The OpenMP Programming Model

CS 8320

J. Walter Larson
Mathematics and Computer Science Division
Argonne National Laboratory
and
Computation Institute
University of Chicago
and
Department of Computer Science
The Australian National University
Over the Next Two Hours...

- Shared-memory platforms, their multicore cousins, and SMP programming
- A (quick?) review of OpenMP
  - history/philosophy
  - elements
    - syntax
      - concurrency directives
      - synchronisation directives
      - data handling / scoping clauses
    - library
    - environment variables
- Advanced OpenMP
- Performance
Shared Memory Platforms--The Original Motivation for OpenMP

- Beginning in the 1980s, numerous hardware vendors began offering systems with multiple processors sharing a single memory bank
- Access was via a shared bus
- These systems were called shared memory parallel systems
- Exploiting this shared-memory parallelism required use of vendor-specific programming models
  - Performance portability was what programmers wanted
  - Needing to deal with vendor-specific cruft was what they got
- By the early 90’s, beginnings of an SMP programming standard were underway
- The 1996 SGI/Cray merger accelerated the process

**Diagram:**

- P0
- PI
- M
- Shared Bus to Memory
What is OpenMP?

- OpenMP := Open specifications for Multi Processing
- An API for implementing explicit shared-memory parallelism
  - Annotation of serial source code
    - compiler directives
    - run time library calls
    - environment variables
  - Provides an incremental / evolutionary path for developing parallel code from serial source
  - Higher-level approach than Pthreads
  - API defined for C/C++ and Fortran
    - directives implemented as CPP #pragma for C/C++
    - directives implemented as comments for Fortran
  - Portable (numerous compiler vendors support it)
  - de facto standard for this style of parallel programming

Grew out of vendor-specific shared-memory parallel code annotation approaches
- First OpenMP standard released in 1997
- OpenMP 3.0 standard finalised in 2008
- Web site http://openmp.org

Thus, same-source parallel code can be maintained
The Structure of OpenMP

Image Source: John Mellor-Crummey’s Lecture Notes
Multicore Platforms Have Renewed Interest in OpenMP

- As we will see, OpenMP has a programming model well-suited to incremental introduction of parallelism into legacy codes.
- Much of the parallel programming community (scientists, as it turns out) embraced OpenMP initially.
- To some extent, the message-passing parallelism model (e.g., MPI) was more successful due to promises of greater scalability on distributed-memory platforms.
- Interest in OpenMP was renewed on clusters that incorporated distributed shared-memory nodes.
- Interest has become much stronger with the advent of multicore architectures due to their similarity to SMPs.
A Quick and Easy Example
void MatrixVectorProduct(int nRows, int nCols, double * restrict a, 
                      double * restrict b, double * restrict c)
{
    int i, j;

    for (i=0; i<nRows; i++)
    {
        a[i] = 0.0;
        for (j=0; j<nCols; j++)
            a[i] += b[i*nCols+j]*c[j];
    }
}
A Very Simple Example -- Parallelisation of Matrix-Vector Multiplication using OpenMP

void MatrixVectorProduct(int nRows, int nCols, double * restrict a, double * restrict b, double * restrict c)
{
    int i, j;

    #pragma omp parallel for default(none) \
    shared(nRows,nCols,a,b,c) private(i,j)
    for (i=0; i<nRows; i++)
    {
        a[i] = 0.0;
        for (j=0; j<nCols; j++)
            a[i] += b[i*nCols+j]*c[j];
    }
}
What Just Happened? The Fork-Join Model

- OpenMP’s style of parallelism is called fork-join
- When execution of a parallel region begins, the single process thread *forks*, spawning a *team of threads*
  - The original (parent) process is thread 0, or the *master thread*
  - Work is parceled out to the team of threads by the underlying system in the OpenMP implementation
  - At the end of the parallel regions, a *join* occurs, after which only the master thread continues execution

*Image Source: John Mellor-Crummey’s Lecture Notes*
Some Terminology...

