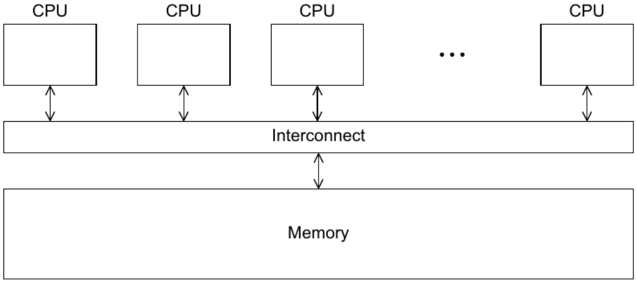
Reference Roadmap

The slide outlines contain references to the main course materials. Not everything has a reference but nearly all the materials can be found in the following references. The references are of the form `(<text>-<pages>)` so (PM-123) means page 123 of An Introduction to Parallel Programming by Peter S. Pacheco and Matthew Malensek.

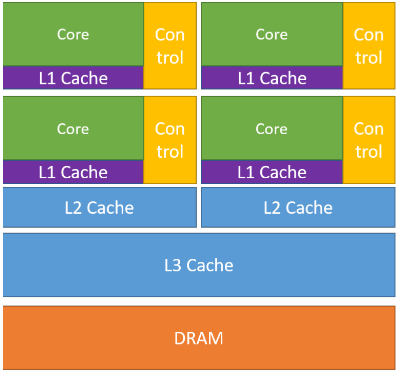
- (VE-V1) — The Science of Computing: The Art of High Performance Computing, Vol 1 by Victor Eijkhout
- (VE-V2) — Parallel Programming in MPI and OpenMP: The Art of HPC, Vol 2 by Victor Eijkhout
- (VE-V3) — Introduction to Scientific Programming in C++17/Fortran2008: The Art of HPC, Vol 3 by Victor Eijkhout
- (VE-V4) — Tutorials for High Performance Scientific Computing: The Art of HPC, Vol 4 by Victor Eijkhout
- (PM) — An Introduction to Parallel Programming by Peter S. Pacheco and Matthew Malensek.
- (KH) — Programming Massively Parallel Processors: A Hands-on Approach by David B. Kirk and Wen-mei W. Hwu.
- (SAB) — High Performance Computing Modern Systems and Practices by Thomas Sterling, Matthew Anderson, and Maciej Brodowicz.
- (GLS) — Using MPI: Portable Parallel Programming with the Message-Passing Interface by William Gropp, Ewing Lusk, and Anthony Skjellum.
- (GHTL) — Using Advanced MPI: Modern Features of the Message-Passing Interface by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk.

Definitions & Theory

- Symmetric Multi-Processor (SMP): Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources, memory, disk, etc. Also referred to as Uniform Memory Access (UMA) architecture.



Definitions & Theory



- Mostly utilized on single motherboard systems, one or more socket(s) (i.e. processors) each with multiple cores.
- Multi-threading approach to program utilization, primarily pthreads and OpenMP.

Definitions & Theory

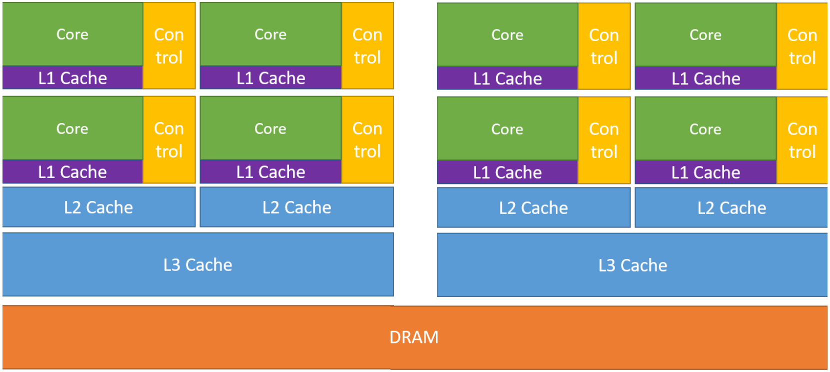
– Run `lscpu`

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          46 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   40
  On-line CPU(s) list:    0-39
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
    CPU family:           6
    Model:                79
    Thread(s) per core:   2
    Core(s) per socket:   10
    Socket(s):            2
    Stepping:             1
    CPU max MHz:          3100.0000
    CPU min MHz:          1200.0000
    BogoMIPS:             4390.03

...

