

# Basic OpenMP

Pangfeng Liu  
National Taiwan University

April 8, 2022

# OpenMP

- OpenMP <sup>1</sup> (Open Multiprocessing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms.

---

<sup>1</sup><http://openmp.org/>

# OpenMP

- Implemented as compiler preprocessor directive.
  - All OpenMP constructs can be safely ignored without affecting the semantic of the original program.
- Easy to understand and easy to use.
  - OpenMP is at a much high level than Pthread and handles all parallelization details.
- Promote “incremental” parallelism.
  - One can start with a working sequential program and “upgrade” it to a parallel program with minimum efforts.

# Hello, world

- Just like the case in Pthread, we start with a sequential program that prints hello world.
- Unlike the case in Pthread, we will only print it once.

# Hello World

## Example 1: Hello, world!

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

## #pragma omp

- #pragma omp is a preprocessor directive.
- All OpenMP directives start with #pragma omp.
- The effect of an OpenMP directive only applies to the *next* statement.
- This directive instructs the compiler to generate OpenMP parallel code if it supports OpenMP.
- This directive will be ignored by the compiler if OpenMP is not supported.

## #pragma omp parallel

- When we use gcc compilation option `-fopenmp` to compile the program, `#pragma omp parallel` will instructs the compiler to generate code to run the statement on *every* core.
  - For example, if the CPU has 4 cores, then `#pragma omp parallel` will spawn 4 threads, and each of them runs `printf` on a core.
- If we do not use `-fopenmp` then gcc will just generate a sequential code.

# Hello World

## Example 2: compilation

```
1 gcc hello.c -o hello-uni  
2 gcc -fopenmp hello.c -o hello-omp
```



# OpenMP

- The compiler can generate both the sequential and parallel code depending on the compilation flag.
- Incremental parallelism by having a working sequential code first, then try to parallelize it.

# Demonstration

- Run the sequential hello-uni program.
- Run the parallel hello-omp program.

# Implementation

- Implement a program that prints 10 lines of “hello” with OpenMP directive.

# Discussion

- Can you tell how many cores are there in your computer by running an OpenMP program?

## omp\_get\_num\_procs

- Sometimes it is very informative to know the number of cores (processing units) in the system.
- We can call `omp_get_num_procs` to know this information.
- Note that this is a function, not a compiler directive, therefore you need to include `<omp.h>` header file so that the compiler can check the prototype.

# Number of Processors

Prototype 3: Get the number of processors

```
1 int omp_get_num_procs(void);
```

## omp\_get\_num\_threads

- It is very informative to know the number of threads in the system.
- We can call `omp_get_num_threads` to know this information.
- Note that this is a function, not a compiler directive.

# Number of Threads

## Prototype 4: Get the number of threads

```
1 int omp_get_num_threads(void);
```

- Add this function to the hello world to know the number of threads running in the system.
- Compare the number with the number of processors.



## Get the Number

- Now we would like to get the number of threads and cores of our system.
- We simply add the function calls to `hello.c`.
- Note that we need to include `<omp.h>` header so that the compiler can check the function prototype.

# Hello World

## Example 5: (hello-get-num.c) Hello, world!

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6  #pragma omp parallel
7      printf("Hello, world.\n");
8      printf("# of proc = %d\n", omp_get_num_procs());
9      printf("# of threads = %d\n", omp_get_num_threads());
10     return 0;
11 }
```

# Demonstration

- Run the parallel hello-get-num-omp program.

# Discussion

- How many threads are reported?
- How many cores are reported?

# Problem

- We would like to get the correct number of threads.
- The problem with the previous program is that the effect of the directive applies only to the next statement, i.e., the `printf` for “hello, world”.
- As a result, when we call `omp_get_num_threads`, there is only one thread left.
- To solve this problem, we use a *compound* statement to enclose *everything* that we would like to run in parallel.

# Hello World

## Example 6: (hello-get-num-all.c) Hello, world!

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6  #pragma omp parallel
7      {
8      printf("Hello, world.\n");
9      printf("# of proc = %d\n", omp_get_num_procs());
10     printf("# of threads = %d\n", omp_get_num_threads());
11     }
12     return 0;
13 }
```

## Compound Statement

- The compound statement has three `printf` statements.
- Each of the threads created by the `parallel` directive will run all these three statements.

