INTRODUCTION TO OPENMP

From online references of UWM, Lawrence Livermore National Lab, National Energy Research Scientific Computing Center, University of Minnesota, OpenMP.org

INTRODUCTION TO OPENMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - "Standard" API for defining multi-threaded shared-memory programs
  - www.openmp.org – Talks, examples, forums, etc.
- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

A PROGRAMMER'S VIEW OF OPENMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP *implementation*!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,* rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize (or detect!) dependencies
  - Guarantee speedup
  - Provide freedom from data races

3

OUTLINE

- Introduction
  - Motivating example
  - Parallel Programming is Hard

- OpenMP Programming Model
  - Easier than PThreads

- Microbenchmark Performance Comparison
  - vs. PThreads

4

CURRENT PARALLEL PROGRAMMING

1. Start with a parallel algorithm
2. Implement, keeping in mind:
   - Data races
   - Synchronization
   - Threading Syntax
3. Test & Debug
4. Debug
5. Debug

5

MOTIVATION – THREADING LIBRARY

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_attr_t attr;
  pthread_t threads[16];
  int tn;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], &attr, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

6

MOTIVATION

- Thread libraries are hard to use
  - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness

7

MOTIVATION

- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  - OpenMP is a small API that hides cumbersome threading calls with simpler *directives*

8

BETTER PARALLEL PROGRAMMING

1. Start with *some* algorithm
   - Embarrassing parallelism is helpful, but not necessary
2. Implement serially, *ignoring*:
   - Data Races
   - Synchronization
   - Threading Syntax
3. Test and Debug
4. Automatically (*magically?*) parallelize
   - Expect linear speedup

9

MOTIVATION – OPENMP

```
int main() {

  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```

10

MOTIVATION – OPENMP

```
int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

11

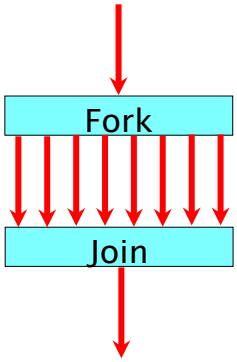
OPENMP PARALLEL PROGRAMMING

1. Start with *a parallelizable* algorithm
   - Embarrassing parallelism is good, loop-level parallelism is necessary
2. Implement serially, *mostly ignoring*:
   - Data Races
   - Synchronization
   - Threading Syntax
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   - Hope for linear speedup
5. Test and Debug

12

PROGRAMMING MODEL - THREADING

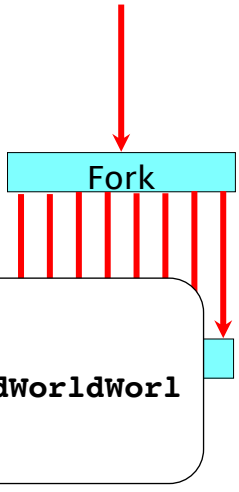
- Serial regions by default, annotate to create *parallel regions*
  - Generic parallel regions
  - Parallelized loops
  - Sectioned parallel regions
- Thread-like Fork/Join model
  - Arbitrary number of *logical* thread creation/ destruction events



13

PROGRAMMING MODEL - THREADING

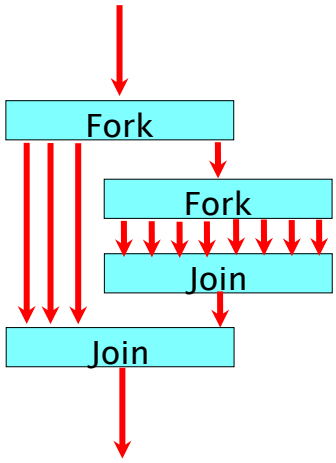
```
int main() {
  // serial region

  printf("Hello…");
  // parallel region

  #pragma omp parallel

  {

    printf("Worl
  // serial agai
  }
  printf("!");

}
```



