

OpenMP

Kenjiro Taura

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

A running example: Sparse Matrix Vector Multiply (SpMV)

- **sparse matrix** : a matrix whose elements are mostly zeros
- i.e. the number of non-zero elements (**nnz**) \ll the number of all elements ($M \times N$)
 - M : the number of rows
 - N : the number of columns

Sparse matrices appear everywhere

- meshes in scientific simulation
 - $A_{i,j}$ = a weight connecting nodes i and j in the mesh
- graphs, which in turn appear in many applications
 - $A_{i,j}$ = the weight of the edge $i \rightarrow j$ (or $j \rightarrow i$)
 - Web, social network, road/traffic networks, citation networks, metabolic pathways, etc.
- many problems can be solved using SpMV
 - eigenvalues (including PageRank, graph partitioning, etc.)
 - partial differential equation
 - ...

What makes “sparse” matrix different from ordinary (dense) matrix?

- the number of non-zero elements are so small that representing it as $M \times N$ array is too wasteful (or just impossible)
- \rightarrow use a data structure that takes memory/computation only (or mostly) for non-zero elements (coordinate list, compressed sparse row, etc.)

Coordinate list (COO)

- represent a matrix as a list of $(i, j, A_{i,j})$'s
- data format:

```
1 struct coo {  
2     int n_rows, n_cols, nnz;  
3     /* nnz elements */  
4     struct { i, j, Aij } * elems;  
5 };
```

- SpMV ($y = Ax$)

```
1 for (k = 0; k < A.nnz; k++) {  
2     i, j, Aij = A.elems[k];  
3     y[i] += Aij * x[j];  
4 }
```

Compressed sparse row (CSR)

- puts elements of a single row in a contiguous range
- an index (number) specifies where a particular row begins in the `elems` array
- \rightarrow *no need to have i for every single element*
- data format:

```
1 struct coo {  
2     int n_rows, n_cols, nnz;  
3     struct { j, Aij } * elems; // nnz elements  
4     int * row_start; // n_rows elements  
5 };
```

`elems[row_start[i]] ... elems[row_start[i + 1]]` are the elements in the i th row

- SpMV ($y = Ax$)

```
1 for (i = 0; i < A.n_rows; i++) {  
2     for (k = A.row_start[i]; k < A.row_start[i+1]; k++) {  
3         j, Aij = A.elems[k];  
4         y[i] += Aij * x[j];  
5     } }
```


- *de fact* standard model for programming shared memory machines
- C/C++/Fortran + directives + APIs
 - by `#pragma` in C/C++
 - by comments in Fortran
- many free/vendor compilers, including GCC, LLVM, NVIDIA HPC SDK

OpenMP reference

- official home page: <http://openmp.org/>
- specification:
<http://openmp.org/wp/openmp-specifications/>
- latest version is 5.0
(<https://www.openmp.org/spec-html/5.0/openmp.html>)
- section numbers below refer to those in OpenMP spec 5.0

Compiling OpenMP programs for multicores

- GCC and LLVM (clang/clang++) : compile with `-fopenmp`

```
1 $ clang -Wall -fopenmp program.c
2 $ gcc -Wall -fopenmp program.c
```

- NVIDIA HPC SDK (nvc/nvc++) : compile with `-mp`

```
1 $ nvc -Wall -mp program.c
```

- In this lecture, we use LLVM and NVIDIA HPC SDK, as they support OpenMP for multicore, GPU offloading, and CUDA

Running OpenMP programs

- run the executable specifying the number of threads with `OMP_NUM_THREADS` environment variable

```
1 $ OMP_NUM_THREADS=1 ./a.out # use 1 thread
2 $ OMP_NUM_THREADS=4 ./a.out # use 4 threads
```

- if `OMP_NUM_THREADS` is unspecified, it uses the number of available processors visible to OS, including hyperthreading
- see 2.6.1 “Determining the Number of Threads for a parallel Region” for more details and other ways to control the number of threads

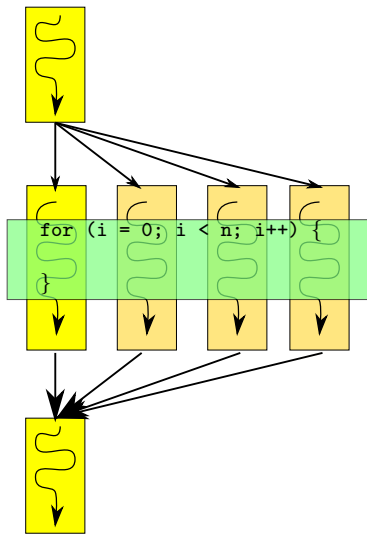
Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Two pragmas you must know first