Caches (sum of all):
  L1d:                    640 KiB (20 instances)
  L1i:                    640 KiB (20 instances)
  L2:                     5 MiB (20 instances)
  L3:                     50 MiB (2 instances)
```

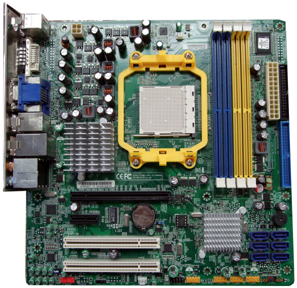
Definitions & Theory

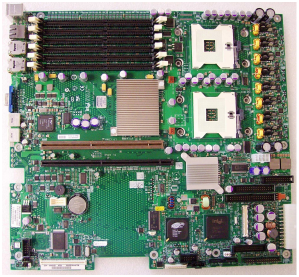


- Dual socket standard core and cache layout.

Definitions & Theory

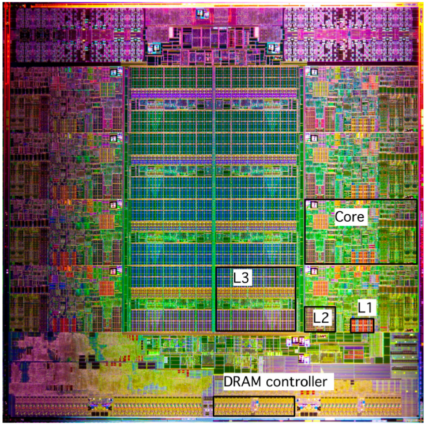
- Node: A Node incorporates all the functional elements required for computation. Essentially an entire computer system, hardware, OS, applications, ... Large scale systems, i.e. nearly all supercomputers, are built from many nodes that work in conjunction with each other.
- Socket: Physical connection for each processor. (VE-V2-343)





- Core: Also called a Compute Core is an independent processing unit. One socket can, and usually does, contain hardware with multiple cores. (VE-V2-344)

Definitions & Theory



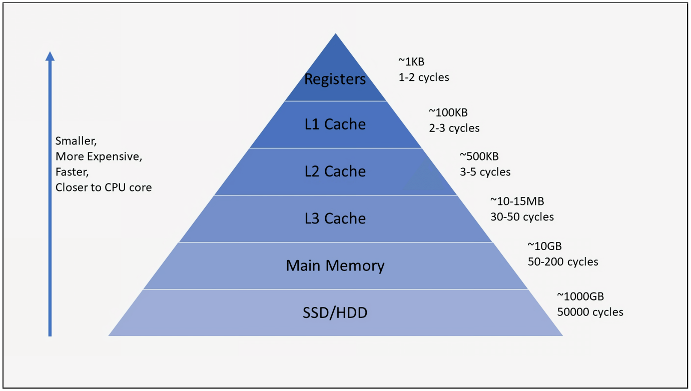
– Hardware Thread: In some architectures each core has multiple hardware threads (hyper-threaded). These can be considered as independent processing units. Many times these will share L1 and L2 cache memory.

Definitions & Theory

- Summary: A Node can have 1 or more sockets (processors, usually 1, 2, or 4), each socket contains a multi-core processor, in some cases each core contains multiple hardware threads (usually 2 or 4).
- Every computer is now a parallel computer so parallel processing techniques are applicable at all levels of computation.

Definitions & Theory

– Cache Memory: Fast memory on the CPU chip. Cache hierarchies and multithreading hardware are examples of ways of mitigating the effects of latency, time to load data and instructions for processing.



Definitions & Theory

- Process: A single computational stream. Can be only one process per core, but a process can contain many threads. When a process contains multiple threads the OS will timeshare the execution of the threads within the process.
- Software Thread: Also called a thread of execution or simply a thread. A thread is an independently schedulable sequence of instructions combined with its private variables and internal control.
  - Usually there are as many threads allocated to the user computation as there are processor cores assigned to the computation. However, this is not required.
  - In some cases oversubscribing will let the OS fill the "gaps" and produce better results.
  - Each thread has its own stack and program counter.
  - There may be system-defined limitations on the number of threads that a program can start. Most systems can start hundreds or thousands of threads.
  - Threads are "lighter weight" then processes, they have less overhead, and can be switched faster. Downside is that the OS has more control over them than you do.

Definitions & Theory

– Speedup: T_1 the execution time on a single processor and T_p the time on p processors. We define the speedup as

$$S_p = \frac{T_1}{T_p}$$

Ideally, $T_p = T_1/p$ and hence $S_p = p$, but in practice $S_p \leq p$. (VE-V1-78)

– Efficiency: Is a measure of how far off we are from an ideal speedup.

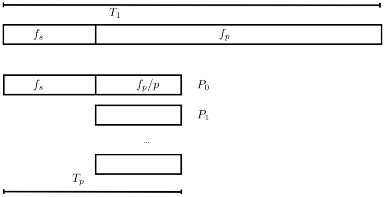
$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$$

Clearly, $0 < E_p \leq 1$. The closer E_p is to 1 the closer our speedup is to ideal.

– Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be **embarrassingly parallel**. Little to no communication between processes is needed, low overhead, $S_p \approx p$ and $E_p \approx 1$.

Definitions & Theory

– Amdahl's Law: Let f_s be the fraction of the problem that is serial and f_p the fraction that can be parallelized. (VE-V1-81)



$$
\begin{aligned}
T_p &= T_1(f_s + f_p/p) \\
S_p &= \frac{T_1}{T_p} = \frac{T_1}{T_1(f_s + f_p/p)} = \frac{1}{f_s + f_p/p} \leq \frac{1}{f_s} \\
E_p &= \frac{S_p}{p} = \frac{1}{pf_s + f_p}
\end{aligned}
$$

Definitions & Theory

– Gustafson's Law: (VE-V1-82)
  - Amdahl's law assumes that there is a fixed computation which gets executed on more and more processors.
  - More realistically, we would assume that there is a sequential fraction independent of the problem size, and a parallel fraction that can be arbitrarily replicated. That is, be able to tailor the size of the problem to the number of available processors.

