Introduction to OpenMP

Eric J. Kostelich

ARIZONA STATE UNIVERSITY
SCHOOL OF **MATHEMATICAL & STATISTICAL SCIENCES**

October 1, 2018

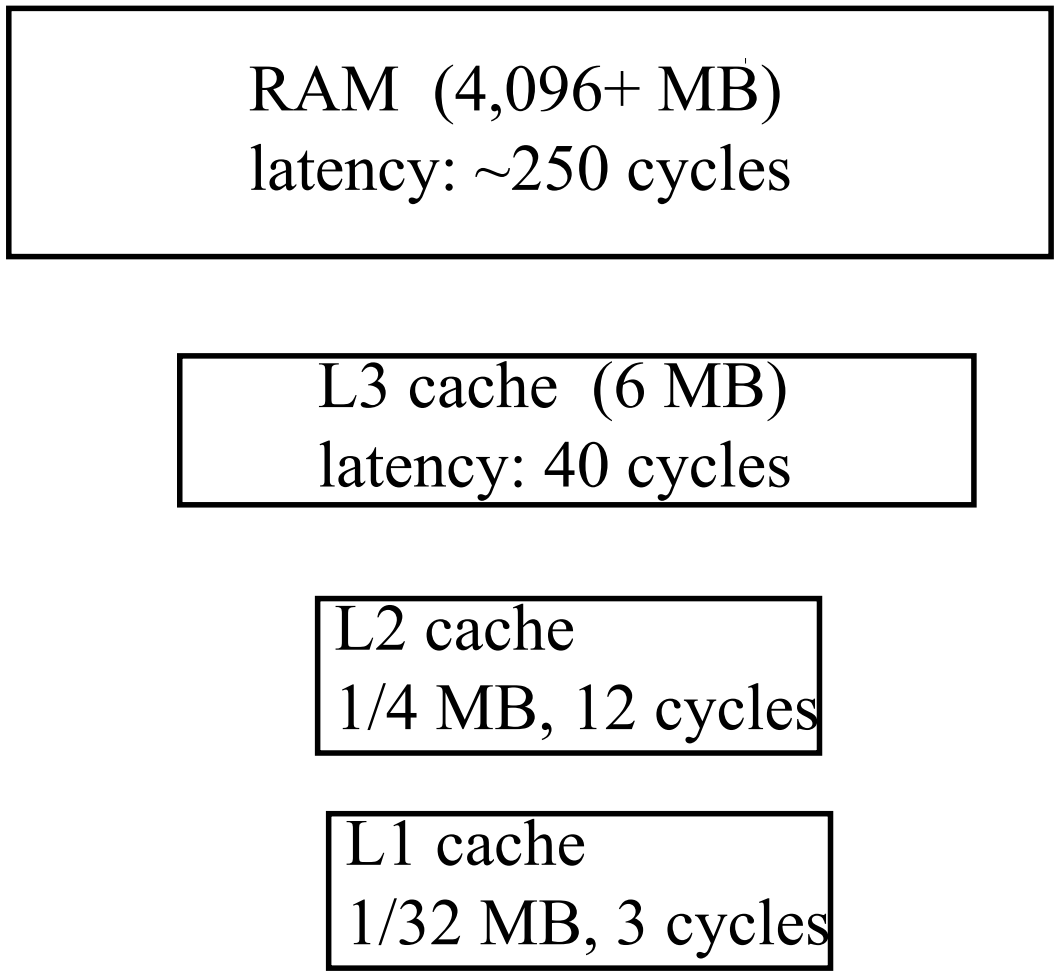
Unix processes

- Starting a program in a Unix/Linux environment involves many steps
- Dynamic linker gets code from shared libraries to resolve undefined references
- The kernel allocates a process id, an address space, and a scheduling slot
- Loader and kernel allocate a stack, a heap, and virtual memory and initializes the program counter
- The ps command-line utility shows the status of some or all running processes

Lightweight processes (“threads”)