14

PROGRAMMING MODEL – NESTED THREADING

- Fork/Join can be nested
  - Nesting complication handled "automagically" at compile-time
  - Independent of the number of threads actually running

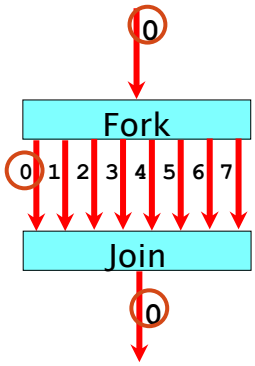


15

PROGRAMMING MODEL – THREAD IDENTIFICATION

Master Thread

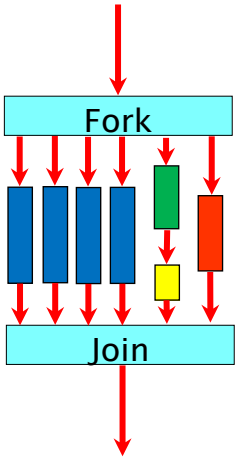
- Thread with ID=0
- Only thread that exists in sequential regions
- Depending on implementation, may have special purpose inside parallel regions
- Some special directives affect only the master thread (like `master`)



16

PROGRAMMING MODEL – DATA/CONTROL PARALLELISM

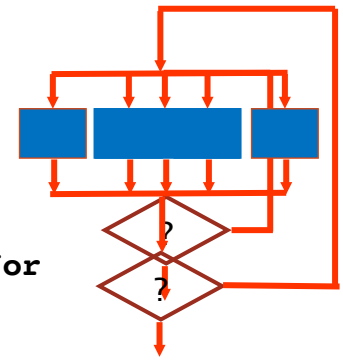
- Data parallelism
  - Threads perform similar functions, guided by thread identifier
- Control parallelism
  - Threads perform differing functions
    - One thread for I/O, one for computation, etc…



17

PROGRAMMING MODEL – CONCURRENT LOOPS

- OpenMP easily parallelizes loops
  - No data dependencies between iterations!
- Preprocessor calculates loop bounds for each thread directly from *serial* source



```
#pragma omp parallel for
for( i=0; i < 25;
i++ ) {
  printf("Foo");
}
```

18

PROGRAMMING MODEL – LOOP SCHEDULING

- `schedule` clause determines how loop iterations are divided among the thread team
  - `static([chunk])` divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default `[chunk]` is `ceil( # iterations / # threads )`
  - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default `[chunk]` is 1
  - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation

PROGRAMMING MODEL – LOOP SCHEDULING

```
#pragma omp parallel for \
 schedule(static)
 for( i=0; i<16; i++ )
 {
   doIteration(i);
 }
```

```
// Static Scheduling

int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;

for( i=base; i<bound; i++ )
{
  doIteration(i);
}

Barrier();
```

PROGRAMMING MODEL – LOOP SCHEDULING

```
#pragma omp parallel for \
 schedule(dynamic)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Dynamic Scheduling

int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(i);
}

Barrier();
```


PROGRAMMING MODEL – DATA SHARING

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)
- **PThreads:**
  - **Global-scoped variables are shared**
  - **Stack-allocated variables are private**
- **OpenMP:**
  - **`shared` variables are shared**
  - **`private` variables are private**

```
int bigdata[1024];

void* foo(void* bar) {
  int tid;

  #pragma omp parallel \
   shared ( bigdata ) \
   private ( tid )
  {
    /* Calc. here */
  }
}
```

int bigdata[1024];

void* foo(void* bar) {
  int tid;

// private, stack
int tid;
{
  /* Calculation goes
     here */
}
}

PROGRAMMING MODEL - SYNCHRONIZATION

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Named or unnamed
    - No *explicit* locks
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require `flush` directive
  - Single-thread regions *within* parallel regions
    - `master`, `single` directives

```
#pragma omp critical
{
  /* Critical code here */
}

#pragma omp barrier

omp_set_lock( lock l );
/* Code goes here */
omp_unset_lock( lock l );

#pragma omp single
{
  /* Only executed once */
}
```

