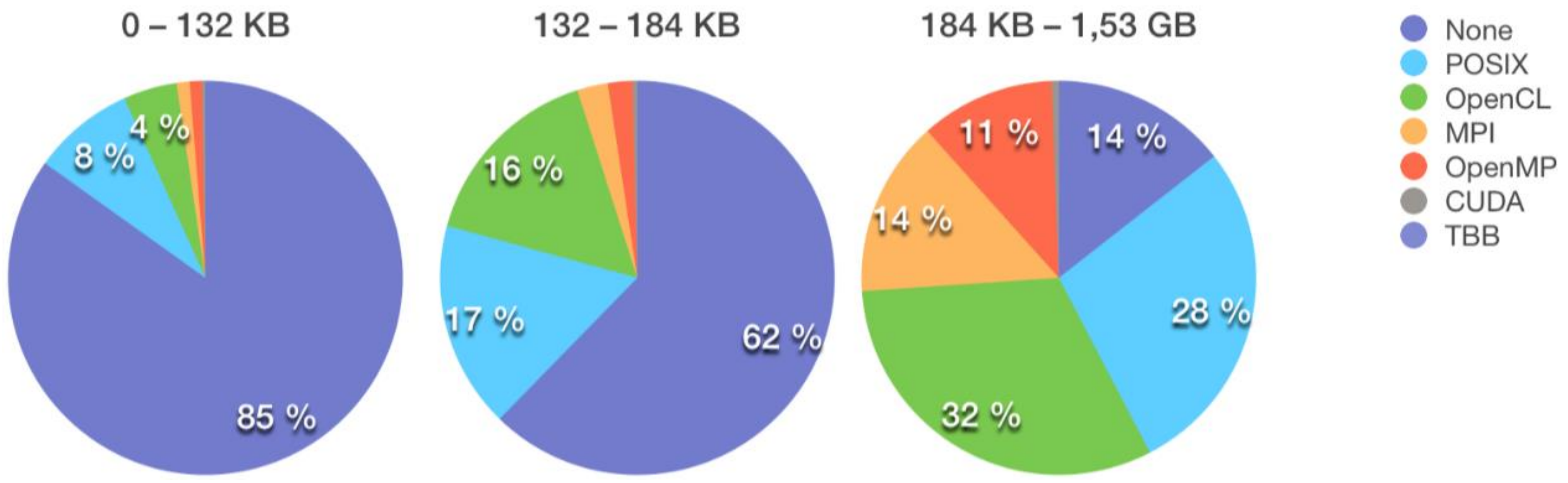
(pvbalakshin@itmo.ru; pvbalakshin@hdu.edu.cn)

Assistant:

Liang Tingting

(liangtt@hdu.edu.cn)

Choosing a language for parallel programming technology



Distribution based on repository size

According to the research of Bernhardt G.V. (2016)

Brief OpenMP review

Parallel architecture	**SMP** (shared memory parallelism)
Specification year	**1997** (the latest version 5.0 – 2018)
Programming languages	C/C++, Fortran
Support from different compilers	https://www.openmp.org/resources/openmp-compilers-tools/ **gcc, icc** – supported most of OpenMP 4.5 **IBM compiler** – OpenMP 4.5 **Clang** – OpenMP 4.5 **MS VS2019** – OpenMP 2.0 Common «denominator» – OpenMP 2.0

Advantages of OpenMP

- Cross-platform support
- Low parallel transformation ratio
- Incremental (gradual) parallelization
- Forward compatibility with old compilers
- Backward compatibility
- Autoscaling
- Support by the largest IT giants (AMD, IBM, Intel, HP, Nvidia, Oracle Corporation, …), good future evolution

Disadvantages of OpenMP

- It's hard to find and correct synchronization and race errors. Handling of standard errors is not developed.
- Applicable only in SMP-systems, so scalability is limited by memory architecture.
- Requires explicit compiler support.
- High overhead costs on providing parallel operation of the program (due to high-level of OpenMP syntax).
- No mechanisms of thread binding to processors.
- Limited support for GPU computing (only since OpenMP 4.0).