- `#pragma omp parallel` to launch a team of threads (2.6)
- then `#pragma omp for` to distribute iterations to threads (2.9.2)

Note: all OpenMP pragmas have the common format: `#pragma omp ...`



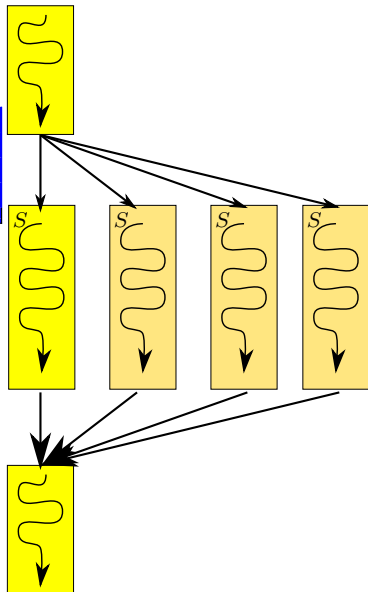
#pragma omp parallel

- basic syntax:

```
1  ...  
2  #pragma omp parallel  
3  S  
4  ...
```

- basic semantics:

- create a team of OMP_NUM_THREADS threads
- the current thread becomes the *master* of the team
- *S will be executed by each member of the team*
- the master thread waits for all to finish *S* and continue



parallel pragma example

```
1 #include <stdio.h>
2 int main() {
3     printf("hello\n");
4     #pragma omp parallel
5     printf("world\n");
6     printf("bye\n");
7     return 0;
8 }
```

```
1 $ OMP_NUM_THREADS=1 ./a.out
2 hello
3 world
4 $ OMP_NUM_THREADS=4 ./a.out
5 hello
6 world
7 world
8 world
9 world
10 bye
```


Remarks : what does `parallel` do?

- you may assume an OpenMP thread \approx OS-supported thread (e.g., Pthread)
- that is, if you write this program

```
1 int main() {  
2   #pragma omp parallel  
3   worker();  
4 }
```

and run it as follows,

```
1 $ OMP_NUM_THREADS=50 ./a.out
```

you will get 50 OS-level threads, each doing `worker()`

How to distribute work among threads?

- `#pragma omp parallel` creates threads, *all executing the same statement*
- it's not a means to parallelize work, *per se*, but just a means to create a number of similar threads
 - Single Program Multiple Data (SPMD) model
- so how to distribute (or partition) work among them?
 - ① do it yourself
 - ② use *work sharing* constructs

Do it yourself: functions to get the number/id of threads

- `omp_get_num_threads()` (3.2.2) : the number of threads *in the current team*
- `omp_get_thread_num()` (3.2.4) : the current thread's id (0, 1, ...) in the team
- they are primitives with which you may partition work yourself by whichever ways you prefer
- e.g.,

```
1  #pragma omp parallel
2  {
3      int t = omp_get_thread_num();
4      int nt = omp_get_num_threads();
5      /* divide n iterations evenly amongst nt threads */
6      for (i = t * n / nt; i < (t + 1) * n / nt; i++) {
7          ...
8      }
9  }
```

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Work sharing constructs

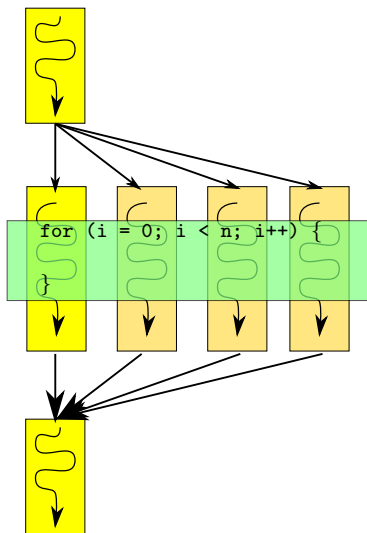
- in theory, `parallel` construct is all you need to do things in parallel
- but it's too inconvenient
- OpenMP defines ways to *partition* work among threads (*work sharing constructs*)
 - `for`
 - `task`

#pragma omp for (work-sharing for)

- basic syntax (2.9.2):

```
1  #pragma omp for
2  for(i=...; i...; i+=...){
3      S
4  }
```

- basic semantics:
the threads in the team
divide the iterations among
them
- but how? \Rightarrow scheduling



#pragma omp for restrictions

- iterations are executed in any order may interleave
 - the program must not rely on the order in which they are executed
- strong syntactic restrictions apply (2.9.1); basically, *the iteration space must be easily identifiable at the beginning* of the loop
 - roughly, it must be of the form:

```
1  #pragma omp for
2  for(i = init; i < limit; i += incr)
3      S
```

except < and += may be other similar operators

- *init*, *limit*, and *incr* must be loop invariant

Parallel SpMV for CSR using `#pragma omp for` for

- it only takes to work-share the outer for loop

```
1 // assume inside #pragma omp parallel
2 ...
3 #pragma omp for
4 for (i = 0; i < A.n_rows; i++) {
5     for (k = A.row_start[i]; k < A.row_start[i+1]; k++) {
6         j, Aij = A.elems[k];
7         y[i] += Aij * x[j];
8     }
9 }
```

- note: the inner loop (k) is executed sequentially

Parallel SpMV COO using `#pragma omp for`?