- **OpenMP Directive**: `#pragma omp`...Something that specifies program behaviour in OpenMP
- **Executable Directive**: An OpenMP directive that is not declarative; in short, all OpenMP directives except `threadprivate`
- **Construct**: An OpenMP executable directive and associated statement, loop, structured block, or program section to which it applies
  - OpenMP parallel construct defines a parallel code region
  - OpenMP work sharing construct defines work break-down structure for team of threads
- **Clauses**: Modifiers that specify how OpenMP should behave in a construct regarding data sharing, execution, scheduling, and synchronisation
OpenMP Directives and Constructs
The omp parallel Construct

- A program that does not have at least one of these is—from an OpenMP viewpoint—a serial program
  - Even if compiled with OpenMP enabled, it will execute sequentially
- Clauses allow further specificity for parallelisation
  - if(scalar-expression)
  - num_threads(integer-expression)
  - reduction(operator:list-of-variables)
  - Other clauses control data scoping/sharing and will be discussed shortly

C/C++
```c
#pragma omp parallel [optional clauses...] structured block...
```

Fortran
```fortran
!$omp parallel [optional clauses...] structured block...
!$omp end parallel
```
OpenMP Directives for Work Sharing

- Distribute iterations across a team of threads
  ```c
  #pragma omp for [optional clauses...] 
  C/C++ for-loop
  ```
  Loop must conform to
  ```c
  for (init-expr; bounds-expr ; incr-expr)
  ```

- Functional decomposition of work units across a team of threads
  ```c
  #pragma omp sections [optional clauses...] 
  {
  #pragma omp section 
  : 
  #pragma omp section 
  : 
  }
  ```
  Each section must be independent
  Each thread executes a section at a time
  Each section is executed only once
  If there are more sections than threads, some threads will execute multiple blocks

- Only one code executes a particular block
  ```c
  #pragma omp single [optional clauses...] 
  code block...
  ```
OpenMP Clauses to Control Data Sharing

- **private** *(list-of-variables)*: Values of listed variables specific to a given thread, and only known by it. Values undefined upon entry to parallel construct and after exiting it.

- **shared** *(list-of-variables)*: Values of listed variables are shared among thread team members. Values defined before fork as well as after parallel region.

- **firstprivate** *(list-of-variables)*: Values of listed variables are identical to value of variables of the same name immediately before entry into parallel region.

- **lastprivate** *(list-of-variables)*: “Last” value among threads is available upon exit of parallel region
  - In work shared loop, value from final iteration that would be last were the loop executed sequentially
  - In a sections construct, value associated with final operation involving the variable in the lexically last section

- **default** *(data-sharing-attribute)*: Default property for variables occurring in a parallel region. In C/C++ API, supported values of *data-sharing-attribute* are *shared* and *none*. If invoked, only exceptions need be listed in other clauses.
OpenMP shared & private Example

```c
void MatrixVectorProduct(int nRows, int nCols, double * restrict a,
                        double * restrict b, double * restrict c)
{
    int i, j;

#pragma omp parallel for default(none) \ 
    shared(nRows,nCols,a,b,c) private(i,j)
    for (i=0; i<nRows; i++)
    {
        a[i] = 0.0;
        for (j=0; j<nCols; j++)
            a[i] += b[i*nCols+j]*c[j];
    }
}
```

• Shared by default:
  • Matrix and vector dimensions
  • Matrix and vector element values

• Private:
  • Loop indices
OpenMP firstprivate Example

- Assign array element values
- Static workload distribution (chunk of n entries per thread)
- Works from global offset of 4 elements
- Each thread determines its own start/stop indices

```c
iOffset = 4;
#pragma omp parallel default(none) firstprivate(iOffset) \ 
    private(i, TID) shared(n,a)
{
    TID = omp_get_thread_num();
    iOffset += n*TID
    for(i=iOffset; i<iOffset+n; i++)
        a[i] = TID + 1
}
```

OpenMP library routine that returns thread ID number
void MatrixVectorProduct(int nRows, int nCols, double * restrict a, double * restrict b, double * restrict c)
{
    int i, j;

#pragma omp parallel for default(shared) private(i,j)
    for (i=0; i<nRows; i++)
    {
        a[i] = 0.0;
        for (j=0; j<nCols; j++)
            a[i] += b[i*nCols+j]*c[j];
    }
}
The `nowait` Clause

- In OpenMP, there is an implicit synchronisation at the end of each work sharing construct.
- For performance tuning, it may be possible to reduce the number of synch points in a code (one should always make sure this will not effect code correctness!)
- OpenMP offers the `nowait` clause to suppress synchronisation at the end of a work sharing construct.

```plaintext
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<iMax; i++)
  {
    ...........
  }

  ...........
}
```

- One cannot, however, suppress synchronisation (the join) that occurs at the end of an OpenMP parallel region.