23

PROGRAMMING MODEL - SUMMARY

- Threaded, shared-memory execution model
  - Serial regions and parallel regions
  - Parallelized loops with customizable scheduling
- Concurrency expressed with preprocessor directives
  - Thread creation, destruction mostly hidden
  - Often expressed *after writing* a serial version through annotation

24

PERFORMANCE CONCERNS

- Is the overhead of OpenMP too high?
  - How do the scheduling and synchronization options affect performance?
  - How does autogenerated code compare to hand-written code?
- Can OpenMP scale?
  - 4 threads? 16? More?
- What should OpenMP be compared against?
  - PThreads?
  - MPI?

25

PERFORMANCE COMPARISON: OMP VS. PTHREADS

- PThreads
  - Shared-memory, portable threading implementation
  - Explicit thread creation, destruction (`pthread_create`)
  - Explicit stack management
  - Synch: Locks, Condition variables
- Microbenchmarks implemented in OpenMP, PThreads
  - Explore OpenMP loop scheduling policies
  - Comparison vs. tuned PThreads implementation

26

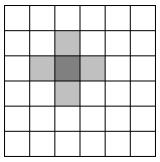
METHODOLOGY

- Microbenchmarks implemented in OpenMP and PThreads, compiled with similar optimizations, same compiler (Sun Studio)
- Execution times measured on a 16-processor Sun Enterprise 6000 (`cabernet.cs.wisc.edu`), 2GB RAM, 1MB L2 Cache
- Parameters varied:
  - Number of processors (threads)
  - Working set size
  - OpenMP loop scheduling policy

27

MICROBENCHMARK: OCEAN

- Conceptually similar to SPLASH-2's ocean
- Simulates ocean temperature gradients via successive-approximation
  - Operates on a 2D grid of floating point values



- "Embarrassingly" Parallel
  - Each thread operates in a rectangular region
  - Inter-thread communication occurs only on region boundaries
  - Very little synchronization (barrier-only)
- Easy to write in OpenMP!

28

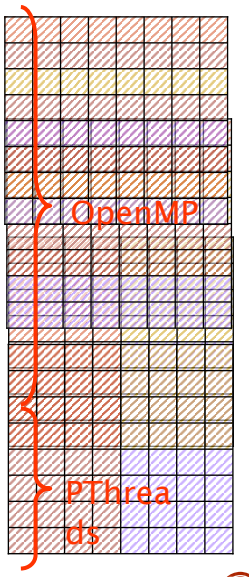
MICROBENCHMARK: OCEAN

```
for( t=0; t < t_steps; t++) {
  #pragma omp parallel for \
   shared(ocean,x_dim,y_dim) private(x,y)
  for( x=0; x < x_dim; x++) {
    for( y=0; y < y_dim; y++) {
      ocean[x][y] = /* avg of neighbors */
    }
  } // Implicit Barrier Synchronization
  temp_ocean = ocean;
  ocean = other_ocean;
  other_ocean = temp_ocean;
}
```

