Basic OpenMP

Pangfeng Liu
National Taiwan University

April 24, 2020

OpenMP

- OpenMP [1] (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.

[1] 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!

```
#include <stdio.h>

int main(void)
{
#pragma omp parallel
  printf("Hello, world.\n");
  return 0;
}
```

#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

```
gcc hello.c -o hello-uni
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

```
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

```
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!

```
#include <omp.h>
#include <stdio.h>

int main(void)
{
#pragma omp parallel
  printf("Hello, world.\n");
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of threads = %d\n", omp_get_num_threads());
  return 0;
}
```

Demonstration

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

Discussion

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

Problem

- Now 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!

```
#include <omp.h>
#include <stdio.h>

int main(void)
{
#pragma omp parallel
  {
    printf("Hello, world.\n");
    printf("# of proc = %d\n", omp_get_num_procs());
    printf("# of threads = %d\n", omp_get_num_threads());
  }
  return 0;
}
```

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

```
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!

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[])
{
  assert(argc == 2);
  omp_set_num_threads(atoi(argv[1]));
#pragma omp parallel
  {
    printf("Hello, world.\n");
    printf("# of proc = %d\n", omp_get_num_procs());
    printf("# of threads = %d\n", omp_get_num_threads());
  }
  return 0;
}
```

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

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

Hello World

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

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int main(int argc, char *argv[])
{
  assert(argc == 2);
  omp_set_num_threads(atoi(argv[1]));
#pragma omp parallel
  {
    printf("Hello, world from thread %d.\n",
           omp_get_thread_num());
    printf("# of proc = %d\n", omp_get_num_procs());
    printf("# of threads = %d\n", omp_get_num_threads());
  }
  return 0;
}
```

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

```
#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

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
int main(int argc, char *argv[])
{
  assert(argc == 3);
  omp_set_num_threads(atoi(argv[1]));
  int n = atoi(argv[2]);
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of loop iterations = %d\n", n);
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    printf("thread %d runs index %d.\n",
           omp_get_thread_num(), i);
  }
  return 0;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAXN 20
int n;  /* a global n */
```

ok

Example 14: (queen.c) main

```
int ok(int position[], int next, int test)
{
  for (int i = 0; i < next; i++)
    if (position[i] == test ||
        (abs(test - position[i]) == next - i))
      return 0;
  return 1;
}
```

queen

Example 15: (queen.c) main

```
int queen(int position[], int next)
{
  if (next >= n)
    return 1;
  int sum = 0;
  for (int test = 0; test < n; test++)
    if (ok(position, next, test)) {
      position[next] = test;
      sum += queen(position, next + 1);
    }
  return sum;
}
```

main

Example 16: (queen.c) main

```
int main (int argc, char *argv[])
{
  assert(argc == 2);
  n = atoi(argv[1]);
  assert(n <= MAXN);

  int position[MAXN];
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    position[0] = i;
    printf("iteration %d # of solution = %d\n",
           i, queen(position, 1));
  }
  return 0;
}
```

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

```
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

```c
int main (int argc, char *argv[])
{
  assert(argc == 2);
  n = atoi(argv[1]);
  assert(n <= MAXN);

  int position[MAXN];
#pragma omp parallel for private (position)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    printf("iteration %d # of solution = %d\n",
           i, queen(position, 1));
  }
  return 0;
}
```

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

```
  assert(argc == 3);
  omp_set_num_threads(atoi(argv[1]));
  int n = atoi(argv[2]);
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of loop iterations = %d\n", n);
  int v = 101;
  printf("before the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
#pragma omp parallel for private(v)
  for (int i = 0; i < n; i++) {
    printf("thread %d runs index %d with v = %d.\n",
           omp_get_thread_num(), i, v);
  }
  printf("after the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
  return 0;
```

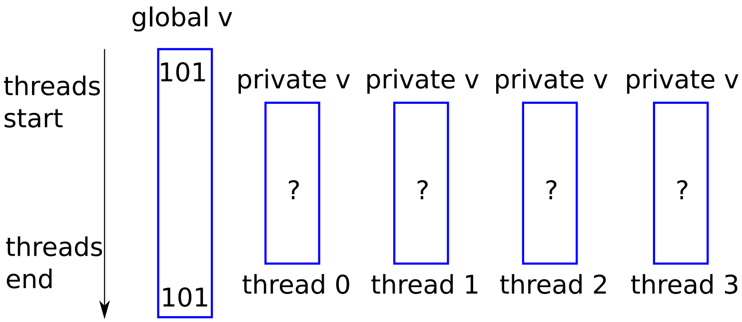
Demonstration

- 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 be executed by the main thread.

main

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

