Improving OpenMP Performance

Johnnie W. Baker

March 2, 2011

References

- MAIN REFERENCE: Using OpenMP: Portable Shared Memory Parallel Progamming, Barbara Chapman, Gabriele Jost, and Rudd Van Der Pas, The MIT Press, 2008, Chapter 5 “How to Get Good Performance by Using OpenMP”.
- Other references may be added later.

Introduction

- While it is relatively easy to quickly write a correctly functioning program in OpenMP, it is more difficult to write a program with a good performance.
  - When performance is poor, it is usually due to fact that basic programming rules have not been followed.
  - If these rules are followed, a base level of performance is usually achieved.
  - This can often be gradually improved by successive fine tuning various aspects of the program.
- The goal of this set of slides is to help programmers achieve this basic level of performance initially.
- An additional subgoal is to provide programmers with enough insight to improve the code after this basic level of performance is achieved.

Performance Considerations for Sequential Programs

- Good scalar performance is a major concern in OpenMP.
- Poor sequential performance is often caused by suboptimal usage of the cache memory subsystem found in modern computers.
  - A cache-miss at the highest level in the cache hierachy is expensive, as data much be fetched from main memory before it can be used.
    - 5 to 10 times as expensive as fetching data from one of the caches
- On shared memory multiprocessor systems, this inefficiency is magnified.
  - The more threads involved, the larger the potential performance problem.
  - A miss at the highest level causes additional traffic on the system interconnect (i.e., bus, interconnection network, etc.
  - None of the systems on the market have sufficient bandwidth to absorb frequent cache misses.

Sequential Memory Organization

- Sequential memory organized as a hierarchy
  - Organized into pages, with a subset being available to each application.
  - The memory levels closer to the processor are successively smaller and faster and are known as cache.
  - When a program is compiled, the compiler will arrange for its data objects to be stored in main memory.
  - These are transferred to cache, as needed.
  - When a value is needed that is not in cache, a cache-miss occurs and it is retrieved from higher levels of memory
    - This can be quite expensive.
    - Program data is brought into cache in chunks called blocks, each of which will occupy a line of cache.
    - Some data currently in cache may have to be removed to make space for this data

Sequential Memory Organization (cont.)

- The memory hierarchy used is not normally controllable by the user of compiler.
  - Data is fetched into cache and evicted dynamically, according to need.
  - Various strategies have been devised that can help the compiler and programmer reduce cache misses.
  - The goal is to organize data accesses so that values are used as often as possible while they are in cache.
  - Common strategies based on fact that programming languages typically specify that elements of an array be stored contiguously in memory
  - When an array value is fetched into cache, nearby elements will also be fetched and stored at the same time.

Sequential Memory Organization (cont.)

- In C, a 2-D array is stored in rows (and in columns in Fortran)
  - Element [0][0] is followed by [0][1], and then by [0][2].
  - When an array element is transferred to cache, typically the entire row (in C) is transferred to cache.
  - For good performance, matrix-based computation should access the elements of the arrays by rows (and by columns in Fortran)
  - When a row is brought into memory, "Unit Stride" refers to using all of the elements in the line before the next line is referenced.

Translation-Lookaside Buffer

- On a system that supports virtual memory, memory adresses for different applications are given logical addresses arranged into virtual pages.
- The sizes of a virtual page depends on the sizes the systems supports and the choices offered by the operating system.
- A typical page size is 4 or 8 Kbytes, but larger ones exist.
- The physical pages available to the progam may be spread out in memory
  - As a result, the virtual pages must be mapped to the physical ones.
  - The operating system sets up a data structure, called a page table, that records this mapping.

Translation-Lookaside Buffer (cont)

- The Page Table resides in main memory
  - This causes the use of this table to be time-consuming
  - A special cache was developed to hold recently addressed entries in the page table.
    - Called the Translation-Lookaside Buffer or TLB
    - Can considerably improve the performance of the system and applications.
  - It is important to make good use of the TLB
    - It is desirable for pages in the TLB to be heavily referenced.
    - It is also important for program to access data in storage order
      - Otherwise, may have frequent reloads of the TLB and a large number of TLB misses may occur.