# Demonstration

- Run the parallel hello-get-num-omp program.



# Discussion

- How many threads are reported?
- How many cores are reported?

# In Control

- Sometimes, it is quite useful to set the number of threads, especially after we know the number of cores.
- The default number of threads is the number of cores, which is not necessarily suitable.
- We can call `omp_set_thread_num` to set the number of threads we want to create.

# Number of Threads

## Prototype 7: Set the number of threads

```
1 void omp_set_num_threads(int num_threads);
```

- Set the number of threads.

# Hello World

## Example 8: (hello-get-num-all-set.c) Hello, world!

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  int main(int argc, char *argv[])
7  {
8      assert(argc == 2);
9      omp_set_num_threads(atoi(argv[1]));
10 #pragma omp parallel
11 {
12     printf("Hello, world.\n");
13     printf("# of proc = %d\n", omp_get_num_procs());
14     printf("# of threads = %d\n", omp_get_num_threads());
15 }
16 return 0;
17 }
```

## Set the Number

- The number of threads is given as the second argument from the command line.
- Then the number of threads is used to call the `omp_set_num_threads` function.

# Demonstration

- Run the parallel hello-get-num-set-omp program.

# Discussion

- Does these output follow any time constrain order?

# Problem

- We cannot identify the threads because their outputs are all the same.
- It will be very useful that each thread can know its identity so that they can work accordingly.
- A thread can call `omp_get_thread_num` to know its index.



# Thread Number

## Prototype 9: Get thread number

```
1 int omp_get_thread_num(void);
```

# Hello World

## Example 10: (hello-get-num-all-set-show-id.c) Hello, world!

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  int main(int argc, char *argv[])
7  {
8      assert(argc == 2);
9      omp_set_num_threads(atoi(argv[1]));
10 #pragma omp parallel
11 {
12     printf("Hello, world from thread %d.",
13           omp_get_thread_num());
14     printf("# of proc = %d", omp_get_num_procs());
15     printf("# of threads = %d", omp_get_num_threads());
16 }
17 return 0;
18 }
```

## Get the Index

- We print the index of a thread by calling `omp_get_thread_num`.
- After knowing the index a thread can do things accordingly.
- The thread number starts with 0.

# Demonstration

- Run the parallel hello-get-num-set-show-id-omp program.

## Discussion

- Does these output follow any time constrain order?

# Loop

- Now we are ready to partition a loop.
- Previously we used `parallel` directive to instruct all threads to do the *same* thing.
- Now we use `parallel for` directive to ask all threads to *share* the workload of a for loop.

# A Parallel For Loop

## Pragma 11: Parallel for pragma

```
1 #pragma omp parallel for
```

# Semantics

- `parallel for` means that the loop will be distributed among all threads for execution.
- The OpenMP library takes care of all the threading details so you do not have to.
- There are restrictions on the condition of the loop.
  - The loop iteration can be determined at the compile time.



# Requirement

- The initialization must be `var = exp1`.
- The condition must be `var cond exp2`.
- The increment must be:
  - `var++`, `++var`.
  - `var--`, `--var`.
  - `var = var + exp3`, `var = var - exp3`.
  - `var += exp3`, `var -= exp3`.

# Steps

- Get the number of threads from the second command line argument.
- Get the number of for loop iterations from the third command line argument.
- Add the `parallel for` pragma to the loop.
- That is it!

# Loop

## Example 12: (for.c) A for loop

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5  int main(int argc, char *argv[])
6  {
7      assert(argc == 3);
8      omp_set_num_threads(atoi(argv[1]));
9      int n = atoi(argv[2]);
10     printf("# of proc = %d\n", omp_get_num_procs());
11     printf("# of loop iterations = %d\n", n);
12     #pragma omp parallel for
13     for (int i = 0; i < n; i++) {
14         printf("thread %d runs index %d.\n",
15             omp_get_thread_num(), i);
16     }
17     return 0;
18 }
```

# Synchronization

- Note that all threads, except the main thread, will only be active during the loop.
- All threads partitioning the loop will synchronize with each other, i.e., do a barrier synchronization before leaving the loop.
- This *join* is automatic, and we do not need to program anything.

# Demonstration

- Run the parallel for-omp program.

# Discussion

- Observe the output of for-omp and describe the rule about which thread executes which loop iterations.

## $n$ Queen with OpenMP

- Now we want to parallelize the  $n$ -queen program previously written for Pthread.
- We only need to parallel the loop in `main`, in which each iteration calls the `queen` we wrote previously.
- We can write the sequential version first, then parallelize it with OpenMP directives.

# Declaration

## Example 13: (queen.c) Declaration