29

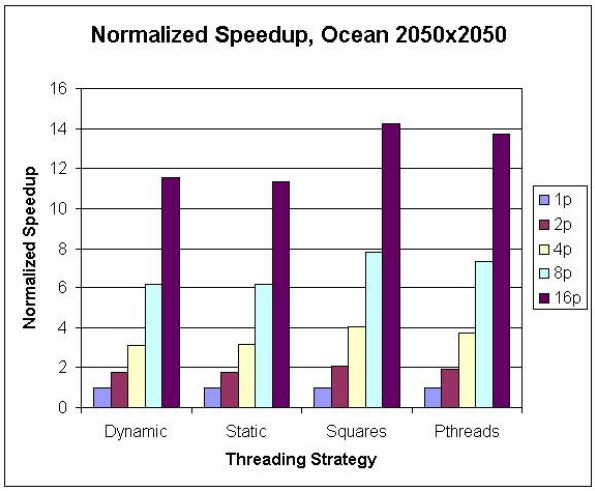
MICROBENCHMARK: OCEAN

- `ocean_dynamic` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `dynamic` scheduling.
- **`ocean_static` – Traverses entire ocean, row-by-row, assigning row iterations to threads with `static` scheduling.**
- **`ocean_squares` – Each thread traverses a square-shaped section of the ocean. Loop-level scheduling not used—loop bounds for each thread are determined explicitly.**
- **`ocean_pthreads` – Each thread traverses a square-shaped section of the ocean. Loop bounds for each thread are determined explicitly.**



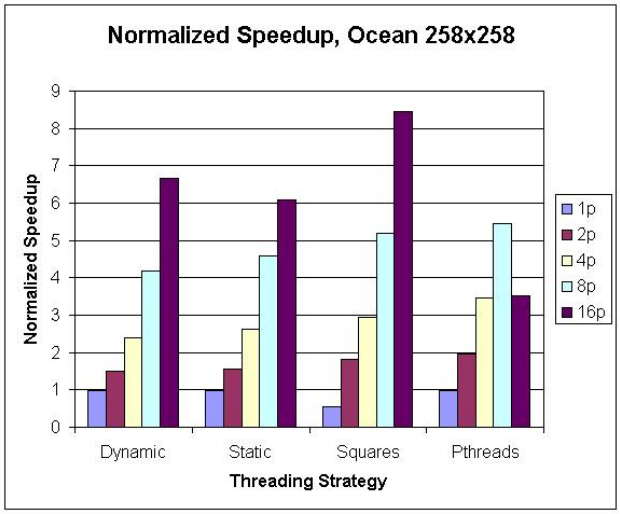
30

MICROBENCHMARK: OCEAN



31

MICROBENCHMARK: OCEAN



32

MICROBENCHMARK: GENETICTSP

- Genetic heuristic-search algorithm for approximating a solution to the traveling salesperson problem
- Operates on a *population* of possible TSP paths
  - Forms new paths by combining known, good paths (*crossover*)
  - Occasionally introduces new random elements (*mutation*)
- Variables:
  - `Np` – Population size, determines search space and working set size
  - `Ng` – Number of generations, controls effort spent refining solutions
  - `rC` – Rate of crossover, determines how many new solutions are produced and evaluated in a generation
  - `rM` – Rate of mutation, determines how often new (random) solutions are introduced

33

MICROBENCHMARK: GENETICTSP

```
while( current_gen < Ng ) {
  Breed rC*Np new solutions:
    Select two parents
    Perform crossover()
    Mutate() with probability rM
    Evaluate() new solution

  Identify least-fit rC*Np solutions:
    Remove unfit solutions from population

  current_gen++
}

return the most fit solution found
```

Outer loop has data dependence between iterations, as the population is not a loop invariant.

Can generate new solutions in parallel, but `crossover()`, `mutate()`, and `evaluate()` have varying runtimes.

Threads can find least-fit population members in parallel, but only one thread should actually delete solutions.