Let $T_p = T(f_s + f_p)$ with $f_s + f_p = 1$ as before. Then $T_1 = f_s T + p \cdot F_p T$ and we get a speedup of

$$S_p = \frac{T_1}{T_p} = \frac{f_s T + p \cdot f_p T}{T(f_s + f_p)} = \frac{f_s + p \cdot f_p}{f_s + f_p} = f_s + p \cdot f_p = p - (p-1) f_s$$

and efficiency of

$$E_p = \frac{S_p}{p} = \frac{1 + (p-1) f_p}{p} = \frac{1}{p} + \frac{p-1}{p} f_p \to f_p$$

Definitions & Theory

- Scalability (VE-V1-86-87)
  - We say that a problem shows *strong scalability* if, partitioned over more and more processors, it shows perfect or near perfect speedup, that is, the execution time goes down linearly with the number of processors. So if the problem size stays constant and we increase the number of processors then the efficiency will become constant.
  - *Weak scalability* describes the behavior of execution as problem size and number of processors both grow, but in such a way that the amount of "work" per processor stays constant. If this relation is linear, one could state that the amount of data per processor is kept constant, and report that parallel execution time is constant as the number of processors grows.
- Load Balancing (VE-V1-158)
  - Distributing the work evenly among all the processors/threads so that no single processor is overloaded with work and others are idle.
  - One possibility is to move data or work from one process to another, but this can be computationally expensive.
  - Another possibility is to adjust the scheduling of tasks so that an idle process can work on the next computation that is needed instead of waiting. This can be tricky in many situations.

OpenMP

- OpenMP stands for "open multiprocessing"
- OpenMP is an API built on C, not C++, so many of the function expect C style parameters. For example, character arrays for strings. You can use C++ constructs with OpenMP but they may need to be converted to C style constructs when calling OpenMP functions.
- OpenMP Design Goals
  - A shared-memory model allowing direct access to global variables by all threads of a user process, where all concurrent threads run the same code block simultaneously.
  - Create a framework where program could be written both sequentially and parallel. If compiled on a system that does not support OpenMP the sequential version would be compiled and if the support is there the parallel version would be compiled.
  - Make the coding of multi-threaded programs easier (takes care of the pthread issues under the hood). Easier to code but loses some low-level control.

OpenMP

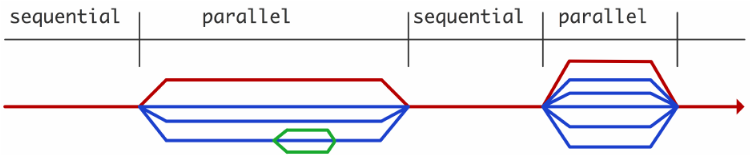
- Produce a system that would allow the programmer to incrementally create a parallel version of a serial program. This is nearly impossible to do with systems like MPI or CUDA where the parallel program needs to be completely redesigned.

  Ian Foster provides an outline of steps in his book Designing and Building Parallel Programs for altering a serial program to a parallel program (Foster's methodology). `https://www.mcs.anl.gov/~itf/dbpp/`

  1. Partitioning. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
  2. Communication. Determine what communication needs to be carried out among the tasks identified in the previous step.
  3. Agglomeration or aggregation. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
  4. Mapping. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

OpenMP

- Fork/Join Model: (VE-V2-345)



- Granularity: The balance between the amount of independent work per processing element, and how often processing elements need to synchronize. We talk of 'large grain parallelism' (Course) if there is a lot of work in between synchronization points, and 'small grain parallelism' (Fine) if that amount of work is small. (VE-V1-98)
- Usually course is better due to fork/join overhead.
- On a shared-memory system communication between threads is mainly through read and write operations to shared variables. Without synchronization, multiple threads accessing a shared-memory location may cause conflicts.

OpenMP

