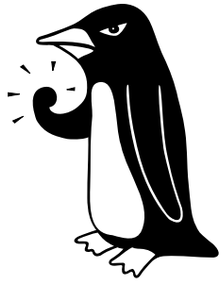
Back to the Test Problem

“The Angry Penguin“, used under creative commons licence from Swantje Hess and Jannis Pohlmann.



Warwick RSE

14/12/2020

Test Problem

- Remember the test problem?
  - Blurring an image by averaging neighbouring pixels
- We do now have enough OpenMP to parallelise this code
- Isn’t that different to the previous codes
  - But now is an actual, if simple, problem

Test Problem

```
SUBROUTINE blur(image, its)

  INTEGER, DIMENSION(0:,0:), INTENT(INOUT) :: image
  INTEGER, INTENT(IN) :: its
  INTEGER :: ix, iy, iit, UB1, UB2
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: temp

  ALLOCATE(temp, SOURCE = image)

  UB1 = UBOUND(image,2) - 1
  UB2 = UBOUND(image,1) - 1

  DO iit = 1, its
    DO iy = 1, UB1
      DO ix = 1, UB2
        temp(ix,iy) = INT(0.25 * REAL(image(ix+1,iy) + image(ix-1,iy) &
            + image(ix,iy+1) + image(ix,iy-1)))
      END DO
    END DO
    image = temp
  END DO

END SUBROUTINE blur
```

Test Problem

```
void blur(int *image, size_t szx, size_t szy, size_t iters){

  int *temp = malloc(sizeof(int) * szx * szy);
  for (size_t i = 0; i< NX*NY; ++i) temp[i]=image[i];

  for (size_t iit = 0; iit < iters; ++iit){
    for (size_t ix = 1; ix< szx-1; ++ix){
      for (size_t iy = 1; iy< szy-1; ++iy){
        temp[iy + ix * szy] = (int)(0.25 * (float)(image[iy + (ix+1) * szy] +
                image[iy + (ix-1) * szy] + image[(iy-1) + ix * szy] +
                image[(iy+1) + ix * szy]) + 0.5);
      }
    }

    for (size_t i = 0; i < (szx * szy); ++i){
      image[i] = temp[i];
    }
  }

  free(temp);

}
```

Test Problem

- So the idea is that you want to run over every element of the array and calculate the new value for every element

- Because elements do not interact you can do it fully in parallel

- At the simplest level, this is done exactly how you'd think it is

Simple Parallel

```
SUBROUTINE blur(image, its)

  INTEGER, DIMENSION(0:,0:), INTENT(INOUT) :: image
  INTEGER, INTENT(IN) :: its
  INTEGER :: ix, iy, iit, UB1, UB2
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: temp

  ALLOCATE(temp, SOURCE = image)

  UB1 = UBOUND(image,2) - 1
  UB2 = UBOUND(image,1) - 1

    DO iit = 1, its
!$OMP PARALLEL DO
      DO iy = 1, UB1
        DO ix = 1, UB2
          temp(ix,iy) = INT(0.25 * REAL(image(ix+1,iy) + image(ix-1,iy) &
              + image(ix,iy+1) + image(ix,iy-1)))
        END DO
      END DO
!$OMP END PARALLEL DO
      image = temp
    END DO

  END SUBROUTINE blur
END MODULE blurmod
```

Simple Parallel

```
void blur(int *image, size_t szx, size_t szy, size_t iters){

  int *temp = malloc(sizeof(int) * szx * szy);
  for (size_t i = 0; i< NX*NY; ++i) temp[i]=image[i];

  for (size_t iit = 0; iit < iters; ++iit){
#pragma omp parallel for
    for (size_t ix = 1; ix< szx-1; ++ix){
      for (size_t iy = 1; iy< szy-1; ++iy){
        temp[iy + ix * szy] = (int)(0.25 * (float)(image[iy + (ix+1) * szy] +
            image[iy + (ix-1) * szy] + image[(iy-1) + ix * szy] +
            image[(iy+1) + ix * szy]) + 0.5);
      }
    }
    for (size_t i = 0; i < (szx * szy); ++i){
      image[i] = temp[i];
    }
  }
  free(temp);
}
```

Does it work?