34

MICROBENCHMARK: GENETICTSP

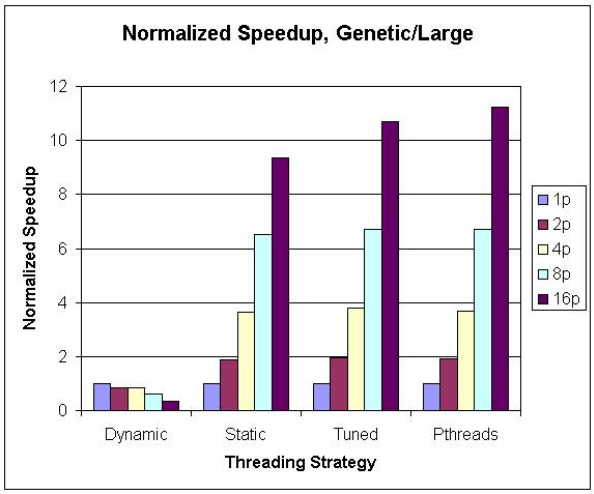
- `dynamic_tsp` – Parallelizes both breeding loop and survival loop with OpenMP's `dynamic` scheduling
- **`static_tsp` – Parallelizes both breeding loop and survival loop with OpenMP's `static` scheduling**
- **`tuned_tsp` – Attempt to tune scheduilng. Uses `guided` (exponential allocation) scheduling on breeding loop, `static` predicated scheduling on survival loop.**
- **`pthreads_tsp` – Divides iterations of breeding loop evenly among threads, conditionally executes survival loop in parallel**

OpenMP

PThreads

35

MICROBENCHMARK: GENETICTSP



36

EVALUATION

- OpenMP scales to 16-processor systems
  - Was overhead too high?
    - In some cases, yes
  - Did compiler-generated code compare to hand-written code?
    - Yes!
  - How did the loop scheduling options affect performance?
    - `dynamic` or `guided` scheduling helps loops with variable interation runtimes
    - `static` or predicated scheduling more appropriate for shorter loops
- Is OpenMP the right tool to parallelize scientific application?

37

SPECOMP (2001)

- Parallel form of SPEC FP 2000 using Open MP, larger working sets
  - Aslot et. Al., Workshop on OpenMP Apps. and Tools (2001)
- Many of CFP2000 were "straightforward" to parallelize:
  - *ammp*: 16 Calls to OpenMP API, 13 `#pragma`s, converted linked lists to vector lists
  - *applu*: 50 directives, mostly `parallel` or `do`
  - *fma3d*: 127 lines of OpenMP directives (60k lines total)
  - *mgrid*: automatic translation to OpenMP
  - *swim*: 8 loops parallelized

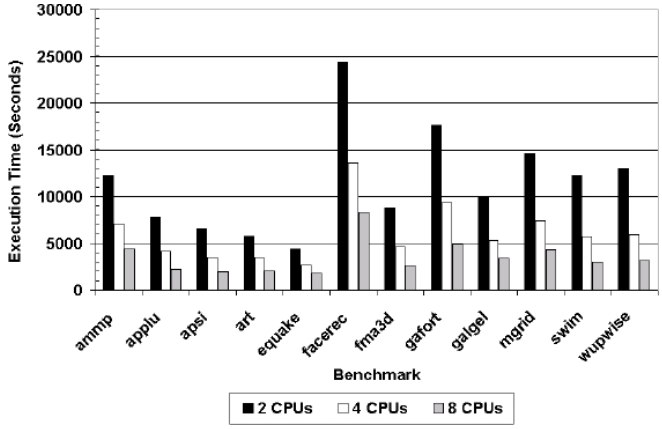
38

SPECOMP

Benchmark	Lines of Code	Parallel Cov.	# Par. Sections
ammp	13,500 (C)	99.2 %	7
applu	4,000	99.9 %	22
apsi	7,500	99.9 %	24
art	1,300 (C)	99.5 %	3
facerec	2,400	99.9 %	2
fma3d	60,000	99.5 %	30
gafort	1,500	99.9 %	6
galgel	15,300	96.8 %	29
equake	1,500 (C)	98.4 %	11
mgrid	500	99.9 %	12
swim	400	99.5 %	8
wupwise	2,200	99.8 %	6

39

SPECOMP - SCALABILITY



Aslot et. Al. Execution times on a "generic" 350Mhz machine.