```
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <assert.h>
5
6 #define MAXN 20
7 int n; /* a global n */
```



ok

## Example 14: (queen.c) main

```
9  int ok(int position[], int next, int test)
10 {
11     for (int i = 0; i < next; i++)
12         if (position[i] == test ||
13             (abs(test - position[i]) == next - i))
14             return 0;
15     return 1;
16 }
```

## queen

## Example 15: (queen.c) main

```
18 int queen(int position[], int next)
19 {
20     if (next >= n)
21         return 1;
22     int sum = 0;
23     for (int test = 0; test < n; test++)
24         if (ok(position, next, test)) {
25             position[next] = test;
26             sum += queen(position, next + 1);
27         }
28     return sum;
29 }
```

## main

## Example 16: (queen.c) main

```
31 int main (int argc, char *argv[])
32 {
33     assert(argc == 2);
34     n = atoi(argv[1]);
35     assert(n <= MAXN);
36
37     int position[MAXN];
38 #pragma omp parallel for
39     for (int i = 0; i < n; i++) {
40         position[0] = i;
41         printf("iteration %d # of solution = %d\n",
42             i, queen(position, 1));
43     }
44     return 0;
45 }
```

## main

- We only need to parallel the loop in `main` by adding `parallel for` directive.
- Each thread will call `queen` with `position[0]` set to different `i`, as we observed in the previous loop partition example.

# Demonstration

- Run and time the sequential queen-uni program.
- Run and time the parallel queen-omp program.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?

## $n$ Queen with OpenMP

- The problem with the previous program is that all threads share the `position` array.
- We now want to ask OpenMP to create a *private* variable of `position` in each thread, so they can produce the correct answer.
- We simply add a `private` clause to do this.
- Note that the private variable has nothing to do with the global variable – they are completely independent.

# Private Clause

## Prototype 17: Private clause

```
1 private (variables)
```

- A private clause is an optional component of a pragma.
- We can add this clause to a `parallel for` pragma.
- Note that the initial value of the private variable position has nothing to do with the global variable position.



## main

## Example 18: (queen-private.c) main

```
31 int main (int argc, char *argv[])
32 {
33     assert(argc == 2);
34     n = atoi(argv[1]);
35     assert(n <= MAXN);
36
37     #pragma omp parallel for
38     for (int i = 0; i < n; i++) {
39         int position[MAXN];
40         position[0] = i;
41         printf("iteration %d # of solution = %d\n",
42             i, queen(position, 1));
43     }
44     return 0;
45 }
```

# Demonstration

- Run and time the sequential queen-private-uni program.
- Run and time the parallel queen-private-omp program.
- Calculate the speedup.

# Implementation

- Implement a program that solves the  $n$  queen problem with OpenMP directive.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?

# Private Variable

- We use the following example to emphasize that a private variable has nothing to do with the global variable.
- One can think of the private variable as a new variable declared with the thread with the same name.

# parallel for

## Example 19: (for-private.c) main

