Parallel Processing with OpenMP

Doug Sondak
Boston University
Scientific Computing and Visualization
Office of Information Technology
sondak@bu.edu
Outline

• Introduction
• Basics
• Data Dependencies
• A Few More Basics
• Caveats & Compilation
• Coarse-Grained Parallelization
• Thread Control Directives
• Some Additional Functions
• Some Additional Clauses
• Nested Parallelism
• Locks
Introduction
Introduction

• Types of parallel machines
  – distributed memory
    • each processor has its own memory address space
    • variable values are independent
      \[ x = 2 \] on one processor, \[ x = 3 \] on a different processor
    • examples: linux clusters, Blue Gene/L
  – shared memory
    • also called Symmetric Multiprocessing (SMP)
    • single address space for all processors
      – If one processor sets \[ x = 2 \], \[ x \] will also equal 2 on other processors (unless specified otherwise)
    • examples: IBM p-series, multi-core PC
Shared vs. Distributed Memory

- **Shared Memory**:
  - CPUs (CPU 0, CPU 1, CPU 2, CPU 3) have direct access to a single memory (MEM), indicating shared memory.

- **Distributed Memory**:
  - CPUs (CPU 0, CPU 1, CPU 2, CPU 3) have separate memories (MEM 0, MEM 1, MEM 2, MEM 3), indicating distributed memory.
Shared vs. Distributed Memory (cont’d)

- **Multiple processes**
  - Each processor (typically) performs independent task with its own memory address space

- **Multiple threads**
  - A process spawns additional tasks (threads) with same memory address space
What is OpenMP?

• Application Programming Interface (API) for *multi-threaded* parallelization consisting of
  – Source code directives
  – Functions
  – Environment variables
What is OpenMP? (cont’d)

• Advantages
  – Easy to use
  – Incremental parallelization
  – Flexible
    • Loop-level or coarse-grain
  – Portable
    • Since there’s a standard, will work on any SMP machine

• Disadvantage
  – Shared-memory systems only
Basics
Basics

- Goal – distribute work among threads
- Two methods will be discussed here
  - Loop-level
    - Specified loops are parallelized
    - This is approach taken by automatic parallelization tools
  - Parallel regions
    - Sometimes called “coarse-grained”
    - Don’t know good term; good way to start argument with semantically precise people
    - Usually used in message-passing (MPI)
Basics (cont’d)

Loop-level

Parallel regions

serial

loop

serial

loop

serial
parallel do & parallel for

- parallel do (Fortran) and parallel for (C) directives parallelize subsequent loop

Use “c$” for fixed-format Fortran

\[
\begin{align*}
!$omp parallel do \\
do & i = 1, maxi \\
& \quad a(i) = b(i) + c(i) \\
enddo
\end{align*}
\]

\[
\begin{align*}
#pragma omp parallel for \\
for & (i = 1; i <= maxi; i++) \{ \\
& \quad a[i] = b[i] = c[i]; \\
& \}
\end{align*}
\]
parallel do & parallel for (cont’d)

• Suppose maxi = 1000

  Thread 0 gets i = 1 to 250
  Thread 1 gets i = 251 to 500
  etc.

• Barrier implied at end of loop
workshare

• For Fortran 90/95 array syntax, the parallel workshare directive is analogous to parallel do

• Previous example would be:

```fortran
!$omp parallel workshare
a = b + c
!$omp end parallel workshare
```

• Also works for forall and where statements
Shared vs. Private

• In parallel region, default behavior is that all variables are *shared* except loop index
  – All threads read and write the same memory location for each variable
  – This is ok if threads are accessing different elements of an array
  – Problem if threads write same scalar or array element
  – Loop index is *private*, so each thread has its own copy
• Here’s an example where a shared variable, \texttt{i2}, could cause a problem

```plaintext
ifirst = 10
do i = 1, imax
   i2 = 2*i
   j(i) = ifirst + i2
enddo
```

```plaintext
ifirst = 10;
for(i = 1; i <= imax; i++){
   i2 = 2*i;
   j[i] = ifirst + i2;
}
```
Shared vs. Private (3)

• OpenMP *clauses* modify the behavior of *directives*

• *Private* clause creates separate memory location for specified variable for each thread
\begin{verbatim}
ifirst = 10
!$omp parallel do private(i2)
do i = 1, imax
    i2 = 2*i
    j(i) = ifirst + i2
enddo
\end{verbatim}

\begin{verbatim}
ifirst = 10;
#pragma omp parallel for private(i2)
for(i = 1; i <= imax; i++){
    i2 = 2*i;
    j[i] = ifirst + i2;
}
\end{verbatim}
• Look at memory just before and after parallel do/parallel for statement
  – Let’s look at two threads

**before**

- `ifirst = 10`
- `⋯`
- `i2_{thread_0} = ???