- the following code does *not* work (why?)

```
1 // assume inside #pragma omp parallel
2 ...
3 #pragma omp for
4 for (k = 0; k < A.nnz; k++) {
5     i,j,Aij = A.elems[k];
6     y[i] += Aij * x[j];
7 }
```

- a possible remedy will be described later

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Scheduling (2.9.2)

- `schedule` clause in work-sharing for loop determines how iterations are divided among threads
- There are three alternatives (`static`, `dynamic`, and `guided`)

static, dynamic, and guided

- `schedule(static[,chunk])`:
predictable round-robin
- `schedule(dynamic[,chunk])`:
each thread repeats fetching *chunk* iterations
- `schedule(guided[,chunk])`:
threads grab many iterations in early stages; gradually reduce iterations to fetch at a time
- *chunk* specifies the minimum granularity (iteration counts)

static



static, 3



dynamic



dynamic, 2



guided



guided, 2



Other scheduling options and notes

- `schedule(runtime)` determines the schedule by `OMP_SCHEDULE` environment variable. e.g.,

```
1 $ OMP_SCHEDULE=dynamic,2 ./a.out
```

- `schedule(auto)` or `no schedule clause` choose an implementation dependent default

Parallelizing loop nests by `collapse`

- `collapse(l)` can be used to partition nested loops. e.g.,

```
1 #pragma omp for collapse(2)
2 for (i = 0; i < n; i++)
3     for (j = 0; j < n; j++)
4         S
```

will partition n^2 iterations of the doubly-nested loop

- `schedule` clause applies to nested loops as if the nested loop is an equivalent flat loop
- restriction: the loop must be “*perfectly nested*” (the iteration space must be a rectangular and no intervening statement between different levels of the nest)

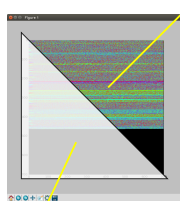
Visualizing schedulers

- seeing is believing. let's visualize how loops are distributed among threads
- write a simple doubly nested loop and run it under various scheduling options

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

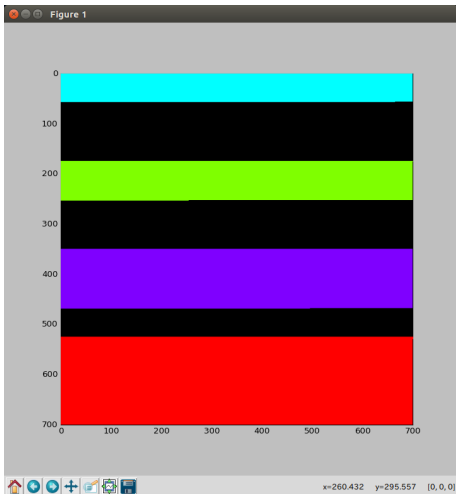
- load per point is systematically skewed:
 - ≈ 0 in the lower triangle
 - randomly drawn from

load $\sim [100, 10000]$ clocks

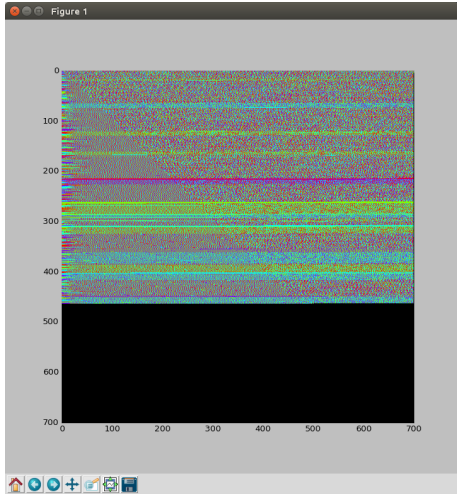


load ≈ 0

Visualizing schedulers



static



dynamic

Scheduling for SpMV on CSR

```
1 // assume inside #pragma omp parallel
2 ...
3 #pragma omp for schedule(???)
4 for (i = 0; i < A.n_rows; i++) {
5     for (k = A.row_start[i]; k < A.row_start[i+1]; k++) {
6         j, Aij = A.elems[k];
7         y[i] += Aij * x[j];
8     }
9 }
```

- **static?** depending on the number of elements in rows, load imbalance may be significant
- **dynamic/guided?** load balancing will be better, but extremely dense rows may still be an issue
- the more robust strategy is to partition non-zeros, not rows

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Task parallelism in OpenMP

- OpenMP's initial focus was simple parallel loops
- since 3.0, it supports task parallelism
- but why it's necessary?
- aren't `parallel` and `for` all we need?