```
9  assert(argc == 3);
10  omp_set_num_threads(atoi(argv[1]));
11  int n = atoi(argv[2]);
12  printf("# of proc = %d\n", omp_get_num_procs());
13  printf("# of loop iterations = %d\n", n);
14  int v = 101;
15  printf("before the loop thread %d with v = %d.\n",
16        omp_get_thread_num(), v);
17  #pragma omp parallel for private(v)
18  for (int i = 0; i < n; i++) {
19      printf("thread %d runs index %d with v = %d.\n",
20            omp_get_thread_num(), i, v);
21  }
22  printf("after the loop thread %d with v = %d.\n",
23        omp_get_thread_num(), v);
24  return 0;
```

# Demonstration

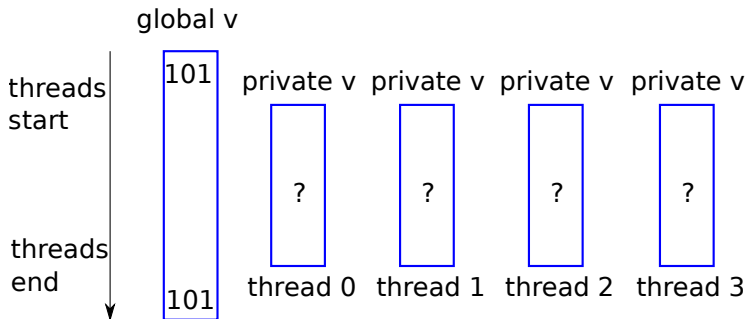
- Run for-private-omp and observe the output.

## Independent variables

- We can clearly see that the values of the private  $v$  have nothing to do with the value of the global  $v$ .
- The value of the global  $v$  is not affected by any operations within the loop.



# Private Variables



# Discussion

- Check queen-private.c and make sure that you understand that even though the global position and private position are independent, the code still works.
- What will happen if we declare `position` *within* the loop?

# Sum

- We now want to compute the total number of solutions.
- We use a variable `num` to store the number of solutions from one iteration and a variable `numSolution` to store the total number of solutions.
- The `parallel for` directive applies only to the `for` loop, so the `printf` after the loop will only run in the main thread.

## main

## Example 20: (queen-private-sum.c) main

```
37  int position[MAXN];
38  int numSolution = 0;
39  int num;
40  #pragma omp parallel for private (position)
41  for (int i = 0; i < n; i++) {
42      position[0] = i;
43      num = queen(position, 1);
44      printf("iteration %d # of solution = %d\n",
45            i, num);
46      numSolution += num;
47  }
48  printf("total # of solutions = %d\n", numSolution);
49  return 0;
50 }
```

# Demonstration

- Run the sequential queen-private-sum-uni program.
- Run the parallel queen-private-sum-omp program.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?

# Problem

- The problem with the previous program is that we forgot to declare `num` as `private`, so all threads use the same copy.
- We simply add `num` to the `private` clause and fix the problem.
- Note that the `numSolution` needs to remain `global` since it is the total number.

## main

## Example 21: (queen-private-sum-num.c) main

```
37  int position[MAXN];
38  int numSolution = 0;
39  int num;
40  #pragma omp parallel for private (position, num)
41  for (int i = 0; i < n; i++) {
42      position[0] = i;
43      num = queen(position, 1);
44      printf("iteration %d # of solution = %d\n",
45            i, num);
46      numSolution += num;
47  }
48  printf("total # of solutions = %d\n", numSolution);
49  return 0;
50 }
```



# Demonstration

- Run and time the sequential queen-private-sum-uni program.
- Run and time the parallel queen-private-sum-omp program.
- Calculate the speedup.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?
- Does the parallel program *a/ways* produce correct answer?

# Problem

- To illustrate that there is also a race condition in the previous program we “amplify” it by adding to a global variable `numSolution`.
- Every time queen finds a solution, it adds 1 to `numSolution`.
- Finally the main program prints `numSolution`

# Declaration

Example 22: (queen-private-sum-race.c) main

```
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <assert.h>
5
6  #define MAXN 20
7  int n; /* a global n */
8  int numSolution;
```

## queen

## Example 23: (queen-private-sum-race.c) main

```
19 int queen(int position[], int next)
20 {
21     if (next >= n)
22         numSolution++;
23     for (int test = 0; test < n; test++)
24         if (ok(position, next, test)) {
25             position[next] = test;
26             queen(position, next + 1);
27         }
28 }
```

## main

## Example 24: (queen-private-sum-race.c) main

```
36  int position[MAXN];
37  #pragma omp parallel for private (position)
38  for (int i = 0; i < n; i++) {
39      position[0] = i;
40      queen(position, 1);
41  }
42  printf("total # of solutions = %d\n",
43        numSolution);
44  return 0;
45 }
```

# Demonstration

- Run the sequential queen-private-sum-uni program.
- Run the parallel queen-private-sum-omp program.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?



# Problem

- To avoid the extremely rare race condition in queen-private-sum we use an array `num` to store the number of solutions from each subtree.
- Later we sequentially sum the numbers in `num` to get the correct total number of solutions.

## main

## Example 25: (queen-private-sum-array.c) main