```
  int position[MAXN];
  int numSolution = 0;
  int num;
#pragma omp parallel for private (position)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    num = queen(position, 1);
    printf("iteration %d # of solution = %d\n",
           i, num);
    numSolution += num;
  }
  printf("total # of solutions = %d\n", numSolution);
  return 0;
}
```

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

```
  int position[MAXN];
  int numSolution = 0;
  int num;
#pragma omp parallel for private (position, num)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    num = queen(position, 1);
    printf("iteration %d # of solution = %d\n",
           i, num);
    numSolution += num;
  }
  printf("total # of solutions = %d\n", numSolution);
  return 0;
}
```

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 *always* 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

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAXN 20
int n;   /* a global n */
int numSolution;
```

queen

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

```
int queen(int position[], int next)
{
  if (next >= n)
    numSolution++;
  for (int test = 0; test < n; test++)
    if (ok(position, next, test)) {
      position[next] = test;
      queen(position, next + 1);
    }
}
```

main

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

```
  int position[MAXN];
#pragma omp parallel for private (position)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    queen(position, 1);
  }
  printf("total # of solutions = %d\n",
         numSolution);
  return 0;
}
```

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

```
  int position[MAXN];
  int num[MAXN] = {0};
#pragma omp parallel for private (position)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    num[i] = queen(position, 1);
    printf("iteration %d # of solution = %d\n",
           i, num[i]);
  }
  int numSolution = 0;
  for (int i = 0; i < n; i++)
    numSolution += num[i];
  printf("total # of solutions = %d\n", numSolution);
  return 0;
}
```

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

```
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

```
  int position[MAXN];
  int numSolution = 0;
#pragma omp parallel for private (position) \
  reduction(+ : numSolution)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    int num = queen(position, 1);
    printf("iteration %d # of solution = %d\n",
           i, num);
    numSolution += num;
  }
  printf("total # of solutions = %d\n", numSolution);
  return 0;
}
```

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 correct answer?
- Does the parallel program produce 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

```
  assert(argc == 3);
  omp_set_num_threads(atoi(argv[1]));
  int n = atoi(argv[2]);
  printf("# of proc = %d\n", omp_get_num_procs());
  printf("# of loop iterations = %d\n", n);
  int v = 101;
  printf("before the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
#pragma omp parallel for firstprivate(v)
  for (int i = 0; i < n; i++) {
    v += i;
    printf("thread %d runs index %d with v = %d.\n",
           omp_get_thread_num(), i, v);
  }
  printf("after the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
```

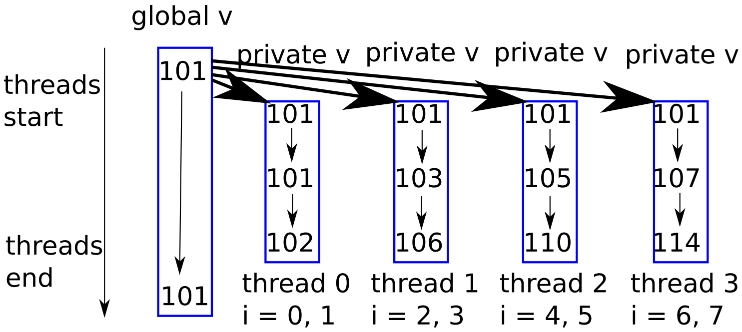
firstprivate

- 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.

First Private Variables



Communication

- `firstprivate` provides a mechanism to pass information into threads.
- We can also do this by placing the information into a global variable.
- However, 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. This 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-redction.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 9 by 9 board with some numbers from 1 to 9 already in it (in black).
- Place numbers from 1 to 9 (in red) into empty squares, so that the every row, every column, and all nine 3 by 3 squares will have numbers 1 to 9.
- We would like to compute the total number of solutions.

main

Example 29: (sudoku.c) main

```
int main(void)
{
  int sudoku[9][9];
  int firstZero = -1;
  for (int i = 0; i < 9; i++)
    for (int j = 0; j < 9; j++) {
      scanf("%d", &(sudoku[i][j]));
      if (sudoku[i][j] == 0 && firstZero == -1)
        firstZero = i * 9 + j;
    }
```

Solution

- We represent the board by a 9 by 9 array of integers `sudoku`.
- The input has all the initial numbers in 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