40

LIMITATIONS

- OpenMP Requires compiler support
  - Sun Studio compiler
  - Intel VTune
  - Polaris/OpenMP (Purdue)
- OpenMP does not parallelize dependencies
  - Often does not *detect* dependencies
  - Nasty race conditions still exist!
- OpenMP is not guaranteed to divide work optimally among threads
  - Programmer-tweakable with scheduling clauses
  - Still lots of rope available

41

LIMITATIONS

- Doesn't totally hide concept of `volatile` data
  - From a high-level, use of OMP's locks can seem like consistency violations if `flush` directive is forgotten
- Workload applicability
  - Easy to parallelize "scientific" applications
  - How might one create an OpenMP web server? Database?
- Adoption hurdle
  - Search www.sourceforge.net for "OpenMP":
    - 3 results (out of 72,000)

42

SUMMARY

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications

43

MORE INFORMATION

- www.openmp.org
  - OpenMP official site
- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial
- www.nersc.gov/nusers/help/tutorials/openmp/
  - Another OpenMP tutorial and reference

44

Backup Slides

Syntax, etc

45

CONSISTENCY VIOLATION?

```
#pragma omp parallel for \
  shared(x) private(i)
for( i=0; i<100; i++ ) {
  #pragma omp atomic
  x++;
}
printf("%i",x);
```

100

```
#pragma omp parallel for \
  shared(x) private(i)
for( i=0; i<100; i++ ) {
  omp_set_lock(my_lock);
  x++;
  #pragma omp flush
  omp_unset_lock(my_lock);
}
printf("%i",x);
```

19060

46

OPENMP SYNTAX

- General syntax for OpenMP directives

```
#pragma omp directive [clause…] CR
```

- *Directive* specifies type of OpenMP operation
  - Parallelization
  - Synchronization
  - Etc.
- *Clauses* (optional) modify semantics of *Directive*

47

OPENMP SYNTAX

- PARALLEL syntax

```
#pragma omp parallel [clause…] CR
     structured_block
```

Ex:

```
#pragma omp parallel
{
  printf("Hello!\n");
} // implicit barrier
```

Output: (T=4)

```
Hello!
Hello!
Hello!
Hello!
```

48

OPENMP SYNTAX

- DO/for Syntax (DO-Fortran, for-C)

```
#pragma omp for [clause…] CR
      for_loop
```

Ex:

```
#pragma omp parallel
{
  #pragma omp for private(i) shared(x) \
                 schedule(static,x/N)
  for(i=0;i<x;i++) printf("Hello!\n");
} // implicit barrier
```

Note: Must reside inside a `parallel` section

49

OPENMP SYNTAX

More on Clauses

- `private()` – A variable in private list is private to each thread
- `shared()` – Variables in shared list are visible to all threads
  - Implies no synchronization, or even consistency!
- `schedule()` – Determines how iterations will be divided among threads
  - `schedule(static, C)` – Each thread will be given C iterations
    - Usually T*C = Number of total iterations
  - `schedule(dynamic)` – Each thread will be given additional iterations as-needed
    - Often less efficient than considered static allocation
- `nowait` – Removes implicit barrier from end of block

50

OPENMP SYNTAX

- PARALLEL FOR (combines `parallel` and `for`)

```
#pragma omp parallel for [clause…] CR
     for_loop
```

Ex:

```
#pragma omp parallel for shared(x)\
                         private(i) \
                         schedule(dynamic)
for(i=0;i<x;i++) {
  printf("Hello!\n");
}
```

51

EXAMPLE: ADDMATRIX

Files:

```
(Makefile)
 addmatrix.c        // omp-parallelized
 matrixmain.c       // non-omp
 printmatrix.c      // non-omp
```

52

OPENMP SYNTAX

- ATOMIC syntax

```
#pragma omp atomic CR
      simple_statement
```

Ex:

```
#pragma omp parallel shared(x)
{
  #pragma omp atomic
  x++;
} // implicit barrier
```