Loop Optimization

- By making minor changes in loops, the program or compiler can improve the use of memory.
- For example a i-loop that has an embedded j-loop may calculate the same results more efficiently if the position of the two loops are switched.
  - May result in accessing data in C in rows instead of in columns.
  - Since much of computational time is spent in loops and since most array access occurs there, a suitable reorganization to exploit cache can significantly improve a programs performance.
- Rule for Interchangeability: If any memory location is referenced more than once in the loop nest and if at least one of those references modify its value, then their relative order must not be changed by the transformation.

Loop Optimization (cont)

- A programs code has to be checked carefully to see if a reordering of statements is allowed and whether it is desirable:
  - This is often done better by programmers than by compilers.
  - Should consider a loop transforming if memory accesses to arrays in the loop nest do not occur in the order they are stored in memory.
  - Also consider loop transforming if loop has a large body and references to an array element or its neighbors are far apart.
  - A simple reordering within the loop may make a difference.
  - Reordering the loop may also allow better exploitation of parallelism or to better utilize the instruction pipeline.
  - They can also be used to increase the size of the parallel area.

Loop Unrolling

- A loop unrolling transformation packs all of the work of several loop iterations into a single pass through the loop.
  - A powerful technique to effectively reduce the overheads of loops
  - May not reduce to one single pass, but reduce the number of passes through the loop (say by a factor of two by performing two loop calculations on each pass through "reduced loop").
    - In example, two is called the "unroll factor".
  - Eliminates number of increment of the loop variable, test for completion, and branches to the start of the loop code.
  - Helps improve cache line utilization by improving data reuse.
  - Can also increase the instruction level parallelism (ILP)
  - See Pg 130 of primary reference book for examples (Chapman, et.al._)

Loop Unrolling (cont)

- Compilers are good at doing loop unrolling.
- One problem is that if the unroll factor does not divide the iteration count, the remaining iterations have to be performed outside of the loop nest
  - Implemented through a second "cleanup" loop.
- If loop already contains a lot of computation, loop unrolling may the cache less efficient.
- If loop contains a procedure call, unrolling the loop results in new overheads that may outweigh the benefits.
- If loop contains branches, the benefits may also be low.
- Loop jamming is similar to loop unrolling and consists of "jamming the body of two inner loops" into a single inner loop that performs the work of both.

Loop Fusion and Loop Fission

- Loop Fusion merges two or more loops to create a bigger loop.
  - May enable data in cache to be reused more frequently
  - Might increase the amount of computation per iteration in order to improve the instruction level parallelism.
  - Would probably also reduce loop overhead because more work is done per loop.
- Loop fission is a transformation that breaks up a loop into several loops.
  - May be useful in improving the use of cache or to isolate a part that inhibits the full optimization of the loop.
  - Most useful when loop nest is large and its data does not fit well into cache or if we can optimize different parts of the loop in different ways.

Loop Tiling or Blocking

- Transformation designed to tailor the number of memory references inside a loop so they can fit within cache.
  - If data sizes are large and memory access is bad or if there is little data reuse in the loop, then chopping the loop into chunks (tiles) may be helpful.
  - Loop tiling replaces the original loop with a couple of loops.]
  - This can be done for as many loops inside the loop nest as needed.
  - May have to experiment with the size of the tiles
  - See main reference for more details (pg 135-6)

Use of Pointers and Contiguuous Memory in C

- While pointers are frequently used in C, they pose a serious challenge for performance tuning.
- Must usually assume that pointers can access any memory location.
  - Called the pointer aliasing problem
  - Prevents the compiler from performing many program optimizations, since cannot determine which are safe.
    - Will cause performance to suffer
  - If pointers can be guaranteed to point to portions of non-overlapping memory due to distinct calls to "malloc" function, more aggressive optimization techniques can be used.
- See Section 5.2.4 of main reference for other techniques.