- Pretty much the only advantage of OpenMP parallelism is speed, so you would usually check if your code is working by checking speed
    - Here tested on a machine with dual Intel Xeon E5-2680 v4 chips for a maximum of 28 physical CPUs

Processors	Time
1	13.1
2	10.1
4	7.6
8	7.0
16	7.3
28	7.8

Does it work?

- Well, sort of
- It is clearly faster than the serial version but it doesn’t scale as hoped
- You’d hope that doubling the number of processor should halve the runtime
- Have we done something wrong?
    - Not wrong, just incomplete

Amdahl's Law

- Formally, imperfect scaling is described by Amdahl's law.

- Part of code is parallel, worked on separately by all processors

- Part is serial, where all processors have to go one by one

Amdahl's Law

- p is fraction of code which is parallel
- s is the number of cores
- S is the resulting speedup
- $S = 1/ (1-p + (p/s))$

Simple Parallel

```
SUBROUTINE blur(image, its)

  INTEGER, DIMENSION(0:,0:), INTENT(INOUT) :: image
  INTEGER, INTENT(IN) :: its
  INTEGER :: ix, iy, iit, UB1, UB2
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: temp

  ALLOCATE(temp, SOURCE = image)

  UB1 = UBOUND(image,2) - 1
  UB2 = UBOUND(image,1) - 1

    DO iit = 1, its
!$OMP PARALLEL DO
      DO iy = 1, UB1
        DO ix = 1, UB2
          temp(ix,iy) = INT(0.25 * REAL(image(ix+1,iy) + image(ix-1,iy) &
              + image(ix,iy+1) + image(ix,iy-1)))
        END DO
      END DO
!$OMP END PARALLEL DO
      image = temp
    END DO

  END SUBROUTINE blur
END MODULE blurmod
```

Simple Parallel

```
void blur(int *image, size_t szx, size_t szy, size_t iters){

  int *temp = malloc(sizeof(int) * szx * szy);
  for (size_t i = 0; i< NX*NY; ++i) temp[i]=image[i];

  for (size_t iit = 0; iit < iters; ++iit){
#pragma omp parallel for
    for (size_t ix = 1; ix< szx-1; ++ix){
      for (size_t iy = 1; iy< szy-1; ++iy){
        temp[iy + ix * szy] = (int)(0.25 * (float)(image[iy + (ix+1) * szy] +
            image[iy + (ix-1) * szy] + image[(iy-1) + ix * szy] +
            image[(iy+1) + ix * szy]) + 0.5);
      }
    }
    for (size_t i = 0; i < (szx * szy); ++i){
      image[i] = temp[i];
    }
  }
  free(temp);
}
```

What's the problem?

- So what's happening is that you're speeding up the blur operation

- But you're not speeding up the bit where you copy the blurred image back

    - Sooner or later this becomes the limiting step

- There are other things in this code that can slow it down but let's try this to start with

Parallel Copy

```
SUBROUTINE blur(image, its)

  INTEGER, DIMENSION(0:,0:), INTENT(INOUT) :: image
  INTEGER, INTENT(IN) :: its
  INTEGER :: ix, iy, iit, UB1, UB2
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: temp

  ALLOCATE(temp, SOURCE = image)

  UB1 = UBOUND(image,2) - 1
  UB2 = UBOUND(image,1) - 1
```

```
    DO iit = 1, its
!$OMP PARALLEL
!$OMP DO
      DO iy = 1, UB1
        DO ix = 1, UB2
          temp(ix,iy) = INT(0.25 * REAL(image(ix+1,iy) + image(ix-1,iy) &
              + image(ix,iy+1) + image(ix,iy-1)))
        END DO
      END DO
!$OMP END DO
!$OMP WORKSHARE
      image = temp
!$OMP END WORKSHARE
!$OMP END PARALLEL
    END DO

  END SUBROUTINE blur
```

Parallel Copy

```
void blur(int *image, size_t szx, size_t szy, size_t iters){

  int *temp = malloc(sizeof(int) * szx * szy);
  for (size_t i = 0; i< NX*NY; ++i) temp[i]=image[i];

  for (size_t iit = 0; iit < iters; ++iit){
#pragma omp parallel
{
#pragma omp for
    for (size_t ix = 1; ix< szx-1; ++ix){
      for (size_t iy = 1; iy< szy-1; ++iy){
        temp[iy + ix * szy] = (int)(0.25 * (float)(image[iy + (ix+1) * szy] +
            image[iy + (ix-1) * szy] + image[(iy-1) + ix * szy] +
            image[(iy+1) + ix * szy]) + 0.5);
      }
    }
#pragma omp for
    for (size_t i = 0; i < (szx * szy); ++i){
      image[i] = temp[i];
    }
  }
}

  free(temp);
}
```

What did I do?