```
#ifdef _OPENMP
  omp_set_num_threads(9);
#endif
  int numSolution = 0;
#pragma omp parallel for reduction(+ : numSolution) \
  firstprivate(sudoku)
  for (int i = 1; i <= 9; i++) {
    if (!conflict(i, firstZero / 9, firstZero % 9, sudoku)) {
      sudoku[firstZero / 9][firstZero % 9] = i;
      numSolution += placeNumber(firstZero, sudoku);
    }
  }
  printf("# of solution = %d\n", numSolution);
  return 0;
}
```

Compile

- We explicitly set the number of threads to 9 for the OpenMP version.
- However, we also want to use to same code to compiler a sequential version.
- The solution is test if `_OPENMP` is defined, which means the `-fopenmp` flag is used, then we will compiler 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 he 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

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

rowColConflict

Example 32: (sudoku.c) `rowColConflict`

```c
int rowColConflict(int try, int row, int col, int sudoku[9][9])
{
  int conflict = 0;
  for (int i = 0; i < 9 && !conflict; i++)
    if (((col != i) && (sudoku[row][i] == try)) ||
        ((row != i) && (sudoku[i][col] == try)))
      conflict = 1;
  return conflict;
}
```

blockConflict

Example 33: (sudoku.c) blockConflict

```
int blockConflict(int try, int row, int col, int sudoku[9][9])
{
  int blockRow = row / 3;
  int blockCol = col / 3;

  int conflict = 0;
  for (int i = 0; i < 3 && !conflict; i++)
    for (int j = 0; j < 3 && !conflict; j++)
      if (sudoku[3 * blockRow + i][3 * blockCol + j]
          == try)
        conflict = 1;
  return conflict;
}
```

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

```
int placeNumber(int n, int sudoku[9][9])
{
  if (n == 81)
    return 1;
  int row = n / 9;
  int col = n % 9;
  if (sudoku[row][col] != 0)
    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 no conflict was found 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 report 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

```
  int numSolution = 0;
  for (int try = 1; try <= 9; try++) {
    if (!conflict(try, row, col, sudoku)) {
      sudoku[row][col] = try;
      numSolution += placeNumber(n + 1, sudoku);
    }
  } /* for */
  sudoku[row][col] = 0;
  return numSolution;
}
```

Demonstration

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

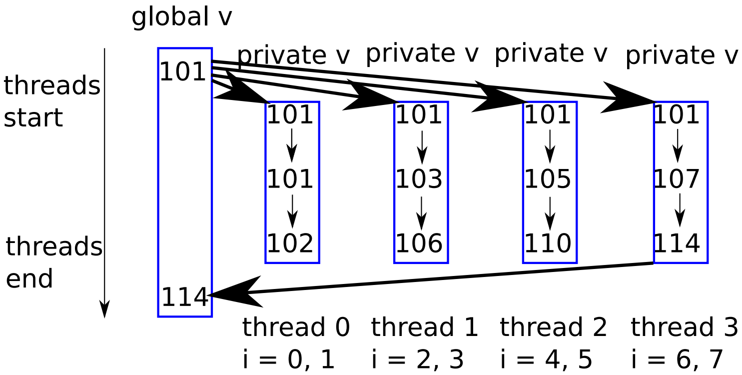
Discussion

- 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.

Both firstprivate and lastprivate



lastprivate

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

```
  int v = 101;
  printf("before the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
#pragma omp parallel for firstprivate(v) \
  lastprivate(v)
  for (int i = 0; i < n; i++) {
    v += i;
    printf("thread %d runs index %d with v = %d.\n",
           omp_get_thread_num(), i, v);
  }
  printf("after the loop thread %d with v = %d.\n",
         omp_get_thread_num(), v);
  return 0;
}
```

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

```
#pragma omp critical
```

queen

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

```
int queen(int position[], int next)
{
  if (next >= n)
#pragma omp critical
    numSolution++;
  for (int test = 0; test < n; test++)
    if (ok(position, next, test)) {
      position[next] = test;
      queen(position, next + 1);
    }
}
```

main

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

```
  int position[MAXN];
#pragma omp parallel for private (position)
  for (int i = 0; i < n; i++) {
    position[0] = i;
    queen(position, 1);
  }
  printf("total # of solutions = %d\n",
         numSolution);
  return 0;
}
```

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?