Using Compiler Features to Improve Performance

Most of the loop optimization techniques presented earlier in this chapter can be implemented using most modern compilers.

- Compilers use a variety of analysis to determine which of techniques can be used.
- Once the correctness of a numerical result is assured, experiment with compiler options to squeeze out an improved performance.
- These options (or flags) differ a great deal from one compiler to another.
- A compiler's ability to transform code is limited by its ability to analyze the program and decide what can safely be modified.
- A different problem arises when the compiler is not able to improve memory usage because that involves changing the structure of nonlocal data.
  - If programmer does some rewriting of source code, some better results may be obtained.

Timing the OpenMP Performance

- A standard practice is to use a standard operating system command.
- The /bin/time command is available on standard UNIX systems. For example

  /bin/time .a.out

  – The "real", "user", and "system" times are then printed after the program has finished execution.
  – For example

    ```
    $ /bin/time .program.exe
    real 5.4
    user 3.2
    sys 1.0
    ```

  – These three numbers can be used to get initial information about the performance.
  – For deeper analysis, a more sophisticated performance tool is needed.

Timing the OpenMP Performance (cont)

- Generally, the sum of the user and system time is referred to as the CPU time.
- The number following "real" tells us that the program took 5.4 seconds from the beginning to the end. The real time is also called the wall-clock time or elapsed time
- The user time of 3.2 seconds is the time spent outside any operating system service.
- The sys time is the time spent on operating system services such as input/output routines
- A common cause for the difference between the wall-clock time of 5.4 seconds and the CPU time is a processor sharing too high a load on the system.
- The **omp_get_wtime()** function provided by OpenMP is useful for measuring the elapsed time of blocks of source code.
- When getting timing, you need to check to see if you are the only person on the system. If your times are varying, the reason may be that you are sharing the processors with one or more other users.

Timing the OpenMP Performance (cont)

- If sufficient processors are available (i.e., not being used by other users), your elapsed time should be less than the CPU time.
- Recall, a parallel program has additional overheads, collectively called the parallel overhead. This includes the time to create, start, and stop threads, the extra work to figure out what each task is to perform, the time spent waiting for barriers and at critical sections and locks, and the time spent computing the same operations redundantly.
- The parallel speedup is calculated by taking the by taking the ratio of the elapsed time on P processors and the elapsed time on the serial version.
  - It is implicitly assumed that all processors requested are available throughout the execution of the program.
  - Amdahl's law and the fact the fraction f of the execution that must be performed sequentially is greater than zero explains why the speedup is normally less than optimal.

Overview of Parallel Overheads

- The manner in which the memory is accessed by individual threads has a major influence on performance
  - If each thread accesses a distinct portion of data consistently through the program, the threads will probably make excellent use of memory.
  - This improvement includes good use of thread-local cache.
  - Increased performance can often be obtained by increasing the fraction of data references that can be handled in cache.
- The fraction of the work that is sequential or replicated.
  - Replicated work refers to computations that occur once in the sequential program but are performed by each thread in the parallel version.
  - Trade-off between cutting communication costs and using processors to do new work.
- The cost for time spent handling OpenMP constructs.
  - Each of the directives and routines in OpenMP comes with some overhead.
  - For example, when a parallel section is created, threads may have to be created or woken up

Overview of Parallel Overheads (cont)

- Some data structures often have to be created to store the information
- The work that has to be performed by each tread normally has to be determined at run-time.
- Known as the (OpenMP) parallelization overheads, since this work is not done on sequential solutions.

- The load inbalance between synchronization points.
  - If the threads perform differing amounts of work in the work-shared region, the faster threads have to wait at the barrier for the slower ones to reach that point.
  - When treads are inactive a barrier, they are not contributing to getting the work done.
  - If operating system uses one thread to carry out an operation, this can lead to a load inbalance.
  - These are called the load inbalance overheads

Overview of Parallel Overheads (cont)

- Other synchronization costs.
  - Threads typically waste time waiting for access to critical regions or a variable involved in an atomic update, or to acquire a lock
  - If threads are unable to perform useful work during the time they are waiting, they remain idle.
  - Called the synchronization overheads.
- Parallel Benefits
  - One positive effects of parallel computation is that there will be more aggregate cache capacity available since each thread has some local cache.
  - Likewise, there will be more memory available for parallel computations.
  - In some cases, this can result in superlinear speedup.