Threads do not synch at end of loop

Threads do synch here!
The schedule Clause

- **Syntax:** `schedule(kind[,chunk-size])`

- **Granularity** expressed in *chunks*, and chunk sizes can be set. Applicable to loop constructs only. When no chunk size is specified, the default size of one iteration is used.

- **Kind** attribute controls manner in which iterations are assigned to threads
  - **static:** Interations chunked by set size, or total number divided so that each thread gets roughly the same number of iterations; doled out round-robin. Lowest associated overhead. Good for predictable loads.
  - **dynamic:** Chunks of iterations assigned as the threads request work; threads work until work on current chunk is completed and no further chunks are available for assignment. Good for unpredictable/poorly balanced loads.
  - **guided:** Like dynamic, but
    - For `chunk-size=1`, chunks are sized in proportion to the number of remaining iterations, divided by the number of threads
    - For `chunk-size=k>1`, chunks are sized to contain no fewer than k iterations, the exception being the final chunk, which may be smaller
  - **runtime:** Schedule kind and chunk size controlled by OpenMP environment variables (`OMP_SCHEDULE`)

- Only the **static** schedule is deterministic, and thus reproducible
Synchronisation Constructs

- Obviously, when memory is shared and multiple actors read/write it, there is potential for mischief.
- OpenMP offers a number of synchronisation constructs for controlling threads’ access to shared data:
  - **barrier**: no thread proceeds past a barrier until all threads in its current team reach the barrier point.
  - **ordered**: iterations in a parallel loop are forced to execute in sequential order.
    - **Printing output**
    - **Debugging**
  - **critical**: Mechanism to ensure multiple threads do not attempt to update same bit of shared data simultaneously.
  - **atomic**: Forces assignment immediately following it to take place atomically; that is, one thread assigns at a given time when there are.
  - **Locks**: Accessed via OpenMP library; low-level, and like semaphores.
  - **master**: Force execution of associated code to take place on master thread only.
The barrier Construct

- Syntax: `#pragma omp barrier`
- Barriers must be inserted so that all or no threads encounter them (otherwise the code will hang)
- The sequence of work-sharing regions and barrier regions must be the same for every thread
- Further restriction for C/C++ OpenMP API: A barrier construct may only be inserted at places in the program where its absence would still yield correct syntax
The ordered Construct

- Syntax: `#pragma omp ordered`
- As previously mentioned, forces sequential ordering of operations within a parallel loop
- Applicable only to loop constructs
- Accepts no further arguments/parameters
- Code outside this block runs in parallel if this construct is part of a larger parallel region
- How it works:
  - Upon entry, the thread executing the first iteration of a loop proceeds without waiting
  - Threads executing later iterations, wait until all iterations previous to their respective workloads have executed, at which time execution of a given thread’s workload takes place
#pragma omp parallel for default(none) ordered schedule(runtime) \
private(i,TID) shared(n,a,b)
for (i=0; i<n; i++)
{
    TID = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n",TID,i);
    a[i] += i;
    #pragma omp ordered
    printf("Thread %d prints the value of a[%d] = %d\n",TID,i,a[i])
}

Thread 0 updates a[3]
Thread 2 updates a[0]
Thread 2 prints the value of a[0] = 0
Thread 3 updates a[2]
Thread 1 updates a[1]
Thread 1 prints the value of a[1] = 2
Thread 3 prints the value of a[2] = 4
Thread 0 prints the value of a[3] = 6
The critical Construct

- Syntax: `#pragma omp critical [name]`
- Ensures multiple threads do not attempt to write the same shared data simultaneously--i.e., avoidance of a race condition
- Beware--if a critical construct is assigned a name, this name is global and thus should be unique to the block in question
- Note: example below is illustrative, not optimal;-) OpenMP reduction operation a better choice

```c
sum = 0;
#pragma omp parallel shared(n,a,sum) private(TID,sumLocal)
{
    TID = omp_get_thread_num();
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)
        sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
        sum += sumLocal;
    }
}
```
The atomic Construct