Limitation of parallel for

- what if you have a parallel loop inside another

```
1 for ( ... ) {  
2     ...  
3     for ( ... ) ...  
4 }
```

- perhaps in a function?

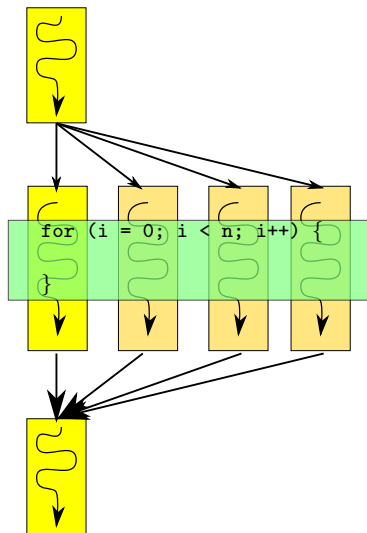
```
1 main() {  
2     for ( ... ) {  
3         ...  
4         g();  
5     }  
6 }  
7 g() {  
8     for ( ... ) ...  
9 }
```

- what about parallel recursions?

```
1 qs() {  
2     if (...) { ... }  
3     else {  
4         qs();  
5         qs();  
6     }  
7 }
```

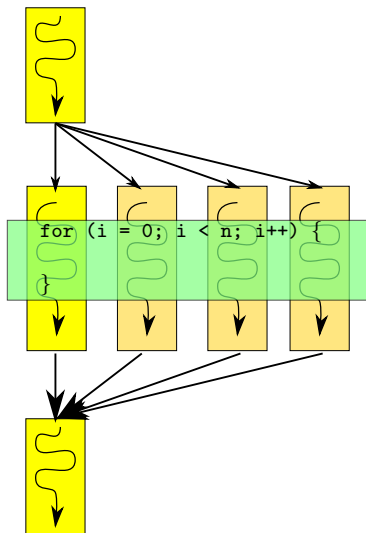
parallel for can't handle nested parallelism

- OpenMP generally ignores nested `parallel` pragma when enough threads have been created by the outer `parallel` pragma, for good reasons



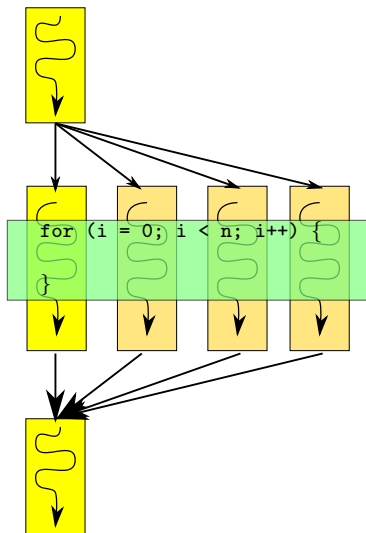
parallel for can't handle nested parallelism

- OpenMP generally ignores nested `parallel` pragma when enough threads have been created by the outer `parallel` pragma, for good reasons
- the fundamental limitation is its simplistic work-sharing mechanism



parallel for can't handle nested parallelism

- OpenMP generally ignores nested `parallel` pragma when enough threads have been created by the outer `parallel` pragma, for good reasons
- the fundamental limitation is its simplistic work-sharing mechanism
- *tasks* address these issues, by allowing tasks to be created at arbitrary points of execution (and a mechanism to distribute them across cores)



Task parallelism in OpenMP

- syntax:

- **task** creates a task executing S (2.10.1)

```
1  #pragma omp task  
2  S
```

- **taskwait** waits for child tasks to finish (2.17.5)

```
1  #pragma omp taskwait
```

OpenMP task parallelism template

- don't forget to create a **parallel** region
- don't also forget to enter a **master** region, which says only the master executes the following statement and others “stand-by”

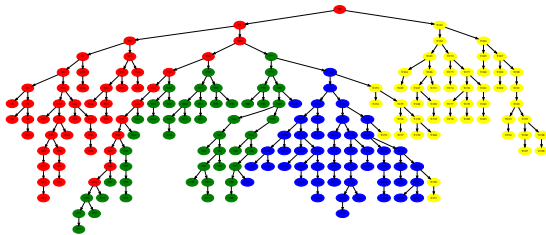
```
1 int main() {  
2     #pragma omp parallel  
3     #pragma omp master  
4     // or #pragma omp single  
5     ms(a, a + n, t, 0);  
6 }
```

- and create tasks in the master region

```
1 void ms(a, a_end, t, dest) {  
2     if (n == 1) {  
3         ...  
4     } else {  
5         ...  
6         #pragma omp task  
7             ms(a, c, t, 1 - dest);  
8         #pragma omp task  
9             ms(c, a_end, t + nh, 1 - dest);  
10        #pragma omp taskwait  
11        ...  
12    }
```

What are tasks good for?