Introduction to OpenMP

1. Use the «–fopenmp» key for the compiler.
   This is an option in gcc. For other compilers, the option has a different look.
2. Add header #include <omp.h>
   This header file is needed only if functions with the "omp_" prefix will be used (without which you can easily do it in a parallel program). If these functions are used, the program loses the forward compatibility property.

Compiler	**Option style**
gcc	-fopenmp
icc (Intel C/C++ compiler)	-openmp
Sun C/C++ compiler	-xopenmp
Visual Studio C/C++ compiler	/openmp
PGI (Nvidia C/C++ compiler)	-mp

```
#if defined(_OPENMP)
   omp_set_num_threads(num_threads)
#else
   printf(“No 2nd arg = %d)\n", omp_get_thread_num(num_threads));
#endif
```

"omp parallel" compiler directive

Result of simple program:

```
printf("Hello, OpenMP!\n");
```

```
Hello, OpenMP!
```

```
#pragma omp parallel
printf("Hello, OpenMP!\n");
```

When running on a 4 cores processor:

```
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
Hello, OpenMP!
```

When running on a single core processor:

```
Hello, OpenMP!
```

num_threads(k) parameter

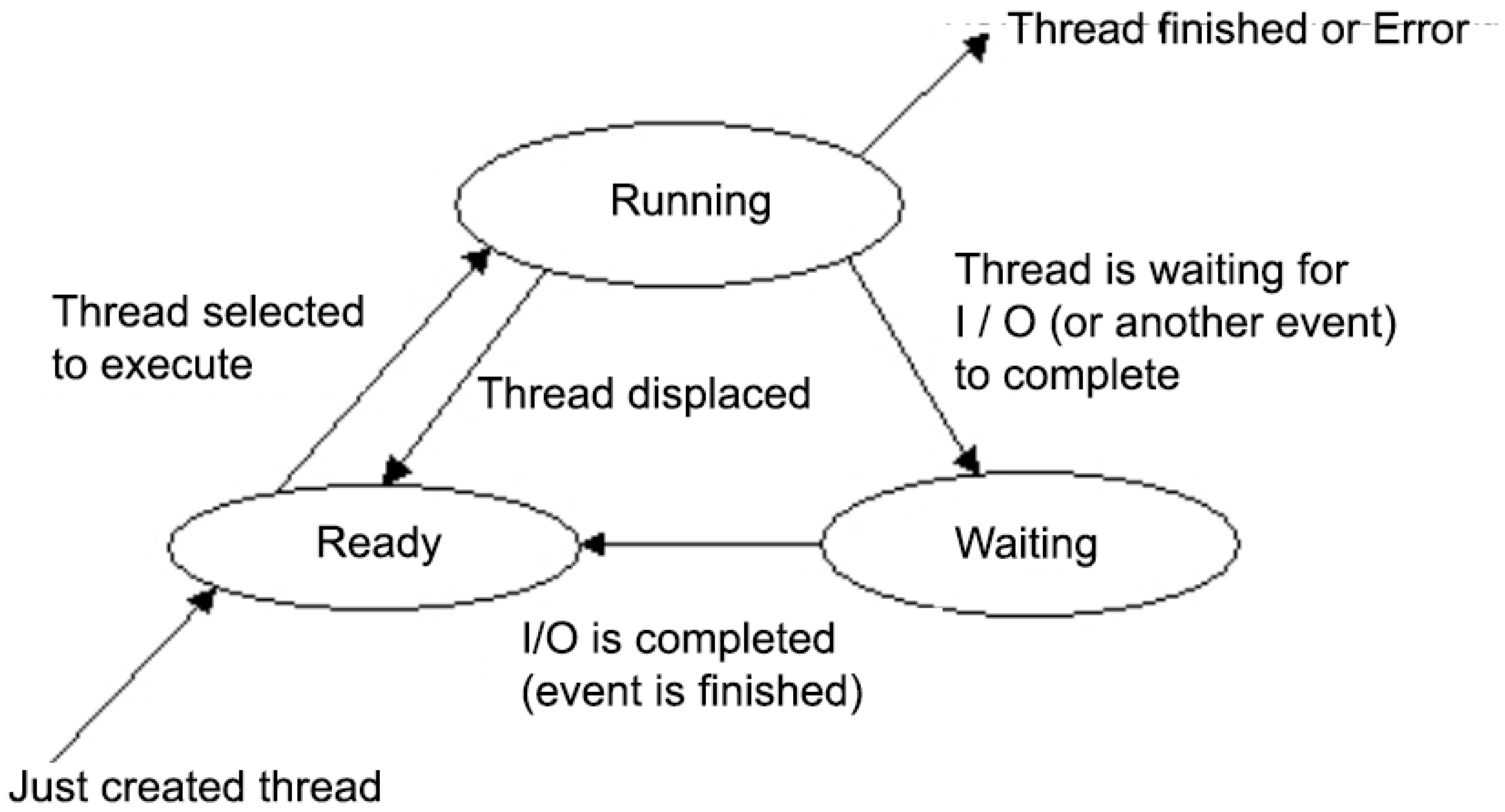
```
#pragma omp parallel num_threads(8)
printf("Hello, OpenMP (thread #%u)\n", omp_get_thread_num());
```