- Syntax: `#pragma omp atomic`
- Applies solely to assignment statement *immediately following it*
- Protects updates of just one memory location--*the left hand side of the assignment statement* to which it applies
- Example: Multiple threads updating a counter

```
count = 0;
#pragma omp parallel shared(n,count) private(i)
  for (i=0; i<n; i++)
    {
      #pragma omp atomic
      count = count + 1;
    }
```

- Another (scary but instructive) example

```
tally = 0;
#pragma omp parallel shared(n,tally) private(i)
  for (i=0; i<n; i++)
    {
      #pragma omp atomic
      tally = tally + myFunc();
    }
```

Note: `atomic` does not prevent other threads from evaluating the function `myFunc()` simultaneously!

Preventing such a situation would require use of OpenMP `critical` instead.
**Locks**

- For even greater flexibility than *atomic* or *critical* constructs, there exists a set of low-level, general-purpose runtime library of locking routines
- Similar to semaphores
- Syntax: `void omp_function_lock(omp_lock_t *lck)`
- Here, *function* is one of `{init, set, test, unset, destroy}`
- Two types of locks
  - simple locks - may not be locked if already in locked state
    - *special type omp_lock_t in C/C++*
  - nestable locks - may be locked multiple times by the same thread
    - *special type omp_nest_lock_t in C/C++*
- Note: These constitute powerful medicine and should be used with care.
  - Misuse can lead to code hangs or hard-to-trace bugs
The master Construct

- Syntax: `#pragma omp master`
- Restricts code block to which it applies to run only on the master thread (thread 0)
- Similar to `single` construct, but restricted to thread 0
  - Thus more deterministic in this sense
- Note: This construct does not have an implied barrier on entry/exit
  - For example, if the master thread is to initialise data needed for a later portion of a parallel region, the user must guarantee no other threads enter this later region until the master initialises their data
**Other OpenMP Clauses**

- **if**(condition): Modifies only the OpenMP parallel construct
  - The attribute condition is a scalar logical expression
    - if true the block is executed in parallel by a team of threads
    - if false the block is executed sequentially by a single thread

- **num_threads**(int): Applicable only to OpenMP parallel construct
  - sets number of threads in team for parallel region

- **reduction**: More optimal alternative to critical construct for performing global reductions that involve associative and commutative operations

- **copyin**(list): Enables copying of threadprivate variables from the master thread into threadprivate variables for a non-master thread

- **copyprivate**(list): broadcast from single thread to other threads of private variables in list
  - Applicable only to single directive
  - Implied barrier inherent in operation; cannot be used with nowait clause
More About reduction

- Syntax: `reduction(operator:list)`
  - `operator`: operation to which the reduction will apply
  - `list`: List of affected variables
  - Supported operations in C/C++
    - Arithmetic operations `{+, *, -}`
    - Bitwise operations `{&, |, ^}`
    - Logical operations `{&&, ||}`

- Restrictions
  - Aggregate types not supported
  - Reduction variable must not be a `const`
  - Operators specified cannot be overloaded w.r.t. the variables that appear in the list
### Which Clauses Apply to Which Directives?

<table>
<thead>
<tr>
<th>Clause</th>
<th>PARALLEL</th>
<th>DO/for</th>
<th>SECTIONS</th>
<th>SINGLE</th>
<th>PARALLEL DO/for</th>
<th>PARALLEL SECTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHEDULE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenMP Library and Environment Variables
How to Query/Manipulate the OpenMP Environment

- OpenMP has five internal control variables:
  - `nthreads-var`: Stores the number of threads requested for future parallel regions
  - `dyn-var`: Controls whether dynamic adjustment of the number of threads to be used in future parallel regions is enabled
  - `nest-var`: Controls whether nested parallelism is enabled
  - `run-sched-var`: Stores scheduling information for loop regions using runtime scheduling
  - `def-sched-var`: Stores implementation-defined default scheduling information for loop regions

- OpenMP has four environment variables that can also control execution
  - `OMP_NUM_THREADS`: This environment variable will be used to initialise `nthreads-var`
  - `OMP_DYNAMIC`: Setting this variable to value `true` allows the OpenMP implementation to adjust the number of threads to use in parallel regions
  - `OMP_NESTED`: Setting this variable to value `true` allows nested parallelism (assuming the implementation supports it). Default value is `false`
  - `OMP_SCHEDULE`: Sets default scheduling mechanism and chunk size
Runtime Library