- the strength of tasks as opposed to for loop is its flexibility
 - create tasks at any point during the computation
 - they get distributed to cores
- especially good for “nested parallelism” and “parallel recursions (divide and conquer)”



- even for loops, you may consider reformulating them into divide-and-conquer as an alternative dynamic load-balancing strategy

Visualizing task parallel schedulers

- the workload is exactly the same as before

```
1 #pragma omp for collapse(2) schedule(runtime)
2 for (i = 0; i < 1000; i++)
3     for (j = 0; j < 1000; j++)
4         unit_work(i, j);
```

load $\sim [100, 10000]$ clocks

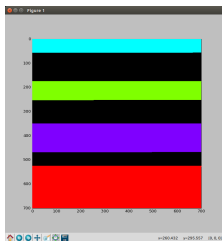


- but we rewrite it into recursions

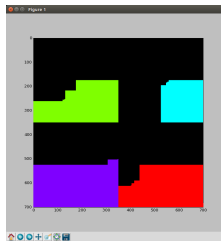
```
1 void work_rec(rectangle b) {
2     if (small(b)) {
3         ...
4     } else {
5         rectangle c[2][2];
6         split(b, c); // split b into 2x2 sub-rectangles
7         for (i = 0; i < 2; i++) {
8             for (j = 0; j < 2; j++) {
9                 #pragma omp task
10                    work_rec(b[i][j]);
11            }
12        }
13    #pragma omp taskwait
14    }
15 }
```

load ≈ 0

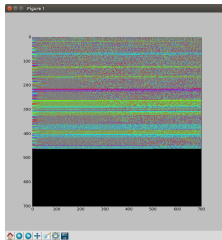
Visualizing schedulers



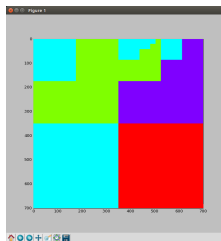
static



2D recursive (midway)



dynamic



2D recursive (end)

SpMV with divide and conquer

- you may recursively divide the matrix A submatrices, until nnz in a submatrix becomes sufficiently small (*divide and conquer*)
- putting memory management issues aside, it is:

```
1 void SpMV_rec(A, x) {
2     if (nnz(A) is small) {
3         return SpMV_serial(A, x, y);
4     } else if (M >= N) {
5         A0_,A1_ = divide_rows(A);
6         y0 = SpMV_rec(A0_, x);
7         y1 = SpMV_rec(A1_, x);
8         return y0 ++ y1; // concatenation
9     } else {
10        A_0,A_1 = divide_cols(A);
11        x0,x1 = divide(x);
12        y0 = SpMV_rec(A_0, x0);
13        y1 = SpMV_rec(A_0, x0);
14        return y0 + y1; // vector addition
15    }
16 }
```

...and there is **taskloop**

- syntax:

```
1  #pragma omp taskloop
2  for(i = init; i < limit; i += incr)
3      S
```

- syntactic restrictions are equivalent to work-sharing for
- conceptually, it creates tasks each of which is responsible for an (or a few) iteration(s)
- unlike work-sharing for, it is generating tasks, so **#pragma omp taskloop** is supposed to be executed by a single thread, like the **task** construct

Pros/cons of various approaches

- `static`:

- partitioning iterations is **simple and does not require communication**
- mapping between work \leftrightarrow thread is **deterministic and predictable** (why it's important?)
- may cause **load imbalance** (leave some threads idle, even when other threads have many work to do)

- `dynamic`:

- **less prone to load imbalance**, if chunks are sufficiently small
- partitioning iterations **needs communication** (no two threads execute the same iteration) and may become a bottleneck
- mapping between iterations and threads is **non-deterministic**
- OpenMP's dynamic scheduler is **inflexible in partitioning nested loops**

Pros/cons of schedulers

- divide and conquer + tasks :
 - less prone to load imbalance, as in dynamic
 - distributing tasks needs communication, but efficient implementation techniques are known
 - mapping between work and thread is non-deterministic, as in dynamic
 - you can flexibly partition loop nests in various ways (e.g., keep the space to square-like)
 - need some coding efforts (easily circumvented by additional libraries; e.g., TBB's `blocked_range2d` and `parallel_for`)