**after**

- `ifirst = 10`
- `⋯`
- `i2_{thread_0} = ???
- `i2_{thread_1} = ???

spawn additional thread
Shared vs. Private (6)

- Both threads access the same shared value of `ifirst`
- They have their own private copies of `i2`
Data Dependencies
Data Dependencies

- Data on one thread can be dependent on data on another thread
- This can result in wrong answers
  - thread 0 may require a variable that is calculated on thread 1
  - answer depends on timing – When thread 0 does the calculation, has thread 1 calculated it’s value yet?
Data Dependencies (cont’d)

• Example – Fibonacci Sequence
  0, 1, 1, 2, 3, 5, 8, 13, ...

```plaintext
  a(1) = 0
  a(2) = 1
do i = 3, 100
    a(i) = a(i-1) + a(i-2)
endo
```

```plaintext
  a[1] = 0;
a[2] = 1;
for(i = 3; i <= 100; i++){
    a[i] = a[i-1] + a[i-2];
}
```
Data Dependencies (3)

• parallelize on 2 threads
  – thread 0 gets i = 3 to 51
  – thread 1 gets i = 52 to 100
  – look carefully at calculation for i = 52 on thread 1
    • what will be values of for i -1 and i - 2 ?
Data Dependencies (4)

• A test for dependency:
  – if serial loop is executed in reverse order, will it give the same result?
  – if so, it’s ok
  – you can test this on your serial code
Data Dependencies (5)

- What about subprogram calls?

```plaintext
do i = 1, 100
  call mycalc(i,x,y)
enddo
```

- Does the subprogram write x or y to memory?
  - If so, they need to be private
- Variables local to subprogram are local to each thread
- Be careful with global variables and common blocks

```plaintext
for(i = 0; i < 100; i++){
  mycalc(i,x,y);
}
```
A Few More Basics
More clauses

- Can make private default rather than shared
  - Fortran only
  - handy if most of the variables are private
  - can use continuation characters for long lines

```fortran
ifirst = 10
$omp parallel do     &
$omp default(private) &
$omp shared(ifirst,imax,j)
do i = 1, imax
  i2 = 2*i
  j(i) = ifirst + i2
enddo
```
More clauses (cont’d)

• Can use **default none**
  – declare all variables (except loop variables) as shared or private
  – If you don’t declare any variables, you get a handy list of all variables in loop
More clauses (3)

```c
ifirst = 10
 !$omp parallel do &
 !$omp default(none) &
 !$omp shared(ifirst,imax,j) &
 !$omp private(i2)
 do i = 1, imax
   i2 = 2*i
   j[i] = ifirst + i2
 enddo
```

```c
ifirst = 10;
#pragma omp parallel for \
  default(none) \
  shared(ifirst,imax,j) \
  private(i2)
 for(i = 0; i < imax; i++){
   i2 = 2*i;
   j[i] = ifirst + i2;
 }
```
Suppose we need a running index total for each index value on each thread

\[
\begin{align*}
\text{iper} &= 0 \\
\text{do } i &= 1, \text{imax} \\
&\quad \text{iper} = \text{iper} + 1 \\
&\quad \text{j}(i) = \text{iper} \\
\text{enddo}
\end{align*}
\]

- if \textit{iper} were declared \texttt{private}, the initial value would not be carried into the loop
Firstprivate (cont’d)

• Solution – firstprivate clause
• Creates private memory location for each thread
• Copies value from master thread (thread 0) to each memory location

iper = 0
!$omp parallel do &
!$omp firstprivate(iper)
do i = 1, imax
  iper = iper + 1
  j(i) = iper
enddo

iper = 0;
#pragma omp parallel for 
  firstprivate(iper)
for(i = 0; i < imax; i++){
  iper = iper + 1;
  j[i] = iper;
}
Lastprivate