- Imported into C/C++ codes by `#include <omp.h>`
- Commonly encountered functions are:
  - `omp_get_num_threads()`: Returns number of threads in current region
  - `omp_set_num_threads(int nThreads)`: Sets number of threads
  - `omp_get_max_threads()`: Returns maximum number of threads available for next parallel region
  - `omp_get_dynamic()`: Returns value of `dyn-var` at runtime
  - `omp_set_dynamic(scalar-integer-expression)`: Modifies value of `dyn-var` at runtime
  - `omp_get_nested()`: Returns indicator of whether nested parallelism is enabled
  - `omp_set_nested(scalar-integer-expression)`: Turns on/off nested parallelism
  - `omp_get_thread_num()`: Returns local thread ID number
  - `omp_get_num_procs()`: Returns total number of processors available
  - `omp_get_in_parallel()`: Returns indication of whether a parallel region is active
- Others? Locks (discussed previously), timers (to be discussed later)
Advanced Features
Nested Parallelism

- Some OpenMP implementations support nested parallelism
  - A thread within a team of threads may fork spawning a child team of threads

Image Source: John Mellor-Crummey’s Lecture Notes
### The flush Directive

- Modern SMPs have complex memory hierarchies (i.e., multiple levels of caches)
- Thus, even for OpenMP shared variables, each thread may have its own specific (and thus not globally consistent!) value of a shared variable
  - shared variable values are reckoned at implementation-specific synchronisation points (user typically unaware of how/where)
- OpenMP offers a mechanism for forcing a thread’s current view of a shared entity consistent with the value in memory--the **flush** directive
  - Syntax: `#pragma omp flush [(list)]`
    - *List of variables optional*
      - In its absence all thread-visible shared variables are flushed
- Potential gotchas:
  - flush affects only the local thread, it does not synch up values for all threads
  - optimisations performed by compilers can move around ordering of operations, but will retain ordering of flushing w.r.t. operations involving flushed variables
- Implicit flush operations occur at
  - All barriers
  - Entry/exit of all **critical** regions
  - Entry/exit of OpenMP’s lock routines
The threadprivate Directive

- Global variables (static variables in C/C++; common blocks in Fortran) are considered “shared,” but in an OpenMP context this may be inappropriate.
- Suppose instead, we want to have data that is private to a thread, but persistent.
- This is precisely what the threadprivate directive does!
- Syntax: `#pragma omp threadprivate (list)`
  - For each variable in the list, a copy of the variable for each thread is maintained.

Threadprivate must be used with caution. The following conditions must be met in order for threadprivate data to be valid:

- The parallel regions must be executed using the same number of threads.
  - `specific threadprivate data will then be handled by the same thread ID as before`
- Neither parallel region is nested inside another parallel region.
- Dynamic scheduling is not used in either region (`dyn-var = false`).
- The value of `nthreads-var` is unmodified throughout and between the two parallel regions.
A Pointer-Intensive Example...

```c
int * pglobal

int main()
{
    .................

    for (i=0; i<n; i++)
    {
        if ((pglobal=(int *)malloc(length[i]*sizeof(int))) != NULL ) {

            for (j=sum=0; j<length[i]; j++) pglobal[j] = j+1;
            sum = calculate_sum(length[i]);
            printf("Value of sum for i = %d is %d\n",i,sum);
            free(pglobal);
        } else {
            printf("Fatal error in malloc - length[%d] = %d\n",i,length[i]);
        }
    }

    .................
```
int * pglobal

int main()
{
    ................
    for (i=0; i<n; i++)
    {
        if ((pglobal=(int *)malloc(length[i]*sizeof(int))) != NULL ) {

            for (j=sum=0; j<length[i]; j++) pglobal[j] = j+1;
            sum = calculate_sum(length[i]);
            printf("Value of sum for i = %d is %d\n",i,sum);
            free(pglobal);
        }
        else {
            printf("Fatal error in malloc - length[%d] = %d\n",i,length[i]);
        }
    }
    ................