```
37  int position[MAXN];
38  int num[MAXN] = {0};
39  #pragma omp parallel for private (position)
40  for (int i = 0; i < n; i++) {
41      position[0] = i;
42      num[i] = queen(position, 1);
43      printf("iteration %d # of solution = %d\n",
44             i, num[i]);
45  }
46  int numSolution = 0;
47  for (int i = 0; i < n; i++)
48      numSolution += num[i];
49  printf("total # of solutions = %d\n", numSolution);
50  return 0;
51 }
```

# Demonstration

- Run and time the sequential queen-private-sum-array-uni program.
- Run and time the parallel queen-private-sum-array-omp program.

# Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?

# Reduction

- It is very common that we need to collect answers from all threads and combine them.
- OpenMP provides a *simple* mechanism called *reduction* to derive the final answer by combining many values into one.

# Reduction Clause

Prototype 26: Reduction clause of a parallel for pragma

1 `reduction (operation : variable)`

**operation** Operation to be performed on the partial answers.

**variable** The partial answer the operation to be perform on.

# Reduction Operation

The reduction supports the following operations.

- + Sum
- \* Product
- & Bitwise and
- | Bitwise or
- && Logical and
- || Logical or

## main

## Example 27: (queen-private-sum-reduction.c) main

```
37  int position[MAXN];  
38  int numSolution = 0;  
39  #pragma omp parallel for private (position) \  
40      reduction(+ : numSolution)  
41  for (int i = 0; i < n; i++) {  
42      position[0] = i;  
43      int num = queen(position, 1);  
44      printf("iteration %d # of solution = %d\n",  
45          i, num);  
46      numSolution += num;  
47  }  
48  printf("total # of solutions = %d\n", numSolution);  
49  return 0;  
50 }
```



# Notes

- The variable for reduction `numSolution` is *private*.
- We want to compute the total number of solutions so we use `+` operator.
- The variable `num` is declared within the loop so it is private to its thread.

# Demonstration

- Run the sequential queen-private-sum-reduction-uni program.
- Run the parallel queen-private-sum-reduction-omp program.

# Discussion

- Does the sequential program produce the correct answer?
- Does the parallel program produce the correct answer?
- Can you derive the characteristic of the operations supported by reduction? What do they have in common?

# Communication

- Remember that the global variable and the private variables are different things.
- In the previous program we initialize the global variable `numSolution` to 0, but how do all private variables `numSolution` knows this initial value???
- We need to concept of `firstprivate`.

# firstprivate

- Usually the private and global variables are different things.
- Sometimes we want to “pass” the value of the global variable to threads as the *initial* values of their private variables.
- We can use a `firstprivate` clause to do this.

# firstprivate

## Example 28: (for-private-first.c) main

```
9  assert(argc == 3);
10  omp_set_num_threads(atoi(argv[1]));
11  int n = atoi(argv[2]);
12  printf("# of proc = %d\n", omp_get_num_procs());
13  printf("# of loop iterations = %d\n", n);
14  int v = 101;
15  printf("before the loop thread %d with v = %d.\n",
16        omp_get_thread_num(), v);
17  #pragma omp parallel for firstprivate(v)
18  for (int i = 0; i < n; i++) {
19      v += i;
20      printf("thread %d runs index %d with v = %d.\n",
21            omp_get_thread_num(), i, v);
22  }
23  printf("after the loop thread %d with v = %d.\n",
24        omp_get_thread_num(), v);
```

# firstprivate

- The first command-line argument specifies the number of iterations.
- The second command-line argument specifies the number of threads.
- The value of global  $v$  is used to initialize the private  $v$  when the threads starts.
- Note that the initialization only happens when a thread starts, not every loop iteration.

# Demonstration

- Run the loop-private-first-omp program with four threads and eight iterations, and observe the output.



global v

private v   private v   private v   private v

threads  
start

threads  
end

Thread	Global v	Private v
thread 0	101	102
thread 1	101	106
thread 2	101	110
thread 3	101	114

i = 0, 1   i = 2, 3   i = 4, 5   i = 6, 7

# Communication

- `firstprivate` provides a mechanism to pass information into threads.
- We can also do this by placing the information into a global variable.
- If the information will be manipulated and modified by each thread individually, then we can use `firstprivate` to make a private copy within each thread, which is very convenient.

# Discussion

- Describe when and the number of times the private variables are initialized.
- Describe how to implement the part of a reduction that we need to pass the initial values of `numSolution` in `queen-private-sum-reduction.c`.

# Sudoku

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

# Sudoku

- There is a nine by nine board with numbers from 1 to 9 already on it (in black).
- Place numbers from 1 to 9 (in red) into empty squares so that every row, every column, and all nine three by three squares will have numbers 1 to 9.
- We would like to compute the total number of solutions.

## main

## Example 29: (sudoku.c) main