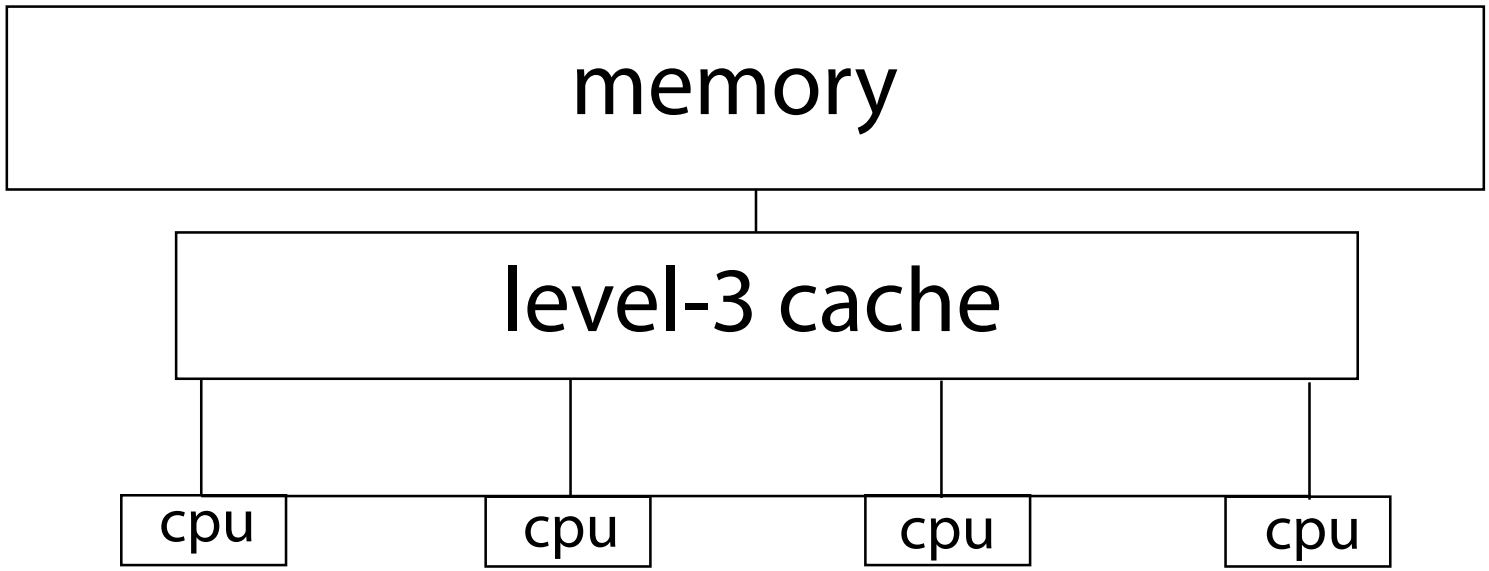
- Starts a partial copy of the main program
- Shares the same process id and code space of the parent
- Non-stack (e.g., static and global) variables are shared among threads
- Each thread maintains its own stack and runs autonomously
- When threads exit, their stack is reclaimed but the main program continues

Memory hierarchies for the Intel i7 Core processor



Multicore processors share the same L3 cache

- Usually L1 and L2 caches are separate (but much smaller than L3)
- Threads are most efficient when they process small amounts of memory at a time on x86 machines



The SMP paradigm has limited scalability

- All CPUs in a program have a common view of memory
- Any thread can change any value in memory
- If CPU 1 changes location n, and the other CPUs have cached the value in location n, then the hardware must refresh each cache
- The Columbia supercomputer at NASA Ames had 10,240 Itanium processors and 20 TB of memory operating as one system image
- Intel's Phi chips have 64–72 cores on one piece of silicon in the current generation

OpenMP is a paradigm for multicore programming

- Unlike POSIX threads, the programmer does not need to write the threading code
- OpenMP is managed through compiler "directives"
- The compiler writes the threading code
- The degree of speedup depends on the parallelizability of the innermost loops of a program and their memory access patterns

General caveats on threading

- A thread takes $\sim$ 500,000 cpu cycles to create
- So you need (at least) several million cycles worth of work for each thread to amortize the the startup cost
- Example: Given 4 cores and 4,000,000 parallelizable cycles of work
- It takes $4 \times 500{,}000 = 2{,}000{,}000$ cycles to create 4 threads (not parallelizable!)
- It takes 1,000,000 cycles to do the work on 4 cpus
- It takes another $\sim$ 1,000,000 cycles to destroy 4 threads

General caveats on threading, 2

- Many threads can easily overfill an L3 cache
- CPUs stall while waiting for memory
- The operating system still needs to run—and takes up memory as well
- The best performance often arises when you leave one core free
- "Hyperthreading" is not beneficial for floating-point code, because each core typically has only one floating-point unit

General caveats on threading, 3

- In general, threading at high levels of the code is more profitable than threading at lower levels
- Threads can be used to overlap computation with i/o
- Use threads to spawn side calculations from a GUI while keeping the interface responsive
- A threaded math library—such as Intel's Math Kernel Library—can provide many of the benefits with little extra work
- The MKL includes LAPACK for (dense) linear algebra, fast Fourier transforms, etc.