    Could parallelise using #pragma omp parallel
    ...except pointer pglobal is shared by default.
    Operations on pglobal constitute race conditions
Usage of threadprivate

```c
int * pglobal

#pragma omp threadprivate(pglobal)

int main()
{
    .................

    #pragma omp parallel for shared(n,length,check) private(TID,i,j,sum)
    for (i=0; i<n; i++)
    {
        TID = omp_get_thread_num();

        if ((pglobal=(int *)malloc(length[i]*sizeof(int))) != NULL ) {

            for (j=sum=0; j<length[i]; j++) pglobal[j] = j+1;
            sum = calculate_sum(length[i]);
            printf("TID %d: Value of sum for i = %d is %d\n",TID,i,sum);
            free(pglobal);
        } else {
            printf("TID %d: Fatal error in malloc - length[%d] = %d\n",TID,i,length[i]);
        }
    }
    .................
```

N.B.: Because we used threadprivate, we never had to look at details of the function calculate_sum()
OpenMP Tasks

- New to OpenMP 3.0
- Introduced to address a weakness of previous versions of the OpenMP standard: user control of load placement/balance in complex situations
- Tasks support parallelisation of irregular problems
- The `task` construct defines an explicit task, and is modified by the following clauses:
  - previously defined `{if, default, private, firstprivate, shared}
  - `untied`: Controls binding of task to threads
Performance
**Basic Performance Tuning Strategies**

- OpenMP offers simplicity, but its simplicity comes at the cost of overheads in creating/initialising thread teams
  - Make parallel regions as large as possible
  - Make parallel regions for compute-intensive portions of the code where OpenMP is likely to pay off
  - Enclose multiple loops in a parallel region
  - Where possible, fuse independent parallel loops with similar indices/bounds
  - In nested loops, parallelise at the outermost loop level

- Help the compiler! Use `#pragma omp parallel for` rather than successive `#pragma omp parallel` followed by `#pragma omp for`

- Minimise the use of synchronisation points such as `barrier`, `critical` sections, and `ordered` regions

- Also, where it does not lead to errors, utilise the `nowait` clause
Importance of Cache Optimisation in OpenMP

- Data stored in cache can be accessed 5-10x faster than a trip out to memory
- This trip out to memory is not only slow, it will also waste system resources and interfere with other data traffic, slowing other threads
- Thus, memory access patterns are important

```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum += a[i][j];
```

Good Memory Access
Loop and array indices are consistent and cache line reuse will occur