Deterministic and predictable schedulers

- programs often execute the same for loops many times, with the same trip counts, and with the same iteration touching a similar region
- such *iterative* applications may benefit from reusing data brought into cache in the previous execution of the same loop
- a deterministic scheduler achieves this benefit



Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

Data sharing

- `parallel`, `for`, `task` pragma accept clauses specifying which variables should be shared among threads or between the parent/child tasks (or otherwise privatized/replicated to each thread)
- 2.19 “Data Environments”
 - `private`
 - `firstprivate`
 - `shared`
 - `reduction` (only for `parallel` and `for`)
 - `copyin`

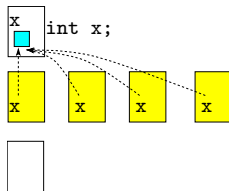
Data sharing/privatizing example

```
1  int main() {  
2      int S;  /* shared */  
3      int P;  /* made private below */  
4      #pragma omp parallel private(P) shared(S)  
5      {  
6          int L; /* automatically private */  
7          printf("S at %p, P at %p, L at %p\n",  
8                &S, &P, &L);  
9      }  
10     return 0;  
11 }
```

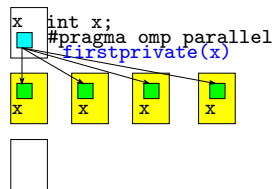
```
1  $ OMP_NUM_THREADS=2 ./a.out  
2  S at 0x..777f494, P at 0x..80d0e28, L at 0x..80d0e2c  
3  S at 0x..777f494, P at 0x..777f468, L at 0x..777f46c
```

Data sharing behavior

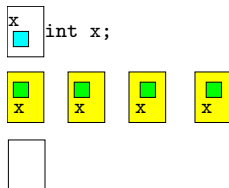
shared



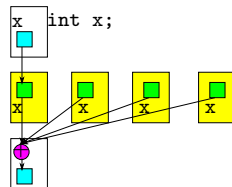
firstprivate



private



reduction



Race condition

- **definition:** there is *a race condition* when concurrent threads access the same location and one of which writes to it
- *a race condition* almost always implies your program won't work
- even something as simple as this (some accumulations may be lost)

```
1 x = 123;
2 #pragma omp parallel // assume we have 5 threads
3 {
4     ...
5     x++;
6     ..
7 }
8 printf("x = %d\n", x)
```

Race condition

thread 1	thread 2
$x(123) \rightarrow t$	
$x \leftarrow 124$	

Race condition

thread 1	thread 2
$x(123) \rightarrow t$	$x(123) \rightarrow t$
$x \leftarrow 124$	$x \leftarrow 124$

Race condition

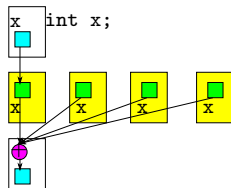
thread 1	thread 2
$x(123) \rightarrow t$	$x(123) \rightarrow t$
$x \leftarrow 124$	$x \leftarrow 124$

- The increment by a thread is “lost”

Two basic tools to resolve race conditions

- “*make it atomic*” `#pragma omp atomic` and `#pragma omp critical`: guarantee the specified operation to be done *atomically*
- “*all you need may be a reduction*” `reduction` clause performs efficient *reduction* operations on behalf of you

thread 1	thread 2
$x (123) \rightarrow t$ $x \leftarrow 124$	
	$x (124) \rightarrow t$ $x \leftarrow 125$



#pragma omp critical

- syntax:

```
1 #pragma omp critical
2     statement
```

- effect: the execution of *statement* will not overlap with other executions of *statement* (or any other statement labeled #pragma omp critical, for that matter)
- note: most general, but likely to be slow

#pragma omp atomic

- syntax:

```
1 #pragma omp atomic
2   var = var op exp
```

op is a predefined operation such as +, -, *, ...

