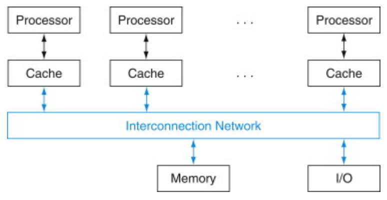
6.5 Multicore and other shared memory multiprocessors

While hardware multithreading improved the efficiency of processors at modest cost, the big challenge of the last decade has been to deliver on the performance potential of Moore's Law by efficiently programming the increasing number of processors per chip.

Given the difficulty of rewriting old programs to run well on parallel hardware, a natural question is: what can computer designers do to simplify the task? One answer was to provide a single physical address space that all processors can share, so that programs need not concern themselves with where their data are, merely that programs may be executed in parallel. In this approach, all variables of a program can be made available at any time to any processor. The alternative is to have a separate address space per processor that requires that sharing must be explicit; we'll describe this option in the COD Section 6.7 (Clusters, warehouse scale computers, and other message-passing multiprocessors). When the physical address space is common then the hardware typically provides cache coherence to give a consistent view of the shared memory (see COD Section 5.8 (A common framework for memory hierarchy)).

As mentioned above, a *shared memory multiprocessor* (SMP) is one that offers the programmer a *single physical address space* across all processors—which is nearly always the case for multicore chips—although a more accurate term would have been shared-*address* multiprocessor. Processors communicate through shared variables in memory, with all processors capable of accessing any memory location via loads and stores. The figure below shows the classic organization of an SMP. Note that such systems can still run independent jobs in their own virtual address spaces, even if they all share a physical address space.

Figure 6.5.1: Classic organization of a shared memory multiprocessor (COD Figure 6.7).



Single address space multiprocessors come in two styles. In the first style, the latency to a word in memory does not depend on which processor asks for it. Such machines are called *uniform memory access (UMA)* multiprocessors. In the second style, some memory accesses are much faster than others, depending on which processor asks for which word, typically because main memory is divided and attached to different microprocessors or to different memory controllers on the same chip. Such machines are called *nonuniform memory access (NUMA)* multiprocessors. As you might expect, the programming challenges are harder for a NUMA multiprocessor than for a UMA multiprocessor, but NUMA machines can scale to larger sizes, and NUMAs can have lower latency to nearby memory.

Uniform memory access (***UMA***): A multiprocessor in which latency to any word in main memory is about the same no matter which processor requests the access.

Nonuniform memory access (***NUMA***): A type of single address space multiprocessor in which some memory accesses are much faster than others depending on which processor asks for which word.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data; otherwise, one processor could start working on data before another is finished with it. This coordination is called *synchronization*, which we saw in COD Chapter 2 (Instructions: Language of the Computer). When sharing is supported with a single address space, there must be a separate mechanism for synchronization. One approach uses a *lock* for a shared variable. Only one processor at a time can acquire the lock, and other processors interested in shared data must wait until the original processor unlocks the variable. COD Section 2.11 (Parallelism and instructions: synchronization) of COD Chapter 2 (Instructions: Language of the Computer) describes the instructions for locking in the ARMv8 instruction set.

Synchronization: The process of coordinating the behavior of two or more processes, which may be running on different processors.

Lock: A synchronization device that allows access to data to only one processor at a time.

Example 6.5.1: A simple parallel processing program for a shared address space.

Suppose we want to sum 64,000 numbers on a shared memory multiprocessor computer with uniforn memory access time. Let's assume we have 64 processors.

Answer

The first step is to ensure a balanced load per processor, so we split the set of numbers into subsets o same size. We do not allocate the subsets to a different memory space, since there is a single memory space for this machine; we just give different starting addresses to each processor. Pn is the number t identifies the processor, between 0 and 63. All processors start the program by running a loop that sur their subset of numbers:

```
sum[Pn] = 0;

for (i = 1000 * Pn; i < 1000 * (Pn + 1); i += 1)
   sum[Pn] += A[i];                  /* sum the assigned areas */
```

(Note the C code `i += 1` is just a shorter way to say `i = i + 1`.)