OpenMP Translation Overheads

- The overhead experienced by creation of parallel regions, sharing work between threads, and all kind of synchronization are a result of the OpenMP used.
- The efficiency of a particular version of OpenMP depend upon the implementation of its compiler and the library and their efficiencies, the target platform, etc.
- The EPCC microbenchmarks were created to help programmers estimate the relative cost of using different OpenMP constructs.
  - They are easy to use and provide an estimate of the overhead required for each feature/ construct used.
  - SPINX is another set of microbenchmarks
  - Information about major OpenMP constructs are given in Figure 5.18 of principal reference.
  - Information about overheads for different kinds of loops is given in Figure 5.19

Best Practices
for Writing Efficient Programs

- In this topic, we will try to provide some general recommendations on how to write efficient OpenMP code.
- Optimize Barrier Use
  - Even efficiently implemented barriers are expensive.
  - The nowait clause can be used to eliminate the barrier that results from several constructs, including the loop construct.
  - Must be careful to do this safely. Especially consider the order of different reads and writes from same portion of memory.
  - A recommended strategy is to first ensure that the OpenMP program works correctly and then avoid the **nowait** clause where possible by carefully inserting explicit barriers at points in the program where needed.
  - See examples in primary reference text for this chapter.

Avoid the Ordered Construct

- The **ordered** construct ensures that the corresponding blocks of code are executed in the order of the loop iterations.
- This construct is expensive to implement.
  - Run time system has to keep track of which iterations have finished and possibly keep threads in wait state until their result is needed!
  - This almost always slows the program down.
- The ordered construct can often (& perhaps always) be avoided. It may be better to wait and perform I/O outside of the parallelized loop.
- These alternate solutions may not be easy to implement.

Avoid Large Critical Regions

- A critical region is used to ensure that no two threads execute a piece of code simultaneously.
- Can be used when the actual order of which threads perform computation is not important.
- However, the more code in the critical region, the greater the time that threads needing this CR will have to wait.
- Reducing size of CR will require re-writing sections of code.
- An example is given in the primary reference text for this set of slides. (E.g., see page 147-8).

Maximize Parallel Regions

- Indiscriminate use of parallel regions may lead to suboptimal performance.
- There are significant overheads associated with starting and terminating parallel regions.
- Large parallel regions offer more opportunities for using data in cache and provide a context for other compiler optimizations.
- If multiple parallel loops exist, we must choose whether to enclose each loop in an individual parallel region or create on region encompassing all of them.
- See example in reference text – page 147-149.

Avoid Parallel Regions in Inner Loop

- Another common technique to improve the performance is to move parallel regions out of the innermost loops.
- Otherwise, we repeatedly incur the overheads of the parallel constuct.
- In reference text, an example on pg 150 is given where the overhead for the **parallel for** construct is incurred n^2 times.
- By moving the parallel construct outside of the loop nest, the parallel construct overheads are minimized.

Improve unbalanced Thread Loads

- In some parallel algorithms, there is a wide variation in the amount of work threads have to do.
- When this happens, threads wait at the next synchronization point until the slowest thread arrives.
- One way to avoid this problem is to use the **schedule** clause with a non-static schedule.
- The problem is that the **dynamic** and **guided** workload distribution schedules have higher overheads than **static**.
- However, if the load balance is severe enough, this cost is offset by the more flexible allocation of work to the threads.
- It is a good idea to experiment with these schemes, as well as with various values for the chuck
- Here, the **runtime** clause comes in handy, as it allows easy experimentation without the need to recompile the program.