- saves value corresponding to the last loop index
  - "last" in the serial sense

```c
!$omp parallel do lastprivate(i)
do i = 1, maxi-1
   a(i) = b(i)
enddo
a(i) = b(1)
```

```c
#pragma omp parallel for \ lastprivate(i)
for(i = 0; i < maxi-1; i++){
   a[i] = b[i];
}
a(i) = b(1);
```
Reduction

- following example won’t parallelize correctly with parallel do/parallel for
  - different threads may try to write to sum1 simultaneously

```c
sum1 = 0.0
for(i = 0; i < imaxi; i++){
    sum1 = sum1 + a[i];
}
```
Reduction (cont’d)

• Solution? – Reduction clause

```c
sum1 = 0.0
!$omp parallel do &
!$reduction(+:sum1)
do i = 1, maxi
   sum1 = sum1 + a(i)
enddo
```

```c
sum1 = 0;
#pragma omp parallel for 
   reduction(+:sum1)
for(i = 0; i < imaxi; i++){
   sum1 = sum1 + a[i];
}
```

• each thread performs its own reduction (sum, in this case)

• results from all threads are automatically reduced (summed) at the end of the loop
Reduction (3)

- Fortran operators/intrinsics: MAX, MIN, IAND, IOR, IEOR, +, *, -, .AND., .OR., .EQV., .NEQV.
- C operators: +, *, -, /, &, ^, |, &&, ||
- Roundoff error may be different than serial case
Ordered

• Suppose you want to write values in a loop:

```fortran
do i = 1, nproc
    call do_lots_of_work(result(i))
    write(21,101) i, result(i)
enddo
```

```c
for(i = 0; i < nproc; i++){
    do_lots_of_work(result[i]);
    fprintf(fid,"%d %f\n","i,result[i]");
}
```

• If loop were parallelized, could write out of order
• **ordered** directive forces serial order
!$omp parallel do
do i = 1, nproc
   call do_lots_of_work(result(i))
!$omp ordered
   write(21,101) i, result(i)
!$omp end ordered
enddo

#pragma omp parallel for
for(i = 0; i < nproc; i++){
do_lots_of_work(result[i]);
#pragma omp ordered
fprintf(fid,"%d %f\n,"i,result[i]);
#pragma omp end ordered
}

Ordered (3)

• since do_lots_of_work takes a lot of time, most parallel benefit will be realized

• ordered is helpful for debugging
  – Is result same with and without ordered directive?
• Number of threads for subsequent loop can be specified with `num_threads(n)` clause
  – $n$ is number of threads
Caveats and Compilation
Caveats

• OpenMP will do what you tell it to do
  – If you try parallelize a loop with a dependency, it will go ahead and do it!

• Do not parallelize small loops
  – Overhead will be greater than speedup
  – How small is “small”?
    • Answer depends on processor speed and other system-dependent parameters
    • Try it!
Compile and Run

- Portland Group compilers (katana):
  - pgf95, pgcc, etc.
  - Compile with `-mp` flag
  - Can use up to 8 threads, depending on node

- AIX compilers (twister, etc.):
  - Use an `_r` suffix on the compiler name, e.g., `cc_r`, `xlf90_r`
  - Compile with `-qsmp=omp` flag

- Intel compilers (skate, cootie):
  - Compile with `-openmp` and `–fpp` flags
  - Can only use 2 threads
Compile and Run (cont’d)

- GNU compilers (all machines):
  - Compile with `-fopenmp` flag
- Blue Gene does not accommodate OpenMP
- environment variable `OMP_NUM_THREADS` sets number of threads
  
  `setenv OMP_NUM_THREADS 4`

- for C, include `omp.h`
Conditional Compilation

- Fortran: if compiled without OpenMP, directives are treated as comments
  - Great for portability
- !$ (c$ for fixed format) can be used for conditional compilation for any source lines
  
  !$ print*, 'number of procs =', nprocs

- C or C++: conditional compilation can be performed with the _OPENMP macro name.