```
54 int main(void)
55 {
56     int sudoku[9][9];
57     int firstZero = -1;
58     for (int i = 0; i < 9; i++)
59         for (int j = 0; j < 9; j++) {
60             scanf("%d", &(sudoku[i][j]));
61             if (sudoku[i][j] == 0 && firstZero == -1)
62                 firstZero = i * 9 + j;
63         }
```

# Solution

- We represent the board by a nine by nine array of integers sudoku.
- The input has all the initial numbers on the board, where an 0 is an empty cell.
- We need to remember the index of the first zero (`firstZero`) so that we can start the recursion from there.

## main

## Example 30: (sudoku.c) main

```
65 #ifdef _OPENMP
66     omp_set_num_threads(9);
67 #endif
68     int numSolution = 0;
69 #pragma omp parallel for reduction(+ : numSolution) \
70     firstprivate(sudoku)
71     for (int i = 1; i <= 9; i++) {
72         if (!conflict(i, firstZero / 9, firstZero % 9, sudoku))
73             sudoku[firstZero / 9][firstZero % 9] = i;
74             numSolution += placeNumber(firstZero, sudoku);
75     }
76 }
77 printf("# of solution = %d\n", numSolution);
78 return 0;
79 }
```



# Compile

- We explicitly set the number of threads to 9 for the OpenMP version.
- However, we also want to use the same code to compile a sequential version.
- The solution is test if `_OPENMP` is defined, which means the `-fopenmp` flag is used, then we will compile the `omp_set_num_threads` call.
- You may use `-E` to verify that the correct code is compiled.

# Solution

- After we read the inputs into `sudoku`, we then use `firstprivate` to send the input to all threads.
- We use nine iterations to compute the total number of solutions, where the  $i$ -th iteration computes the number of solutions if we place the first empty cell with  $i$ .
- An iteration first tries to place a number into the first empty cell if there is no conflict, then call `placeNumber` to compute the number of solutions.
- We use a reduction on the variable `numSolution`, which is the number of solutions found by a recursive function `placeNumber`.

# conflict

## Example 31: (sudoku.c) conflict

```
28 int conflict(int try, int row, int col, int sudoku[9][9])  
29 {  
30     return (rowColConflict(try, row, col, sudoku) ||  
31             blockConflict(try, row, col, sudoku));  
32 }
```

# rowColConflict

## Example 32: (sudoku.c) rowColConflict

```
4  int rowColConflict(int try, int row, int col, int sudoku[9][9])
5  {
6      int conflict = 0;
7      for (int i = 0; i < 9 && !conflict; i++)
8          if (((col != i) && (sudoku[row][i] == try)) ||
9              ((row != i) && (sudoku[i][col] == try)))
10             conflict = 1;
11     return conflict;
12 }
```

# blockConflict

## Example 33: (sudoku.c) blockConflict

```
14 int blockConflict(int try, int row, int col, int sudoku[9][9])
15 {
16     int blockRow = row / 3;
17     int blockCol = col / 3;
18
19     int conflict = 0;
20     for (int i = 0; i < 3 && !conflict; i++)
21         for (int j = 0; j < 3 && !conflict; j++)
22             if (sudoku[3 * blockRow + i][3 * blockCol + j]
23                 == try)
24                 conflict = 1;
25     return conflict;
26 }
```

# placeNumber

- The function `placeNumber` tries placing a number at the  $n$ -th position on the board `sudoku`, where `n` goes from 0 to 80.
- If `n` is 81 then we have placed all numbers and a solution is found.
- Otherwise we determine the row and column of this cell. If this cell has a number, then we skip it since it is a part of the input.

# placeNumber

## Example 34: (sudoku.c) placeNumber