- Critical Section: A critical section is code executed by multiple threads that updates a shared resource, and the shared resource can only be updated by one thread at a time.
- OpenMP allows the compiler and run-time system to determine some of the details of thread behavior, so it can be simpler to code some parallel behaviors using OpenMP. The cost is that some low-level thread interactions can be more difficult to program.

OpenMP

– Examples: HelloOMP and HelloOMPMake
  - Compile and run, several times, nondeterministic.
    ```
    gcc -fopenmp -g -Wall HelloOMP.c
    ```
  - Makefile
  - Trace and syntax. C vs C++.
  - Note number of default threads.
  - Set threads with environment variable,
    ```
    export OMP_NUM_THREADS=8
    ```
    Rerun. Note that you can use a bash script to simplify the setting of environment variables and reduce typing. For example, the script
    ```
    #!/bin/bash

    export OMP_NUM_THREADS=1000
    ./prog
    ```
    Will run prog with 1000 threads.
  - You can start new session to clear environment variable or with the bash script you don't need to worry about resetting.
  - Add `omp_set_num_threads(8);` before parallel region. Rerun.

OpenMP

  - Remove `omp_set_num_threads(8);` edit directive to `#pragma omp parallel num_threads(8) ...` Rerun.
  - Add in shared variable, show race condition. (VE-V1-106)
  - Directive syntax in general. (VE-V2-347)
  - Add `omp_get_num_procs();` Rerun.
  - Add `omp_get_max_threads();` Rerun.
  - Discuss and experiment with variable scope. (VE-V2-349–350)
- Examples: HelloOMPcpp and HelloOMPcppMake
  - Compile and run, several times, nondeterministic, streaming race condition.

    ```
    g++ -fopenmp -g -Wall HelloOMP.c
    ```
  - Makefile
  - Trace and syntax. stringstream addition.
- π Approximation: Midpoint rule from Calculus.

$$\pi = 4 \cdot \tan^{-1}(1) = 4 \int_0^1 \frac{1}{1+x^2}\, dx \approx \frac{4}{n} \sum_{i=0}^{n-1} \frac{1}{1+((i+0.5)/n)^2}$$

OpenMP

- PiApproxSerial: Compile, run, analyze.
- PiApproxParallel001: Compile, run, analyze. Note the critical section (VE-V1-107)
- PiApproxParallel002: Compile, run, analyze. Note lag. (VE-V2-374). Addition of (mutex)
  `#pragma omp critical`
- PiApproxParallel003: Compile, run, analyze. One way to remove the critical section, at the cost of storage. Experiment with thread numbers, find max, system monitor.
- PiApproxParallel004: Compile, run, analyze. Using the reduction method, discuss. (PM-344)
- Timing: Go over syntax on the two methods, `omp_get_wtime()`, experiment, calculate speedup and efficiency.
  - ▹ PiApproxParallel002Chrono
  - ▹ PiApproxParallel003Chrono
  - ▹ PiApproxParallel004Chrono
  - ▹ PiApproxParallel002Wtime
  - ▹ PiApproxParallel003Wtime
  - ▹ PiApproxParallel004Wtime

OpenMP

- Syntax for OpenMP directives.
  `#pragma omp <directive> <clauses> <code block>`
  - Pragmas are typically added to a system to allow behaviors that aren't part of the basic C/C++ specification.
  - This is a single line statement so a line continuation character "/" is needed for multi-line style/readability.
- Reductions syntax:
  `reduction(<operator>: <variable list>)`
  - `<operator>` can be any of `+, *, -, &, |, ^, &&, ||`
  - OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section, and the values stored in the private variables are combined, using the reduction operation, in this critical section. The result is stored in the shared variable from the variable list.
  - The private variables created for a reduction clause are initialized to the identity value for the operator. For example, if the operator is multiplication, the private variables would be initialized to 1 and if the operator is addition, the private variables would be initialized to 0.

OpenMP

- OpenMP requires that the compiler supports the pragma extensions. Most C++ compilers do but in the case where they don't the directives will simply be ignored during compilation. On the other hand, includes and function calls will be compile errors if OpenMP is not supported.
- Error checking can be done with ifdef directives using `_OPENMP` to determine the specification used by compiler. We will assume our compiler can handle OpenMP and will not clutter our code with the checking. (VE-V2-346)