  #ifdef _OPENMP
  ...
  ... do stuff ...
  #endif
Basic OpenMP Functions

- `omp_get_thread_num()`
  - returns ID of current thread
- `omp_set_num_threads(nthreads)`
  - subroutine in Fortran
  - sets number of threads in next parallel region to `nthreads`
  - alternative to `OMP_NUM_THREADS` environment variable
  - overrides `OMP_NUM_THREADS`
- `omp_get_num_threads()`
  - returns number of threads in current parallel region
• **parallel** and **do/for** can be separated into two directives.

```
!$omp parallel do
do i = 1, maxi
    a(i) = b(i)
enddo

#pragma omp parallel for
for(i=0; i<maxi; i++){
    a[i] = b[i];
}
```

is the same as

```
!$omp parallel

!$omp do
do i = 1, maxi
    a(i) = b(i)
enddo

!$omp end parallel

#pragma omp parallel
#pragma omp for
for(i=0; i<maxi; i++){
    a[i] = b[i];
}
#pragma omp end parallel
```
Parallel (cont’d)

• Note that an end parallel directive is required.
• Everything within the parallel region will be run in parallel (surprise!).
• The do/for directive indicates that the loop indices will be distributed among threads rather than duplicating every index on every thread.
Parallel (3)

- Multiple loops in parallel region:

```c
#pragma omp parallel
#pragma omp for
for(i=0; i<maxi; i++)
    a[i] = b[i];
}
#pragma omp for
for(i=0; i<maxi; i++)
    c[i] = a[2];
}
#pragma omp end parallel
```

- `parallel` directive has a significant overhead associated with it.
- The above example has the potential to be faster than using two `parallel do/parallel for` directives.
Coarse-Grained

- OpenMP is not restricted to loop-level (fine-grained) parallelism.
- The !$omp parallel or #pragma omp parallel directive duplicates subsequent code on all threads until a !$omp end parallel or #pragma omp end parallel directive is encountered.
- Allows parallelization similar to “MPI paradigm.”
Coarse-Grained (cont’d)

```c
!$omp parallel &
!$omp private(myid,istart,iend,nthreads,nper)
nthreads = omp_get_num_threads()
nper = imax/nthreads
myid = omp_get_thread_num()
istart = myid*nper + 1
iend = istart + nper – 1
call do_work(istart,iend)
do i = istart, iend
    a(i) = b(i)*c(i) + ...
enddo
!$omp end parallel
```

```c
#pragma omp parallel \\    #pragma omp private(myid,istart,iend,nthreads,nper) 
 #pragma omp private(myid,istart,iend,nthreads,nper)
nthreads = OMP_GET_NUM_THREADS();
nper = imax/nthreads;
myid = OMP_GET_THREAD_NUM();
istart = myid*nper;
iend = istart + nper - 1;
do_work(istart,iend);
for(i=istart; i<=iend; i++){
    a[i] = b[i]*c[i] + ...
}
#pragma omp end parallel
```
Thread Control Directives
• barrier synchronizes threads

$$\texttt{#omp parallel private}(\texttt{myid}, \texttt{istart}, \texttt{iend})$$
$$\texttt{call myrange}(\texttt{myid}, \texttt{istart}, \texttt{iend})$$
$$\texttt{do i = istart, iend}$$
  $$\texttt{a(i) = a(i) - b(i)}$$
$$\texttt{enddo}$$
$$\texttt{!omp barrier}$$
$$\texttt{myval}(\texttt{myid+1}) = a(\texttt{istart}) + a(1)$$

$$\texttt{#pragma omp parallel private}(\texttt{myid}, \texttt{istart}, \texttt{iend})$$
$$\texttt{myrange}(\texttt{myid}, \texttt{istart}, \texttt{iend})$$;
$$\texttt{for}(i=\texttt{istart}; i<=\texttt{iend}; i++){\texttt{\{}$$
  $$\texttt{a[i] = a[i] - b[i];}$$
$$\texttt{\}}$$
$$\texttt{#pragma omp barrier}$$
$$\texttt{myval[myid] = a[istart] + a[0]}$$

• Here barrier assures that a(1) or a[0] is available before computing myval
Master

- if you want part of code to be executed only on master thread, use `master` directive
- `master example`
- “non-master” threads will skip over `master` region and continue
Single

- want part of code to be executed only by a single thread
- don’t care whether or not it’s the master thread
- use single directive
- single example
- Unlike the end master directive, end single implies a barrier.
Critical