53

OpenMP Syntax

- **CRITICAL syntax**

```
#pragma omp critical CR
      structured_block
```

Ex:

```
#pragma omp parallel shared(x)
{
  #pragma omp critical
  {
    // only one thread in here
  }
} // implicit barrier
```

54

OPENMP SYNTAX

ATOMIC vs. CRITICAL

- Use ATOMIC for "simple statements"
  - Can have lower overhead than CRITICAL if HW atomics are leveraged (implementation dep.)
- Use CRITICAL for larger expressions
  - May involve an unseen implicit lock

OPENMP SYNTAX

- MASTER – only Thread 0 executes a block

```
#pragma omp master CR
     structured_block
```

- SINGLE – only one thread executes a block

```
#pragma omp single CR
     structured_block
```

- No implied synchronization

OPENMP SYNTAX

- BARRIER

```
#pragma omp barrier CR
```

- Locks
  - Locks are provided through `omp.h` library calls
  - `omp_init_lock()`
  - `omp_destroy_lock()`
  - `omp_test_lock()`
  - `omp_set_lock()`
  - `omp_unset_lock()`

57

OPENMP SYNTAX

- FLUSH

```
#pragma omp flush CR
```

- Guarantees that threads' views of memory is consistent
- Why? Recall OpenMP directives…
  - Code is generated by directives at compile-time
    - Variables are not always declared as `volatile`
    - Using variables from registers instead of memory can seem like a consistency violation
  - Synch. Often has an implicit flush
    - ATOMIC, CRITICAL

58

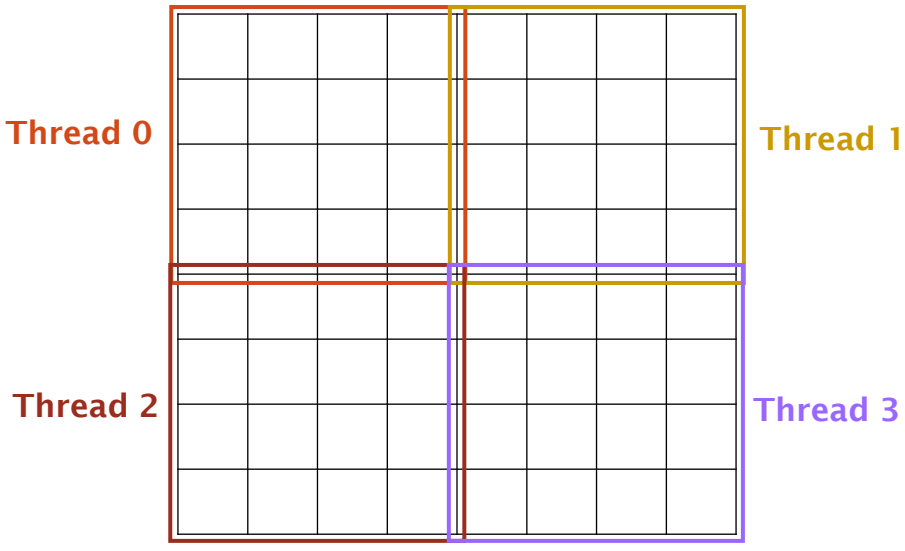
OPENMP SYNTAX

- Functions

```
omp_set_num_threads()
omp_get_num_threads()
omp_get_max_threads()
omp_get_num_procs()
omp_get_thread_num()
omp_set_dynamic()
omp_[init destroy test set unset]_lock()
```