```
#ifdef _OPENMP
    tid = omp_get_thread_num();
#else
    tid = 0;
#endif
```

- Example HelloOMPEC, compile and run
- Join synchronization at the end of the concurrent threads is enforced unless explicitly avoided through added directives for this purpose (nowait). When all the concurrent threads have completed at the join synchronization point, the computation can proceed; in each case by the master thread alone until the next thread fork is encountered.

OpenMP

- Parallel for Loops:
  - OpenMP only parallelizes for loops, neither whiles nor do-whiles.
  - Loop body must be a structured block of code, no breaks or returns.
  - Loop must be in canonical form,

$$
\textbf{for}\left(\begin{array}{lllll}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index-} \\
 & & \texttt{index <= end} & & \texttt{-index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)
$$

    - ▹ The loop variable index must have integer or pointer type.
    - ▹ Values/variables of start, end, and incr must have a compatible type to index.
    - ▹ The expressions start, end, and incr must not change during execution of the loop.
    - ▹ During execution of the loop, the variable index can only be modified by the "increment expression" in the statement.
    - ▹ The loop variable index is made private even if it is declared outside the parallel for structure.
  - The compiler needs to determine the number of iterations at compile time.

OpenMP

- Examples
  - PiApproxParallelFor
  - PiApproxParallelFor2
  - PiApproxParallelFor3
- Data dependence and loop-carried dependence. Occurs when a result of one iteration is used in a subsequent iteration. Hence if the iterations are in an arbitrary order, as is the case in parallelizing loops, the results are unpredictable.
  - OpenMP compilers don't check for dependencies among iterations in a loop that's being parallelized. It's up to the programmer to identify these dependencies.
  - A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP without using features such as tasking.
- Examples
  - FibSerial
  - FibParallel001

OpenMP

- To detect a loop-carried dependence, we should only concern ourselves with variables that are updated by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another.
- Estimating π again .... This time with a series also from Calculus and also from the arc-tangent.

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right) = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

- Examples: Run and analyze.
    - PiSeriesSerial
    - PiSeriesParallel
    - PiSeriesParallel2
    - PiSeriesParallel3

OpenMP

- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a block.
- A variable that can be accessed by all the threads in the team has shared scope, while a variable that can only be accessed by a single thread has private scope.
- Shared variables have the same value in the parallel or parallel for block that they had before the block, and their value after the block is the same as their last value in the block.
- Private variables have a different copy for each thread. Depending on the clause you use, the private variable may be uninitialized or have an initial value for all threads, that of the variable before the parallel block when using the firstprivate clause.
- OpenMP Scoping Rules
  - Variables declared outside a parallel region are shared by all the threads generated by the region.
  - Static variables are shared by all the threads generated by the region.
  - Variables declared inside the structured block are private to each thread.

OpenMP

  - Loop variables in a parallel for are private to each thread, no matter where they were declared.
  - Stack variables in functions called from parallel regions are private to each thread.
  - Reduction variables are private to each thread.

- Explicit Scoping: private, shared, and default.
  - `private(<variable list>)` — Makes the variables in the list private to each thread.
  - `shared(<variable list>)` — Makes the variables in the list shared by all threads.
  - `default(none)` — Forces the programmer to list all variables in the structured block as either private or shared using the above constructs. The default clause can also can take shared and private as parameters.
    - `default(private)` will default each of structured block variables to being private and the programmer can specify the shared variables, `default(private) shared(x, y)`
    - `default(shared)` will default each of structured block variables to being shared and the programmer can specify the private variables, `default(shared) private(x, y)`

OpenMP

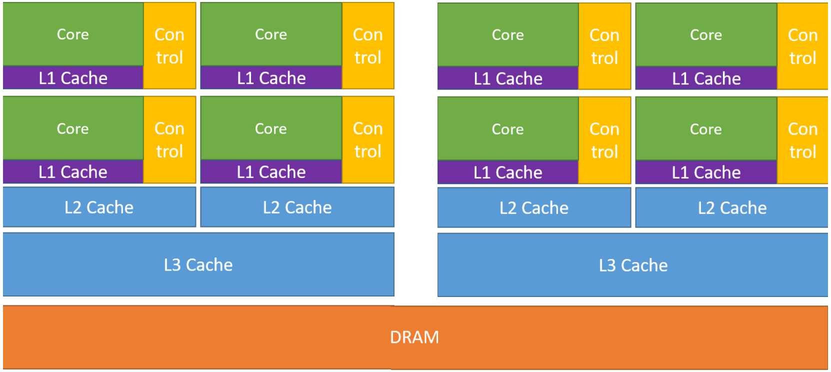
- `firstprivate(<variable list>)` — firstprivate is a variation on private with the value of the variable on each thread initialized by the value of the variable outside the parallel region.
- `lastprivate(<variable list>)` — lastprivate is also a variation on private, must be in a parallel for and not just a parallel construct. The value of the variable is set to the last value updated in the loop.
- For arrays, statically allocated data (int A[10]) can be shared or private, depending on the clause you use. Dynamically allocated data (new and malloc) can only be shared. The firstprivate clause with dynamically allocated data works like shared.
- For vectors, private will crash the program (as with dynamically allocated arrays). The firstprivate clause initializes the vector data and treats it as private.
- Examples
  - ▹ OpenMPScoping
  - ▹ OpenMPArrayScoping
  - ▹ OpenMPArrayScoping2
  - ▹ OpenMPVectorScoping

OpenMP

- Odd-Even Transposition Sort
  - Go over method, and loop-dependencies in bubble-sort. (PM-361)
  - Do the two possible parallel frameworks.
  - Nice example of decreasing efficiency and negative returns due to thread creation overhead.
  - Example: OddEvenSort
- Loop Scheduling
  - Default parallel for thread scheduling is block scheduling. Same as `schedule(static)`
  - Three threads with 12 iterations would be scheduled as
  - schedule clause syntax and types (PM-367). `schedule(type [, chunk size])`
  - Type layout diagrams (PM-369)
  - static examples (PM-370)
  - dynamic and guided (PM-371)
  - What to choose: Experiment and suggestions (PM-375)
  - Examples
    - ▹ SchedulingCyclic
    - ▹ SchedulingRuntime and bash scripts.

OpenMP

- Caches, Cache Coherence, and False Sharing



- Temporal and spatial locality, cache line / cache block. (PM-389)
- Cache coherence problem and cache line invalidation (PM-390)
- Discuss matrix vector multiplication and the serial code.(PM-390)

OpenMP

- Run example MatVecMult on several different size matrices and note timing differences in both serial and parallel versions. Also calculate efficiencies.
- Define cache write-miss and cache read-miss.
  - write-miss occurs when a core tries to update a variable that's not in cache, and it has to access main memory.
  - read-miss occurs when a core tries to read a variable that's not in the cache, and it has to access main memory.
- Run Valgrind cache profiles on the program with different matrix sizes.