- have section of code that:
  1. must be executed by every thread
  2. threads may execute in any order
  3. threads must not execute simultaneously
- use critical directive
- critical example
- no implied barrier
Sections

- In parallel region, have several independent tasks
- Divide code into sections
- Each section is executed on a different thread.
- **section example**
- Implied barrier
- Fixes number of threads
- Note that sections directive delineates region of code which includes section directives.
Parallel Sections

- Analogous to do and parallel do or for and parallel for, along with sections there exists a parallel sections directive.
- parallel_sections_example
Task

- Parallel sections are established upon *compilation*
  - Number of threads is fixed
- Sometimes more flexibility is needed, such as parallelism within *if* or *while* block
- Task directive will assign tasks to threads as needed
- [Task example](from OpenMP standard 3.0)
Threadprivate

- Only needed for Fortran
  - Usually legacy code
- common block variables are inherently shared, may want them to be private
- threadprivate gives each thread its own private copy of variables in specified common block.
  - Keep memory use in mind
- Place threadprivate after common block declaration.
common /work1/ work(1000)
 !$omp threadprivate(/work1/)
Copyin

- `threadprivate` renders all values in common block private
- may want to use some current values of variables on all threads
- `copyin` initializes each thread’s copy of specified common-block variables with values from the master thread
- `copyin example`
Atomic

• Similar to critical
  – Allows greater optimization than critical
• Applies to single line following the atomic directive
• atomic example
Flush

- Different threads can have different values for the same *shared* variable
  - example – value in register
- **flush** assures that the calling thread has a consistent view of memory
- **flush example** (from OpenMP spec)
Some Additional Functions
Dynamic Thread Assignment

- *On some systems* the number of threads available to you may be adjusted dynamically.
- The number of threads requested can be construed as the *maximum* number of threads available.
- Dynamic threading can be turned on or off:

```c
logical mydyn = .false.
call omp_set_dynamic(mydyn)
```

```c
int mydyn = 0;
omp_set_dynamic(mydyn);
```
Dynamic Thread Assignment (cont’d)

- Dynamic threading can also be turned on or off by setting the environment variable `OMP_DYNAMIC` to `true` or `false`
  - call to `omp_set_dynamic` overrides the environment variable

- The function `omp_get_dynamic()` returns a value of “true” or “false,” indicating whether dynamic threading is turned on or off.
OMP_GET_MAX_THREADS

- integer function
- returns maximum number of threads available in current parallel region
- same result as `omp_get_num_threads` if dynamic threading is turned off
OMP_GET_NUM_PROCS

• integer function
• returns maximum number of processors in the system
  – indicates amount of hardware, not number of available processors
• could be used to make sure enough processors are available for specified number of sections
OMP_IN_PARALLEL

• logical function
• tells whether or not function was called from a parallel region
• useful debugging device when using “orphaned” directives
  – example of orphaned directive: `omp parallel in “main,” omp do or omp for in routine called from main`
Some Additional Clauses
Collapsing loops can be used to improve parallelism in OpenMP. The `collapse` clause allows parallelism over multiple nested loops.

```plaintext
!$omp parallel do collapse(2)
  do j = 1, nval
    do i = 1, nval
      ival(i,j) = i + j
    enddo
  enddo
enddo
```
Schedule

- **schedule** refers to the way in which loop indices are distributed among threads

- default is **static**, which we had seen in an earlier example
  - each thread is assigned a contiguous range of indices in order of thread number
    - called **round robin**
  - number of indices assigned to each thread is as equal as possible
Schedule (cont’d)

- **static example**
  - loop runs from 1 to 51, 4 threads

<table>
<thead>
<tr>
<th>thread</th>
<th>indices</th>
<th>no. indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-13</td>
<td>13</td>
</tr>
<tr>
<td>1</td>
<td>14-26</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>27-39</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>40-51</td>
<td>12</td>
</tr>
</tbody>
</table>
Schedule (3)

- number of indices doled out at a time to each thread is called the *chunk size*
- can be modified with the `SCHEDULE` clause

```c
$omp do schedule(static,5)
#pragma omp for schedule(static,5)
```

```c
#pragma omp for schedule(static,5)
```
Schedule (4)