- **effect:** guarantee the read-update is done atomically (is not lost); that is, *var* is not updated by someone else between the read and update
- **note:** semantically, it is like

```
1   e = exp;
2 #pragma omp critical
3   var = var op e
```

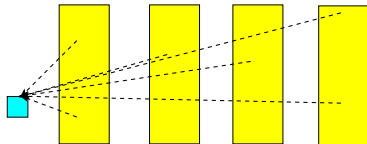
but typical implementations take advantage of atomic instructions supported by CPU, such as fetch-and-add or compare-and-swap

Reduction

- in general, “reduction” refers to an operation to combine many values into a single value. e.g.,
 - $v = v_1 + \dots + v_n$
 - $v = \max(v_1, \dots, v_n)$
 - ...
- simply sharing the variable (v) does not work (race condition)
- one way to fix is to make updates atomic, but it will be slow

```
1 v = 0.0;  
2 for (i = 0; i < n; i++) {  
3     v += f(a + i * dt) * dt;  
4 }
```

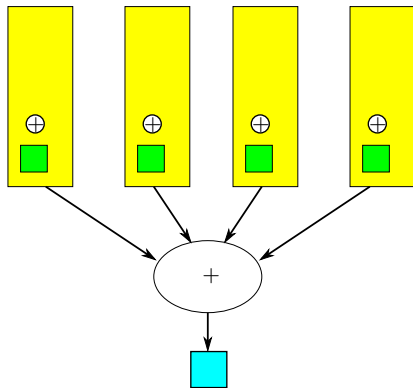
```
1 v = 0.0;  
2 #pragma omp parallel for  
3 for (i = 0; i < n; i++) {  
4     #pragma omp atomic  
5     v += f(a + i * dt) * dt;  
6 }
```



Reduction clause in OpenMP

- a more efficient strategy:
 - let each thread work (reduce) on its private variable, and
 - when threads finish, combine their partial results into one
- reduction clause in OpenMP does just that (2.19.5)

```
1  v = 0.0;
2  #pragma omp parallel for
   reduction(+:v)
3  for (i = 0; i < n; i++) {
4      v += f(a + i * dt) * dt;
5  }
```



Builtin reduction and user-defined reduction (2.9.2)

- reduction syntax:

```
1 #pragma omp parallel reduction(op:var,var,...)  
2     S
```

- builtin reductions
 - *op* is one of +, *, -, &, ^, |, &&, and ||
 - (Since 3.1) min or max
- builtin reductions are *limited to simple types and common operations* → *user-defined reductions* (since 4.0)

Why do you want user-defined reductions?

- consider how to do reduction on 3-element vector
- e.g., how to parallelize this loop safely

```
1  typedef struct {  
2      double a[3];  
3  } vec_t;  
4  
5  int main() {  
6      vec_t y;  
7      vec_init(&y);          /* y = {0,0,0} */  
8      #pragma omp parallel  
9      #pragma omp for  
10     for (long i = 0; i < 10000; i++) {  
11         y.a[i % 3] += 1;  
12     }  
13 }
```

- you cannot say `reduction(+:y.a[0], y.a[1], y.a[2])`
(what if you have 100 elements?)
- we define a reduction operation on `vec_t` type instead

User-defined reduction

- **syntax:** (2.19.5.7)

```
1 #pragma omp declare reduction (name : type : combine_statement)
```

OR

```
1 #pragma omp declare reduction (name : type : combine_statement) initializer  
    (init_statement)
```

- **effect:**

- you can specify `reduction(name : var)` for a variable of type *type*
- *init_statement* is executed by each thread before entering the loop, typically to initialize its private copy of *var*
- *combine_statement* is executed to merge a partial result to another variable

User-defined reduction: a simple example

- introduce reduction

```
1 #pragma omp declare reduction \  
2   (vp : vec_t : vec_add(&omp_out,&omp_in)) \  
3   initializer(vec_init(&omp_priv))
```

`vec_add` must be defined somewhere and not shown

- add `reduction(vp : y)` to the for loop

```
1 int main() {  
2     vec_t y;  
3     vec_init(&y);                /* y={0,0,0} */  
4     #pragma omp parallel  
5     #pragma omp for reduction(vp : y)  
6     for (long i = 0; i < 10000; i++) {  
7         y.a[i % 3] += 1;  
8     }  
9 }
```

User-defined reduction : how it works

with

```
1 #pragma omp declare reduction \  
2   (vp : vec_t : vec_add(&omp_out,&omp_in)) \  
3   initializer(vec_init(&omp_priv))
```

```
1 #pragma omp for reduction(vp : y)  
2   for (long i = 0; i < 10000; i++) {  
3     y.a[i % 3] += 1;  
4   }
```

≈

```
1   vec_t y_priv; // thread-local copy of y  
2   vec_init(&y_priv); // initializer  
3 #pragma omp for  
4   for (long i = 0; i < 10000; i++) {  
5     y_priv.a[i % 3] += 1;  
6   }  
7   // merge the partial result into the shared variable  
8   // actual implementation may be (is likely to be) different  
9   vec_add(&y, &y_priv); // y += y_priv
```