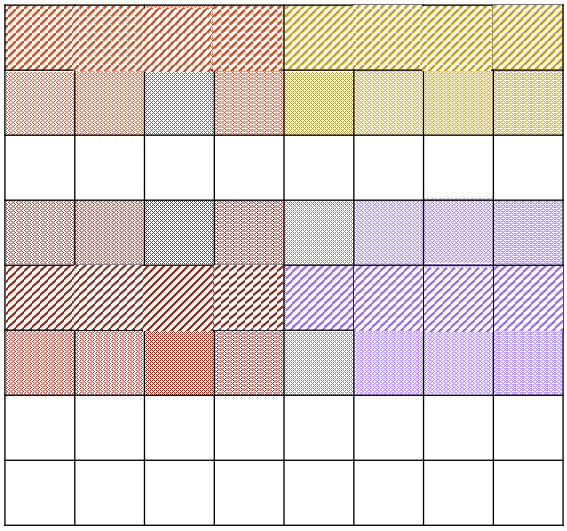
59

MICROBENCHMARK: OCEAN



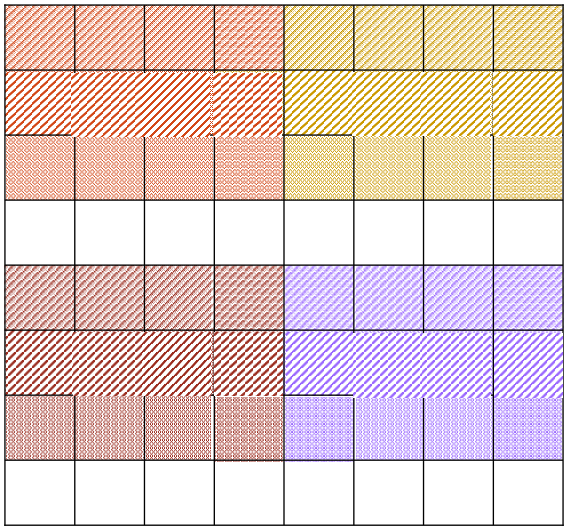
60

MICROBENCHMARK: OCEAN



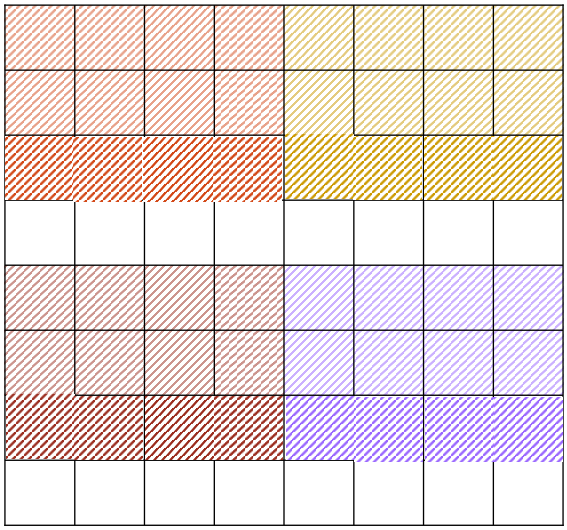
61

MICROBENCHMARK: OCEAN



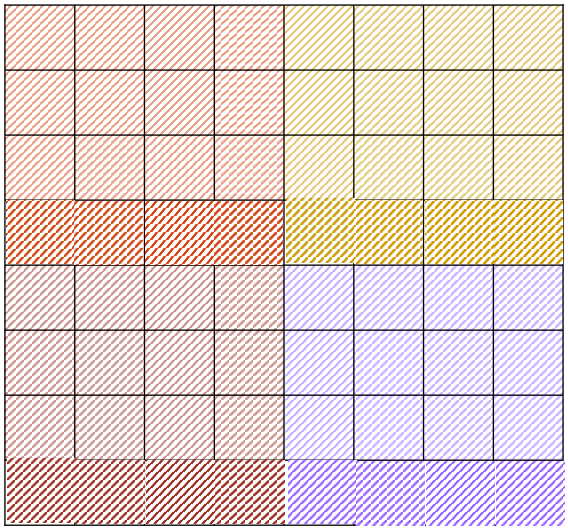
62

MICROBENCHMARK: OCEAN



63

MICROBENCHMARK: OCEAN



64

MICROBENCHMARK: OCEAN