```
34 int placeNumber(int n, int sudoku[9][9])
35 {
36     if (n == 81)
37         return 1;
38     int row = n / 9;
39     int col = n % 9;
40     if (sudoku[row][col] != 0)
41         return(placeNumber(n + 1, sudoku));
```

## placeNumber

- If there is no number in the cell, then we check if we can place a number try into this cell by calling `conflict`. If we can, then we go to the next level of recursion.
- Since we found no conflict, we place the number into the cell and call `placeNumber` to go for the next cell.
- After we tried all numbers from 1 to 9, we reported the number of solutions we found.
- Note that we need to reset the cell back to 0 so later recursion will not mistake it for being an input.



# placeNumber

## Example 35: (sudoku.c) placeNumber

```
43  int numSolution = 0;
44  for (int try = 1; try <= 9; try++) {
45      if (!conflict(try, row, col, sudoku)) {
46          sudoku[row][col] = try;
47          numSolution += placeNumber(n + 1, sudoku);
48      }
49  } /* for */
50  sudoku[row][col] = 0;
51  return numSolution;
52 }
```

# Demonstration

- Run and time the sudoku-uni program.
- Run and time the sudoku-omp program.
- Compute the speedup.

# Discussion

- Do you think the sudoku program balance the workload evenly? Explain your answer.
- Is there any way to balance the workload better?

# Communication

- Sometimes we do need to pass the value of one private variable back to its global counterpart.
- The `lastprivate` clause will pass the value of private variable of the thread which runs the *last* iteration.
- Note that this is not the thread that runs last – it is the thread that runs the *last* iteration.

global v

private v private v private v private v

threads start

threads end

thread 0 thread 1 thread 2 thread 3

$i = 0, 1$   $i = 2, 3$   $i = 4, 5$   $i = 6, 7$

# lastprivate

## Example 36: (for-private-first-last.c) main

```
14  int v = 101;
15  printf("before the loop thread %d with v = %d.\n",
16        omp_get_thread_num(), v);
17  #pragma omp parallel for firstprivate(v) \
18    lastprivate(v)
19    for (int i = 0; i < n; i++) {
20      v += i;
21      printf("thread %d runs index %d with v = %d.\n",
22            omp_get_thread_num(), i, v);
23    }
24  printf("after the loop thread %d with v = %d.\n",
25        omp_get_thread_num(), v);
26  return 0;
27 }
```

# Demonstration

- Run the loop-private-first-last-omp program with four threads and eight iterations, and observe the output.

# Discussion

- Describe a scenario that `lastprivate` will be useful.



# Critical Section

- It is very common that we need to implement a critical section that only one thread can access at a time.
- OpenMP provides a *critical* pragma to implement a critical section.
- A `critical` directive will make the following statement a critical section.

# Critical Pragma

## Prototype 37: Critical pragma

```
1 #pragma omp critical
```

## queen

## Example 38: (queen-private-sum-race-critical.c) main

```
19 int queen(int position[], int next)
20 {
21     if (next >= n)
22 #pragma omp critical
23     numSolution++;
24     for (int test = 0; test < n; test++)
25         if (ok(position, next, test)) {
26             position[next] = test;
27             queen(position, next + 1);
28         }
29 }
```

## main

## Example 39: (queen-private-sum-race-critical.c) main

```
37  int position[MAXN];
38  #pragma omp parallel for private (position)
39  for (int i = 0; i < n; i++) {
40      position[0] = i;
41      queen(position, 1);
42  }
43  printf("total # of solutions = %d\n",
44        numSolution);
45  return 0;
46  }
```

# Notes

- Whenever queen finds a solution, it adds 1 to numSolution.
- This addition is a critical section because we put a `#pragma omp critical` in front of it.
- If you want to place multiple statements into a critical section, then you need to put them into a compound statement.

# Demonstration

- Run the sequential queen-private-sum-critical-uni program.
- Run the parallel queen-private-sum-critical-omp program.

## Discussion

- Does the sequential program produce correct answer?
- Does the parallel program produce correct answer?
- Time and compare the correct solutions you know so far.

## The Big Question

Do you think it is easier to write  
OpenMP program than Pthread  
program?