OpenMP book notes

Data: 20-12-2022
Related: OpenMP
Tags: #ARQC #SoftwareEngineering #C

```c
#include <stdio.h>
#include <omp.h>     // The OpenMP include file

int main()
{
   #pragma omp parallel
   {
      printf(" Hello ");
      printf(" World \n");
   }
}
```

```
Hello World
Hello World
Hello World
Hello World
```

The four threads, however, execute concurrently. The instructions executed by each thread are unordered with respect to the other threads. The printf statements within each thread follow the order defined by the program, but between the different threads, they have no specified order at all. So the output may be something like:

```
Hello Hello World
World
Hello  World
Hello  World
```

> Quote
>
> Another way to think about this is that every legally allowed way to interleave the statements defines a possible execution order for the program. Your challenge as a parallel programmer is to make sure that all possible interleavings yield correct results.

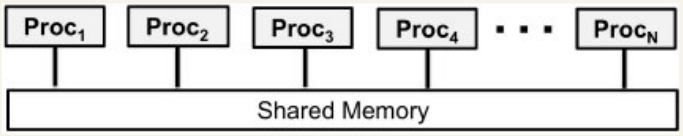

Figure 1.2: **A Symmetric Multiprocessor computer consisting of N Processors (Procs) sharing a region of memory.**

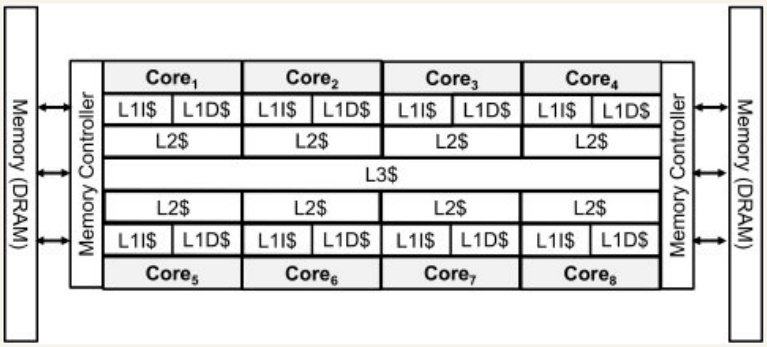

Figure 1.3: **A typical multicore CPU with 8 cores** – Each Core has level one caches for data (L1D$) and instructions (L1I$), a unified level 2 cache (L2$) and a shared level 3 cache (L3$). This CPU has two memory controllers each with three channels to access off-chip memory (DRAM).

Memory is hierarchical in a modern system. Consider a typical multicore CPU as shown in Figure 1.3. There is a small cache right next to each processor to hold data and instructions for the program. These are called level one caches since they are the closest to the cores; the L1D$ (Data) and L1I$ (Instruction) caches. Each core has a level 2 cache (L2$) that holds both data and program instructions, hence this is called a unified cache. Finally, all the cores in a CPU may share an additional level of cache, a level 3 or L3$. The caches are small. For a typical high-end CPU, the sizes[1] of these caches are:

- 32 KiloBytes Level 1 Data cache per core
- 32 KiloBytes Level 1 Instruction cache per core
- 256 KiloBytes Level 2 cache, unified (holds both data and instructions) per core
- 8 MegaBytes Level 3 cache shared between cores

not the case. The time required to access a value in memory depends on where that value is in the memory hierarchy. Consider the time required to access a single value from memory, that is, the latency of a memory access. Using values for a typical, high-end CPU the values range across the memory hierarchy are as follows:

- L1 Cache latency = 4 cycles
- L2 Cache latency = 12 cycles
- L3 Cache latency = 42 cycles
- DRAM access = 250 cycles

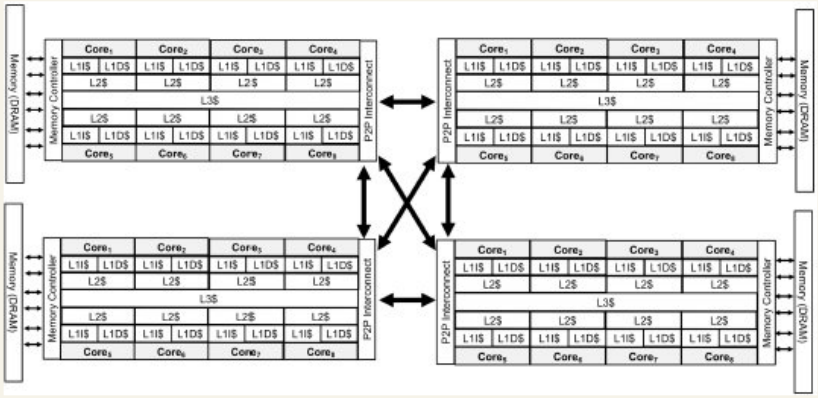

Figure 1.4: **A nonuniform memory architecture (NUMA) system with four CPUs connected by point to point (P2P) interconnects** – All DRAM is accessible to all the cores meaning the cost of accessing different regions of memory varies significantly across the system.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 4

void *PrintHelloWorld(void *InputArg)
{
    printf(" Hello ");
    printf(" World \n");
}