The next step is to add these 64 partial sums. This step is called a *reduction*, where we divide to conqu Half of the processors add pairs of partial sums, and then a quarter add pairs of the new partial sums, so on until we have the single, final sum. The figure below illustrates the hierarchical nature of this reduction.

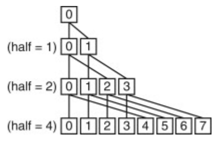
In this example, the two processors must synchronize before the "consumer" processor tries to read th
result from the memory location written by the "producer" processor; otherwise, the consumer may rea
old value of the data. We want each processor to have its own version of the loop counter variable `i`, s
must indicate that it is a "private" variable. Here is the code (`half` is private also):

```
half = 64;                          /* 64 processors in multiprocessor */

do
   synch();                         /* wait for partial sum completion */
   if (half % 2 != 0 && Pn == 0)
      sum[0] += sum[half - 1];
                                    /* Conditional sum needed when half is
                                       odd; Processor0 gets missing elemen
      half = half / 2;              /* dividing line on who sums */

      if (Pn < half)
         sum[Pn] += sum[Pn + half];
while (half > 1);                   /* exit with final sum in Sum[0] */
```

Reduction: A function that processes a data structure and returns a single value.

Figure 6.5.2: The last four levels of a reduction that sums results from each processor, from bottom to top (COD Figure 6.8).

For all processors whose number i is less than half, add the sum produced by processor number (i + half) to its sum.



Hardware/Software Interface

Given the long-term interest in parallel programming, there have been hundreds of attempts to build parallel programming systems. A limited but popular example is *OpenMP*. It is just an *Application Programmer Interface* (API) along with a set of compiler directives, environment variables, and runtime library routines that can extend standard programming languages. It offers a portable, scalable, and simple programming model for shared memory multiprocessors. Its primary goal is to parallelize loops and perform reductions.

Most C compilers already have support for OpenMP. The command to use the OpenMP API with the UNIX C compiler is just:

```
cc -fopenmp foo.c
```

OpenMP extends C using *pragmas*, which are just commands to the C macro preprocessor like `#define` and `#include`. To set the number of processors we want to use to be 64, as we wanted in the example above, we just use the command

```
#define P 64                        /* define a constant that
                                       we'll use a few times */
#pragma omp parallel num_threads(P)
```

That is, the runtime libraries should use 64 parallel threads.

To turn the sequential for loop into a parallel for loop that divides the work equally between all the threads that we told it to use, we just write (assuming `sum` is initialized to 0)

```
#pragma omp parallel for
for (Pn = 0; Pn < P; Pn += 1)
   for (i = 0; 1000 * Pn; i < 1000 * (Pn + 1); i += 1)
      sum[Pn] += A[i];              /* sum the assigned areas */
```

To perform the reduction, we can use another command that tells OpenMP what the reduction operator is and what variable you need to use to place the result of the reduction.

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
   FinalSum += sum[i];              /* Reduce to a single number */
```

Note that it is now up to the OpenMP library to find efficient code to sum 64 numbers efficiently using 64 processors.

While OpenMP makes it easy to write simple parallel code, it is not very helpful with debugging, so many programmers use more sophisticated parallel programming systems than OpenMP, just as many programmers today use more productive languages than C.

OpenMP: An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Given this tour of classic MIMD hardware and software, our next path is a more exotic tour of a type of MIMD architecture with a different heritage and thus a very different perspective on the parallel programming challenge.

PARTICIPATION ACTIVITY 6.5.1: Check yourself: Shared memory.

1) Shared memory multiprocessors cannot take advantage of task-level parallelism.
   - ○ True
   - ○ False

Elaboration

Some writers repurposed the acronym SMP to mean symmetric multiprocessor, to indicate that the latency from processor to memory was about the same for all processors. This shift was done to contrast them from large-scale NUMA multiprocessors, as both classes used a single address space. As clusters proved much more popular than large-scale NUMA multiprocessors, in this book we restore SMP to its original meaning, and use it to contrast against those that use multiple address spaces, such as clusters.

Elaboration

An alternative to sharing the physical address space would be to have separate physical address spaces but share a common virtual address space, leaving it up to the operating system to handle communication. This approach has been tried, but it has too high an overhead to offer a practical shared memory abstraction to the performance-oriented programmer.