  ```
  valgrind --tool=cachegrind ./prog
  ```

  - "8000000 X 8" should have more write-misses mainly due to the size of the y vector.
  - "8 X 8000000" should have more read-misses mainly due to the size of the x vector.
  - "8 X 8000000" should also have the worst efficiency. This is mainly due to y being stored completely in the cache of each thread/processor. When one processor updates y the cache line on all the other processor is invalidated and the cache needs to be refreshed from memory, many more CPU cycles needed in the process.

OpenMP

- False Sharing: Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name false sharing. (PM-394-395)

OpenMP

- OpenMP Tasks
  - Comes in handy when using a parallel for construct would be difficult. For example, dynamic problems, including recursive algorithms, such as graph traversals, or producer-consumer style programs. (PM-396-397)
  - `#pragma omp task`
    A new task is generated by the OpenMP run-time that will be scheduled for execution.
  - Standard structure of task generation, (PM-397). Note the single directive. This will have the program create a single task using one thread, which one is not specified and determined at runtime (non-deterministic). Do not need single directive, each thread can generate a task if that is the algorithm.
  - Thread creation overhead can be substantial and the new task threads may not execute immediately.
  - Examples
    - ▹ FibTask

OpenMP

- Thread-safety
  - A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.
  - Most C/C++ functions are not thread-safe but some have thread-safe versions.
  - Consider the C function strtok (tokenizing function). (PM-401).
  - Examples
    - ▹ Tokenize
    - ▹ Tokenize_r

OpenMP

- Producers and Consumers
  - Section 5.8 of (PM-375-389) gives a good discussion of Producer/Consumer problems and issues involved. I have taken a slightly different approach and created a wrapper class for the STL deque to store queue information.
  - Discuss general setup. Never assume that any container class is thread-safe.
  - Example MessagePassing:
    - ▹ Go over base templated Queue class.
    - ▹ Identify enqueue and dequeue critical sections, and reasons.
    - ▹ Discuss the need for a barrier `#pragma omp barrier` and what it does. Remove it and show the result.
    - ▹ Discuss what `#pragma omp single` does.
  - Example MessagePassingCount:
    - ▹ No change to the templated Queue class.
    - ▹ Discuss the `#pragma omp atomic` directive and syntax. (PM-380)
  - Example MessagePassingLock:
    - ▹ Discuss locks (PM-383-384)
    - ▹ Go over changes to the base templated Queue class.

OpenMP

- ▹ Go over the switch from critical to locks in main, advantages.
- Example MessagePassingLockEnc:
  - ▹ Go over changes to the base templated Queue class, encapsulating the locks.
  - ▹ Go over the changes to the main, pros and cons.

– Three mechanisms for enforcing mutual exclusion: locks, atomic, critical, which to use?
  - atomic is probably the fastest, but least flexible. Also, depending on the implementation, it may be coded (in OpenMP) as an unnamed critical section.
  - Locks and the critical directive, on many systems, run at about the same rate. Also the critical directive is usually easier to code. So the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.
  - Things to watch (PM-386-387)
    - ▹ You should not mix the different types.
    - ▹ There is no guarantee of fairness,