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        sum += a[i][j];
```

Bad Memory Access
Loop and array indices are reversed; cache thrashing will result
On systems with virtual memory, addresses are virtualised; that is given logical addresses and arranged in virtual pages.

Physical pages available to a program may be spread throughout the system’s memory.

Thus a mapping between physical and virtual pages—a page table—is necessary.

The page table resides in main memory, and lookups are costly.

- Some systems implement a transition lookaside buffer (TLB), which is a special cache that stores recently accessed page table entries.
- Thus, when a given page table entry is in the TLB, it is best to reuse it as much as possible.
- If the page table entry requested is not in the TLB, a TLB miss occurs.
Loop Transformations

Loop transformations may be performed if the following criterion is met:
- If any memory location is referenced more than once in the loop nest and if at least one of those references modifies its value, then their relative ordering must not be changed by the transformation.

Common loop transformations:
- Loop unrolling
  - Reduces overheads due to loop execution (incrementing index, testing for completion...)
  - Can boost cache line reuse and improve instruction level parallelism
- Unroll and jam
  - Applicable to loop nests with multiple loops
- Loop interchange
  - Changing ordering of loops (see previous cache opt. example)
- Loop fusion
  - Merging of two or more loops
- Loop fission
  - breaking a loop into two or more loops
- Loop tiling/blocking
  - tailoring of memory references in a loop iteration so they will fit in cache
False Sharing

- A consequence of cache-line granularity of cache-coherence in shared memory systems
  - When a thread modifies a variable in cache, cache coherence notifies other caches holding that particular cache line
    - other caches invalidate the cache line in question
    - if the data is still required by the application, a fresh version of the cache line is retrieved from main memory

- Naturally, this is not done at the byte/variable level of granularity, and frequently cache coherence will force dumping of data that would have been perfectly valid for what a particular thread of an application wanted

- The remedy is often array padding, literally spacing out the data in an array so that false associations are less likely to occur
OpenMP Lab on Nested Parallelism and Tasks
Nested Parallelism
**Nested Parallelism**

- Some OpenMP implementations support nested parallelism
  - A thread within a team of threads may fork spawning a child team of threads

*Image Source: John Mellor-Crummey’s Lecture Notes*
More about Nested Parallelism

- It is possible to enable/disable nested parallelism via OpenMP’s internal control variable `nest-var`
  - set either the `OMP_NESTED` environment variable or invoke the library routine `omp_set_nested()`
  - If nested parallelism is enabled, when a thread on a team within a parallel region encounters a new parallel construct, an additional team of threads is forked off of it, and it becomes their master
  - If nested parallelism is disabled, when a thread on a team within a parallel region encounters a new parallel construct, execution continues on this additional single thread only
- The Sun OpenMP implementation provides additional environment variables for controlling nested parallelism:
  - `SUNW_MP_MAX_POOL_THREADS`: Sets limit on the number of threads in the pool that OpenMP maintains for use as slave threads in parallel regions. The default value is 1023.
  - `SUNW_MP_MAX_NESTED_LEVELS`: Sets limit on the number of levels of nesting of parallel regions. Default value is undefined.
Behaviour of OpenMP Library Functions in Nested Parallel Regions

- In particular, consider the following OpenMP library calls
  - `omp_set_num_threads()`
  - `omp_get_max_threads()`
  - `omp_set_dynamic()`
  - `omp_get_dynamic()`
  - `omp_set_nested()`
  - `omp_get_nested()`

- **set** methods affect only parallel regions at same or inner nesting levels encountered by the calling thread
- **get** methods return values set by the calling thread; upon creation of a team of threads, slave threads inherit values from the master thread.
**Tips on Using Nested Parallelism**

- Obviously, nesting parallel regions is an immediate way to engage more threads in a computation.
- Beware! Nesting parallel regions easily can create large number of threads as their number is the product of the number of threads forked at each level of nested parallelism. This can easily oversubscribe the system. Impose some discipline by setting appropriate values of `SUNW_MP_MAX_POOL_THREADS` and `SUNW_MP_MAX_NESTED_LEVELS`.
- Creating any parallel region will entail some overhead. Overhead from nesting of parallel regions may incur overheads greater than necessary if, for example, an outer region could simply employ more threads in a computation. Examples of good reasons to employ nested parallelism are:
  - Insufficient parallelism at outer level
  - Load balance problems
  - Part of a larger parallel region where other code blocks are not sufficiently parallel to warrant more threads.
OpenMP Tasks

- New to OpenMP 3.0
- Introduced to widen the applicability of the OpenMP programming model beyond loop-level and functional parallelism
  - Unbounded loops
  - Recursion
  - Producer/consumer
  - List and tree traversal
- Tasks support parallelisation of irregular problems
- Tasks are work units whose execution may be deferred or performed immediately
- Syntax: `#pragma omp task [clauses...]` above a structured block
- The `task` construct defines an explicit task, and is modified by the following clauses:
  - previously defined `{if, default, private, firstprivate, shared}`
  - untied: Controls binding of task to threads
- Tasks may be nested inside parallel regions, other tasks, and other work sharing constructs


**Elements of a Task**

- Three key elements
  - Code to execute
  - Data (it literally owns its data)
  - An assigned thread that executes the code on the task’s data

- Task processing
  - Each thread encountering a thread construct packages a new instance of a task
  - Some thread in the team executes the task at some later time

- Some other terminology:
  - Task construct: the `task` directive plus the structured block
  - Task: the code plus instructions for allocating data created when a thread encounters a task construct
  - Task region: the dynamic sequence of instructions produced by execution of a task by a thread
Example - Parallel Pointer Chasing Using Tasks

```c
#pragma omp parallel
{
    #pragma omp single private(p)
    {
        p = listhead ;
        while (p) {
            #pragma omp task
            process (p)
            p = next (p) ;
        }
    }
}
```

*spawn to call process(p)*

*implicit taskwait synch*

*point here*
Task switching

- In the code example below, tasks are generated, and lots of them!
- Eventually, the generating task is suspended and executing thread shifts to performing long task that was previously created.
- Other threads work through previously generated tasks and start to starve for work.
- With thread switching, the generating task resumes and makes new work.

```c
#pragma omp single
{
    #pragma omp task untied
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task
}
```