OpenMP directives

- OpenMP is implemented by compiler directives
- The directives look like comments and are ignored unless the compiler is running in OpenMP mode
- Example: Parallel SAXPY:

```
!$omp parallel do
do j=1,n
      y(j)=y(j)+a*x(j)
enddo
```

- The work is divided as equally as possible between threads

Example of operation

- The program operates in a single thread of execution until it reaches the parallel do
- At that point, threads are spawned and scheduled to do the work
- Fortran rule: At the end of a parallel do there is an implicit barrier—the main program waits until all threads have finished
- Following the enddo, the program continues in a single thread of execution
- The maximum number of threads may be determined dynamically or fixed with the environment variable OMP_NUM_THREADS

Example of operation, 2

- Requirements for parallelizability are similar as for vectorization
- Each iteration of the loop must be independent of the others
- The order of execution of the iterations must not matter
- Threads operate asynchronously and in unpredictable order
- All threads can access and update all elements of x and y—the programmer must assure that they are nonoverlapping

Compiling and running an OpenMP program

- With Intel Fortran:

  ifort –c –qopenmp mymod.f90
  ifort –c –qopenmp prog.f90
  ifort –qopenmp prog.o mymod.o [*MKL libraries*] –o prog

- The –qopenmp-report=1 option generates additional info on parallelized regions (and –qopenmp-report=2 even more info)
- Running the program:

  export OMP_NUM_THREADS=4 *spawn up to 4 threads*

  ./prog *arguments*

Fortran 2008 alternative: do concurrent

- Example: Parallel SAXPY:

```
do concurrent (j=1:n)      note parentheses and colon
    y(j)=y(j)+a*x(j)
enddo
```

- The do concurrent statement asserts that the loop is vectorizable and/or parallelizable
- Whether the compiler vectorizes or parallelizes is usually controlled by a compiler option or directive
- Otherwise, the requirements for the loop are similar to those for OpenMP threading

Elemental and pure subprograms

- A pure subroutine or function has no side effects
- A pure subroutine can modify arguments specified as intent(out) or intent(inout)
- A pure function may not modify its arguments—it simply returns a value
- pure routines may call only other pure (or elemental) routines
- They can access global (module) variables but may not alter their value
- They may not perform i/o or contain a stop statement

Example of a pure procedure

```
pure subroutine weird_add(a, b, c)
real,intent(in):: a, b
real,intent(out):: c
if(sin(a) > 0) then
   c = sin(a) + sin(b)
else
   c = cos(a) + cos(b)
endif
return
end subroutine weird_add
```

- All built-in Fortran functions are pure

Elemental subprograms

- Elemental subprograms are a restricted class of pure subprograms (elemental $\subset$ pure)
- The arguments of elemental subprograms must be scalar
- But an elemental subprogram can be called with array arguments, in which case the action is performed elementwise on every element
- Vectorization requires an inter-procedural optimization step if the elemental procedure is called outside of the module that defines it

Example of an elemental subroutine

```
elemental subroutine weird_add(a, b, c)
real,intent(in):: a, b
real,intent(out):: c
if(sin(a) > 0) then
   c = sin(a) + sin(b)
else
   c = cos(a) + cos(b)
endif
return
end subroutine weird_add
...
real:: x(10),y(10),z(10)
assign values to x and y
call weird_add(x,y,z)     loop on all 10 elements
```

Example of an elemental function

```
elemental real function weird(a, b)
real,intent(in):: a, b
if(sin(a) > 0) then
   weird = sin(a) + sin(b)
else
   weird = cos(a) + cos(b)
endif
return
end function weird
...
real:: x(10),y(10),z(10)
assign values to x and y
z=weird(x,y)     loop on all 10 elements
```

pure and elemental are aids to parallelization

- Pure and elemental routines can appear safely in OpenMP parallel regions and in do concurrent constructs
- Recommendation: If you identify a parallelizable section of code, rewrite it into a pure procedure (or an elemental one, if it is to be applied elementwise to an array)
- pure and elemental procedures must not have read or write statements, since changing the state of a file is considered a side effect