- example – same as previous example (loop runs from 1-51, 4 threads), but set chunk size to 5

<table>
<thead>
<tr>
<th>thread</th>
<th>chunk 1 indices</th>
<th>chunk 2 indices</th>
<th>chunk 3 indices</th>
<th>no. indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-5</td>
<td>21-25</td>
<td>41-45</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>6-10</td>
<td>26-30</td>
<td>46-50</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>11-15</td>
<td>31-35</td>
<td>51</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>16-20</td>
<td>36-40</td>
<td>-</td>
<td>10</td>
</tr>
</tbody>
</table>
Dynamic Schedule

- `schedule(dynamic)` clause assigns chunks to threads dynamically as the threads become available for more work
- default chunk size is 1
- higher overhead than STATIC

```c
!$omp do schedule(dynamic,5)

#pragma omp for
schedule(dynamic,5)
```
Guided Schedule

• `schedule(guided)` clause assigns chunks automatically, exponentially decreasing chunk size with each assignment

• specified chunk size is the *minimum* chunk size except for the last chunk, which can be of any size

• default chunk size is 1
Runtime Schedule

• schedule can be specified through omp_schedule environment variable

setenv OMP_SCHEDULE “dynamic,5”

• the schedule(runtime) clause tells it to set the schedule using the environment variable

!$omp do schedule(runtime)

#pragma omp for schedule(runtime)
Nested Parallelism
Nested Parallelism

- parallel construct within parallel construct:

```plaintext
!$omp parallel do
do j = 1, jmax
  !$omp parallel do
    do i = 1, i,max
      call do_work(i,j)
    enddo
  enddo
enddo
```

```plaintext
#pragma omp parallel for
for(j=0; j<jmax; j++){
  #pragma omp parallel for
    for(i=0; i<imax; i++){
      do_work(i,j);
    }
}
```
Nested Parallelism (cont’d)

- OpenMP standard allows but does not require nested parallelism.
- If nested parallelism is not implemented, previous examples are legal, but the inner loop will be serial.
- Logical function `omp_get_nested()` returns `.true.` or `.false.` (1 or 0) to indicate whether or not nested parallelism is enabled in the current region.
Nested Parallelism (3)

- `omp_set_nested(nest)` enables/disables nested parallelism if argument is true/false
  - subroutine in Fortran (.true./.false.)
  - function in C (1/0)
- nested parallelism can also be turned on or off by setting the environment variable `omp_nested` to true or false
  - calls to `omp_set_nested` override the environment variable
Locks
Locks

• **Locks** offer additional control of threads
• a thread can take/release ownership of a lock over a specified region of code
• when a lock is owned by a thread, other threads cannot execute the locked region
• can be used to perform useful work by one or more threads while another thread is engaged in a serial task
Lock Routines

- Lock routines each have a single argument
  - argument type is an integer (Fortran) or a pointer to an integer (C) long enough to hold an address
  - OpenMP pre-defines required types

```plaintext
integer(omp_lock_kind) :: mylock
omp_nest_lock_t *mylock;
```
Lock Routines (cont’d)

• `omp_init_lock(mylock)`
  – initializes `mylock`
  – must be called before `mylock` is used
  – subroutine in Fortran

• `omp_set_lock(mylock)`
  – gives ownership of `mylock` to calling thread
  – other threads cannot execute code following call until `mylock` is released
  – subroutine in Fortran
• `omp_test_lock(mylock)`
  - logical function
  - if `mylock` is presently owned by another thread, returns `false`
  - if `mylock` is available, returns `true` and sets lock, i.e., gives ownership to calling thread
  - be careful: `omp_test_lock(mylock)` does more than just test the lock
Lock Routines (4)

- `omp_unset_lock(mylock)`
  - releases ownership of `mylock`
  - subroutine in Fortran

- `omp_destroy_lock(mylock)`
  - call when you’re done with the lock
  - complement to `omp_init_lock(mylock)`
  - subroutine in Fortran
Lock Example

- an independent task takes a significant amount of time, and it must be performed serially
- another task, independent of the first, may be performed in parallel
- improve parallel efficiency by doing both tasks at the same time
- one thread locks serial task
- other threads work on parallel task until serial task is complete
- lock example