int main()
{
    pthread_t threads[NUM_THREADS];
    int id;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for (id = 0; id < NUM_THREADS; id++) {
        pthread_create(&threads[id], &attr, PrintHelloWorld, NULL);
    }

    for (id = 0; id < NUM_THREADS; id++){
        pthread_join(threads[id], NULL);
    }

    pthread_attr_destroy(&attr);
    pthread_exit(NULL);
}
```

There are only two reasons to write a parallel program: to solve a fixed size problem in less time, or to solve a larger problem in a reasonable amount of time.
In either case, it is all about performance.

> Quote
>
> Speedup is a ratio of run times for a program. Ideally you run a program with the best available serial algorithm on one processor. Then run it again with parallel software on *P* processors. If we define *Ts* as the run time for the serial program and *TP* as the run time for a parallel program running on P processors, we define the speedup as:

In the ideal case, the speedup is equal to the number of processors. If you double the number of processors, the performance should double. When a program follows this speedup trend we say that it has “perfect linear speedup”. We measure how closely we track perfect linear speedup with a scaled speedup called the *parallel efficiency*.

$$eff = \frac{S(P)}{P}$$

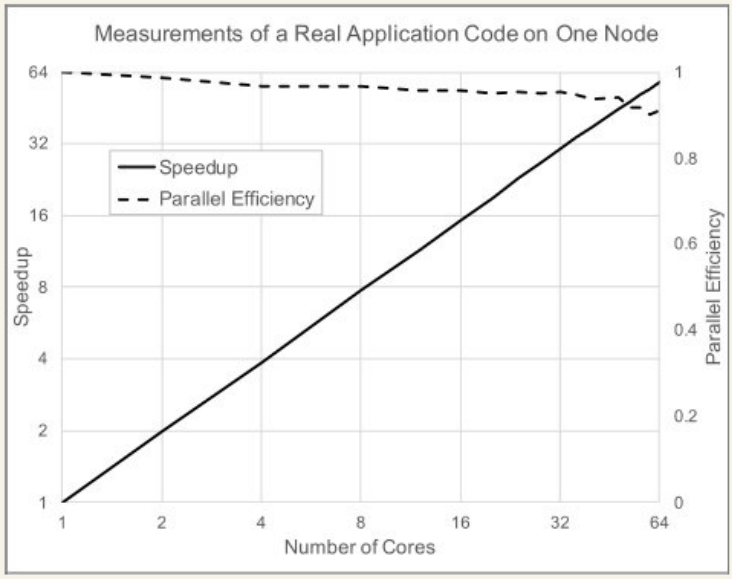

Figure 2.1: **Speedup and Efficiency as a function of the number of cores** – Performance measured on a single node of a cluster using log base 2 for the x and y axes for the speedup and log-linear scales for the corresponding parallel efficiency. The application is the fast multipole code Octo-tiger parallelized with the HPX task driven programming model. The node is an Intel(R) Xeon Phi(TM) 7250 68C processor running at 1.4GHz.

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel regions, teams of threads, structured blocks, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads() void omp_set_num_threads()	The SPMD Pattern: Create a parallel region and split up the work using the number of threads and thread IDs
double omp_get_wtime()	Timing blocks of code, Speedup, and Amdahl's law
export OMP_NUM_THREADS=N	Internal control variables and setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Implications of interleaved execution, race conditions and synchronization
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op: list)	Reductions of values across a team of threads
schedule(static [, chunk]) schedule(dynamic) [,chunk])	Loop schedules, loop overheads and load balance
private (list) firstprivate (list) shared(list)	The OpenMP Data environment: the default rules and the clauses that modify default behavior
default (none)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on worksharing constructs, the high cost of barriers, and flushing memory
#pragma omp single	Work done by a single thread
#pragma omp task #pragma omp taskwait	Tasks, Task completion and the data environment for tasks

Critical

The most basic synchronization construct defines a mutual exclusion relationship for code running with multiple threads. Mutual exclusion stipulates that if one thread is executing a block of code and a second thread tries to execute the same code, that second thread will pause and wait until the first thread has finished with the code. In OpenMP, we define a block of code that executes with mutual exclusion with a critical construct as shown in Table 4.9.

Table 4.9: **A *critical construct* defines a block of code that executes with mutual exclusion** – i.e., only one thread at a time executes the code with threads potentially waiting their turn at the beginning of the construct. The curly braces and `END CRITICAL` are not required when the block contains only a single statement.

#pragma omp critical
{
`...  one or more lines of code`
}

!$omp critical `...  one or more lines of code` **!$omp end critical**

```
 1  #include <stdio.h>
 2  #include <omp.h>
 3
 4  int main()
 5  {
 6      float  res = 0.0;
 7      #pragma omp parallel
 8      {
 9          float B; int i, id, nthrds;
10          id = omp_get_thread_num();
11          nthrds = omp_get_num_threads();
12          for (i = id; i < niters; i += nthrds) {
13              B = big_job(i);
14              #pragma omp critical
15                  res += consume(B);
16          }
17      } // end of parallel region
18  }
```

To prevent data races, we want the consume() function to run to completion one thread at a time; that is, we want the function to execute with mutual exclusion. This is precisely what placing the function call inside a critical construct accomplishes.