```
while(1) {
    #pragma omp critical
    x = f(y);
}
```

OpenMP

could block a thread forever.
- ▹ Nesting mutual exclusion is almost always a bad thing to do, high probability of deadlock.

- Examples
  - ▹ Deadlock
  - ▹ NamedCritical

– Nested Parallel Regions (VE-V2-354-355)
  - By default, the nested parallel region will have only one thread.
  - To allow nested thread creation.
    - ▹ Use the environment variable OMP_MAX_ACTIVE_LEVELS (default: 1) to set the number of levels of parallel nesting.
    - ▹ There are functions omp_set_max_active_levels and omp_get_max_active_levels.
  - Examples
    - ▹ NestedParallel

OpenMP

– Collapsing Nested Loops
  - In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.
  - By default, `#pragma omp parallel for` parallelizes the outer loop.
  - For perfectly nested loops (with no dependencies) we can collapse the loop levels for usually faster computation. (VE-V2-367-368)

```
#pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = B[i][j] + C[i][j];
```

  - Only works for perfectly nested loops.
  - This directive will parallelize the entire “loop space” instead of just the outer loop.
  - Examples, run timings.
    - ▹ Collapse
    - ▹ Collapse2

OpenMP

– Overriding the Implicit Synchronization
  - Adding the `nowait` clause will release a thread as soon as it is finished with its workload. (VE-V2-370)
  - Be careful with this and possible race conditions. Note from the text example below, there is a data dependency between the two loops so having the same schedule between the loops guarantees that the computations needed for the second loop (thread for thread) will be completed before it is run.