Overlapping Computation and I/O

- An example of this is given in primary text on pg 151-152.
  - This is under the topic of addressing poor load balance.
- This helps avoid having all but one processors wait while the I/O is handled.
- In general, when possible, a general rule for MIMD parallelism in general is to overlap computation and communications so that the total time taken is less that the sum of the times to do each of these.
- However, when using OpenMP in a data parallel fashion, this general guideline might not always be possible.

Single Construct versus Master Construct

- The functionality of the single and master constructs are similar
- The difference is that the single construct can be executed by any thread while the master construct can only be executed by the master thread.
- The single construct also has an implied barrier, but can be overridden using a *nowait* clause
- It would appear that the master construct would usually be the more efficient unless another thread would usually arrive at the construct first.
- However, which is the more efficient depends on implementation and applications details.
- See pg 153 of primary reference for more details.

Avoid False Sharing

- An important efficiency concern is “false sharing”. It can also severely restrict scalability.
- It is a side effect of the cache-line granularity of cache coherence implemented in shared memory systems.
  - See Section 1.2 on Pg 3 of main reference.
- The cache coherency implementation keeps track of the status of cache lines by appending “state bits” to indicate whether the data on the cache line is still valid or is outdated.
- Any time a cache line is modified, cache coherence software notifies other caches holding a copy of the same line that its line is invalid.
- If data from that line is needed, a new updated copy must be fetched.

Avoid False Sharing (cont)

- A problem is that state bits do not keep track of which part of the line is outdated, but indicates the whole line is outdated.
- As a result, a processor can not detect which part of the line is still valid and instead requests a new copy of the entire line
- As a result, when two threads update different data elements in the same cache line, they interfere with each other.
- Suppose all threads contain a copy of the vector a with eight elements in it and thread 0 modifies a[0].
- Everything in the cache line with a[0] is invalidated, so a[1], …, a[7] are not accessible until the cache is updated in all thread, even though their data has not changed and is valid
- So threads i can not access a[i] for i>0 until each of their cache lines containing a[0], … , a[7] have been updated.

Avoid False Sharing (cont)

- In this case, we can solve the problem by *padding* the array by dimensioning it as a[n] [8] and changing the indexing from a[i] to a[i] [0].
- Access to different elements a[i] [0] are now separated by a cache line and updating an array element a[i] [0] does not invalidate a[j] [0] for $i \neq j$.
- Although this array padding works well, it is a low-level solution that depends on the size of the cache line and may not be portable.
- In a more efficient implementation, the variable a is declared and used as a private variable for each thread instead of having thread i to access the global array position a[i].

Avoid False Sharing (cont)

- False sharing is likely to significantly impact performance under the following condition
  a. Shared data is modified by multiple threads
  b. The access pattern is such that multiple threads modify the same cache line(s).
  c. These modifications occur in rapid succession
- All of these conditions needs to be met for the degradation to be noticeable.
- There is no false sharing on read-only data, as lines are not invalidated.

Avoid False Sharing (cont)

- In general, using private data instead of shared data significantly reduces the risk of false sharing.
  - Unlike padding, this is a portable solution.
  - One exception to being portable is when different private copies are held in the same cache line
  - A second exception is where the end of one copy shares a cache line with the start of another.
  - Although above exceptions may occur occasionally, the performance impact is not likely to be significant.

Private versus Shared Data

- The programmer may choose whether data should be shared or private.
- Either choice may lead to the correct solution, but performance can be seriously impacted if the wrong choice is made.
- If threads need read/write access to a 1-D array, a 2-D shared array can be declared so each thread could access one row.
- Alternately, each thread could allocate a 1-D private array within the parallel region.
- In general, the latter approach is preferred over the former.
  - In the first case, false sharing may occur if modifications are frequent and a data element for one thread is in the same cache line as a data element modified by another thread.
  - There is much less chance of interference with private data.

Private versus Shared Data (cont)

  - Accessing shared data also requires dereferencing a pointer, which incurs a performance overhead.

- If data is read but not written in a parallel region, it could be shared, with each thread having read-access to it.
- It could also be privatized, so that each thread has a local copy of it, using the **firstprivate** clause.
- In above cases, the first solution of sharing the data seems to be the best choice.