User-defined reduction : limitations

- *combine-statement* can reference only two local variables (`omp_in` and `omp_out`)
 - it should reduce (merge) `omp_in` into `omp_out` (e.g., `omp_out += omp_in`)
- *init-statement* can reference only two local variables (`omp_priv` and `omp_orig`)
 - `omp_priv` : the private copy *init-statement* should initialize
 - `omp_orig` : the original shared variable
- \Rightarrow local contexts necessary for initialization and reduction must be encapsulated in the variables subject to reduction

An exercise : reduction on variable-length vectors

- a variable-length version of the previous example

```
1 typedef struct {  
2     long n;      // number of elements (variable)  
3     double * a; // n elements  
4 } vec_t;
```

- and a reduction for it

```
1     vec_t y;  
2     long n = 100;  
3     vec_init(&y, n); // n is a local context  
4 #pragma omp parallel  
5 #pragma omp for      // how to do a proper reduction for y?  
6     for (long j = 0; j < 1000000; j++) {  
7         y.a[j % n] += 1;  
8     }
```

- the point is you cannot reference *n* in the initializer

```
1 (!) #pragma omp declare reduction \  
2     (vp : vec_t : vec_add(&omp_out,&omp_in)) \  
3     initializer(vec_init(&omp_priv, n))
```

An exercise : reduction on variable-length vectors

- initializer can reference `omp_orig` to obtain the context (i.e. vector length in this example)
- \Rightarrow define a function, `vec_init_from`, which takes the shared `y` and initialize the private copy of `y`

```
1 int vec_init_from(vec_t * v, vec_t * orig) {  
2     long n = orig->n;  
3     double * a = (double *)malloc(sizeof(double) * n);  
4     for (long i = 0; i < n; i++) {  
5         a[i] = 0;  
6     }  
7     v->n = n;  
8     v->a = a;  
9     return 0;  
10 }
```

- and say

```
1 #pragma omp declare reduction \  
2     (vp : vec_t : vec_add(&omp_out,&omp_in)) \  
3     initializer(vec_init_from(&omp_priv, &omp_orig))
```

Contents

- 1 A Running Example: SpMV
- 2 `parallel` pragma
- 3 Work sharing constructs
 - loops (`for`)
 - scheduling
 - task parallelism (`task` and `taskwait`)
- 4 Data sharing clauses
- 5 SIMD constructs

SIMD constructs

- `simd pragma` (2.9.3)
 - allows an explicit vectorization of for loops
 - syntax restrictions similar to `omp for` pragma apply
- `declare simd pragma` (2.9.3.3)
 - instructs the compiler to generate vectorized versions of a function
 - with it, loops with function calls can be vectorized

simd pragma

- basic syntax (similar to `omp for`):

```
1 #pragma omp simd clauses
2 for (i = ...; i < ...; i += ...)
3     S
```

- clauses
 - `aligned(var,var,...:align)`
 - `uniform(var,var,...)` says variables are loop invariant
 - `linear(var,var,...:stride)` says variables have the specified stride between consecutive iterations

declare simd pragma

- basic syntax (similar to omp for):

```
1 #pragma omp declare simd clauses  
2 function definition
```

- clauses
 - those for simd pragma
 - notinbranch
 - inbranch

SIMD pragmas, rationales

- most automatic vectorizers give up vectorization in many cases
 - ① conditionals (lanes may branch differently)
 - ② inner loops (lanes may have different trip counts)
 - ③ function calls (function bodies are not vectorized)
 - ④ iterations may not be independent
- `simd` and `declare simd` directives should eliminate obstacles 3 and 4 and significantly enhance vectorization opportunities

A note on GCC OpenMP SIMD implementation

- GCC `simd` and `declare simd` \approx existing auto vectorizer – dependence analysis
- `declare simd` functions are first converted into a loop over all vector elements and then passed to the loop vectorizer

```
1 #pragma omp declare simd  
2 float f(float x, float y) {  
3     return x + y;  
4 }
```

→

```
1 float8 f(float8 vx, float8 vy) {  
2     float8 r;  
3     for (i = 0; i < 8; i++) {  
4         float x = vx[i], y = vy[i]  
5         r[i] = x + y;  
6     }  
7     return r;  
8 }
```

- the range of vectorizable loops in a recent version I investigated (7.3.0) seems very limited
 - innermost loop with no conditionals
 - doubly nested loop with a very simple inner loop

Strategies for SpMV

- parallelize only across different rows (a single row is processed sequentially)
 - especially natural for CSR
 - extremely long rows may limit speedup
- parallelize all non-zeros, with careful handling of `y[i] +=`
 - atomic accumulation (`#pragma omp atomic`)
 - reduction (`#pragma omp reduction`). you must have user-defined reduction
- divide rows until the number of non-zeros becomes small (e.g., ≤ 5000)
 - further divide a single row if a row contains many zeros
 - can be done naturally with tasks