```
#pragma omp parallel
{
    x = local_computation()
#pragma omp for schedule(static) nowait
    for (int i = 0; i < N; i++) {
        x[i] = ...
    }
#pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
        y[i] = ... x[i] ...
    }
}
```

  - Examples, go over the thread work imbalance run timings.
    ▷ nowait

OpenMP

- Sections (VE-V2-383)
  - If you have a pre-determined number of independent work units, the sections construct may be appropriate.
  - In a sections construct can be any number of section constructs.
  - These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.
  - Examples
    - ▹ Sections

OpenMP

– More Advanced Example: N-Body Simulation
  - The *n*-body problem is one of the most famous problems in mathematical physics, with its first complete mathematical formulation dating back to Newton's Principia. Classically, it refers to the problem of predicting the motion of *n* celestial bodies that interact gravitationally. Nowadays, other problems, such as those from molecular dynamics, are also often referred to as *n*-body problems.
  - Given two particles i and j with masses m_i and m_j respectively and positions $\vec{r}_i(t)$ and $\vec{r}_j(t)$ at some time t. Then the force $f_{i,j}(t)$ on i that is exerted by j is given by

$$\vec{f}_{i,j}(t) = -\frac{Gm_i m_j}{\|\vec{r}_i(t) - \vec{r}_j(t)\|^3}(\vec{r}_i(t) - \vec{r}_j(t))$$

Where $G = 6.673 \times 10^{-11} m/(kg \cdot s^2)$ and $\|\vec{v}\| = \sqrt{x^2 + y^2 + z^2}$ is the norm of $\vec{v}$.

OpenMP

- Given n particles that are interacting with each other then the total force that is exerted on particle i by all the other particles is the sum of the forces of each particle exerting on i. Specifically,

$$\begin{aligned}\vec{F}_i(t) &= \sum_{\substack{j=0\\ i\neq j}}^{n-1} \vec{f}_{i,j}(t) \\ &= -Gm_i \sum_{\substack{j=0\\ i\neq j}}^{n-1} \frac{m_j}{\|\vec{r}_i(t)-\vec{r}_j(t)\|^3}(\vec{r}_i(t)-\vec{r}_j(t))\end{aligned}$$

- By Newton's second law of motion, $\vec{F} = m\vec{a}$, hence from the above calculations, and $\vec{a} = \frac{1}{m}\vec{F}$, we can calculate the acceleration that is exerted on each particle at each time step of the simulation and get an updated value for the particle's position and velocity.
- For the simulation we use Euler's approximation method from differential equations.
  - ▹ Start with n particles each with some given mass, initial position, and initial velocity. For our simulation, these will be chosen at random within ranges that are defined by the user of the program, that is, inputs.

OpenMP

- ▹ Calculate the force vector on each particle due to the other $n-1$ particles in the system.
- ▹ Calculate the acceleration vector due to this force.
- ▹ Update each of the particle's position and velocity from the induced acceleration for a given time step Δt. The value of Δt will also be specified by the user.
- ▹ Save the particle information to an array structure.
- ▹ Repeat this process for each time step in the simulation.

- There are several places where parallelization can be applied.
- There are several different methods we can use as well. The one that worked the best for my experiments was the use of a reduction of the inner loop in the total force calculation. This also shows an example of how the programmer can create a user-defined reduction based off an operator overload of a class structure. (VE-V2-377-381)
- A few notes on numerical consistency between the serial and parallel versions.
  - ▹ As has been discussed before, the order of floating point operations is different between the serial and parallel versions of the run. In fact, the order of the parallel version will differ on each run.
  - ▹ Floating point arithmetic on the CPU is neither commutative nor associative, unlike in exact mathematics.

OpenMP

- ▹ If we were doing only one time step the error between the two would be very minimal. On the other hand, when we increase the number of time steps these small round-off errors will accumulate and the particle positions (and velocities) between the two simulations will drift.
- ▹ For example, if there are 1000 particles and we are doing 1000 time steps in the simulation each particle will do the $\vec{f}_{i,j}(t)$ calculations about 1,000,000 times. Small changes in the $\vec{r}$ vectors will alter the norm calculations, which are then cubed and divided into the mass m_j, compounding the discrepancy. Furthermore, since each particle interacts with all other particles in the system these errors are not localized to a portion of the system but diffuse to the entire system.

- Examples
  - ▹ NBodySim
  - ▹ NBodySimPara
  - ▹ PointSimulationViewer: This is a viewer for the simulation data produced by the other two examples here. Read the README file to see what additional libraries you need to compile this program. It is not needed to understand the parallelization of the N-Body Simulation calculations.