When running on a *n* cores processor (any *n*):

```
Hello, OpenMP (thread #0)
Hello, OpenMP (thread #7)
Hello, OpenMP (thread #4)
Hello, OpenMP (thread #5)
Hello, OpenMP (thread #6)
Hello, OpenMP (thread #1)
Hello, OpenMP (thread #2)
Hello, OpenMP (thread #3)
```

- The order of code execution by threads is unpredictable.
- The threads are numbered from zero.
- Thread number 0 is called a master thread.
- When calling the omp_get_thread_num function in the single thread mode, it always returns 0 (= the number of the master thread).

Thread lifecycle


Thread finished or Error
Running
Thread selected to execute
Thread displaced
Thread is waiting for I / O (or another event) to complete
Ready
Waiting
I/O is completed (event is finished)
Just created thread

omp_get_thread_num function

Sequential program

```
sort_matrix(A);
transpose_matrix(B);
```

Parallel program

```
#pragma omp parallel num_threads(2)
if (omp_get_thread_num( ) == 0)
   sort_matrix(A);
else
   transpose_matrix(B);
```

OpenMP will create two threads, one of which will sort matrix_**A** (a master thread) and the other one will simultaneously transpose matrix_**B** (thread with #1). Parallelization is performed without conflicts as these two operations do not affect each other in any way.

An example when paralleling is impossible (the second operation cannot be performed until the first one is executed):

```
sort_matrix(A, lines, columns);
transpose_matrix(A, lines, columns);
```

omp_set_num_threads & *omp_get_num_threads* functions

```
#pragma omp parallel num_threads(3)
   #pragma omp master
      {
         printf_s("%d\n", omp_get_num_threads( ));
      }
```

Parallel program #1

```
omp_set_num_threads(3);
#pragma omp parallel
   {
   #pragma omp master
      {
         printf_s("%d\n", omp_get_num_threads( ));
      }
   }

printf_s("%d\n", omp_get_num_threads( ));
```

Parallel program #2

section parameter

Sequential program
= old compiler

```
sort_matrix(A);
transpose_matrix(B);
```

Parallel program

```
#pragma omp parallel sections {
   #pragma omp section
   sort_matrix(A);
   #pragma omp section
   transpose_matrix(B);
}
```

OpenMP will create two threads, one of which will sort matrix_**A** (a master thread) and the other one will simultaneously transpose matrix_**B** (thread with #1). This solution is similar to the previous ones except that it is impossible to say which of the threads (0 or 1) will perform the sorting and which one will perform the transposition.

The number of threads created can be increased by simply adding another #pragma omp section.