- I've split my OpenMP into an explicit parallel section in both C and Fortran
    - There's a cost associated with entering and leaving a parallel section so want as much as possible in one section
- I've then got two **omp for** sections in C - one for the blur and one for the copy back of the array
- In Fortran, I have one **OMP DO** for the blur and I introduce a new OpenMP section **WORKSHARE**
    - WORKSHARE is intended to parallelise Fortran whole array operations and is very useful to Fortran programmers
    - Just enclose Fortran array code (including many array intrinsic functions and elemental functions) in a WORKSHARE / END WORKSHARE block and they will be parallelised

Does it work?

- What's the speed up that we get this time?
  - Note that we won't see quite the same performance for the 1 processor result for a lot of (minor) reasons

Processors	Time
1	12.5
2	6.3
4	3.2
8	1.8
16	1.0
28	0.8

Does it work?

- How good is the scaling overall?
  - Pretty good really
  - Excellent to 4, good to 16 and acceptable to all 28

Processors	Time	%age of “perfect”
1	12.5	100
2	6.3	99.2
4	3.2	97.7
8	1.8	86.8
16	1.0	78.1
28	0.8	55.8

Does it work?

- Yes!
- The code now roughly doubles in speed up to 16 processors and still shows a speed up to 28
- Is this a decent performance?
  - Maybe
    - It’s certainly a useful speedup
  - Even if it isn’t, there isn’t much more that can be done in parallel left so this is probably the best that you can get
    - Check on a larger problem to see if it goes better

Bigger problem

- Increase NX and NY by a factor of 2 (each)
  - Might need to increase stack space using `**ulimit -s unlimited**`

Processors	Time	%age of “perfect”
1	51.8	100
2	24.9	104
4	12.7	102
8	7.0	93
16	3.7	88
28	2.7	69

Does it work?

- Get much better speedup up to 16 processors
- Still rolling off at 28 processors
  - This is quite common even in well written parallel code
    - Modern computers can’t keep the processors supplied with data at all times under all circumstances
    - Hard use of all of the processors really stresses the memory subsystem and this often shows as reducing scaling when moving over to using all the processors
      - This is especially true if you don’t do much work in each loop

Final Notes on Test Problem

- You might have noticed that I put the OpenMP loop directives on the outer loop of the two loops over the rank 2 (2D) array
- Was this random or did I have a reason?
  - There's a reason
  - If you do it the other way then it is **much** slower and doesn't scale at all (for the problem size here)
- You need to make sure that each thread has enough work to do - parallelising the inner loop means that you are doing far less work before the threads have to synchronise at the end of the loop
- You can parallelise over both loops in various ways (we show one in the example code that uses the OpenMP collapse directive to parallelise over both loops) but performance often doesn't improve

OpenMP Optional Programming

OpenMP optional

- In both C and Fortran OpenMP **directives** are ignored by non OpenMP compilers
  - In Fortran they are comments
  - In C compilers **might** warn about unknown pragmas for OpenMP but very few modern compilers do (even those that don't implement OpenMP themselves)
- The only bits of your code that are OpenMP specific are those to do with the OpenMP runtime library

OpenMP optional

- C and C++ has a standard way of including and removing code using **preprocessor directives**

- #ifdef _OPENMP
  PRINT *,"OpenMP Code"
  #else
  PRINT *,"Serial Code"
  #endif

- The OpenMP standard requires that _OPENMP be defined if the C/C++ compiler is compiling in OpenMP mode

OpenMP optional

- Fortran lacks a standard way of doing the same thing so OpenMP defines two ways of implementing the same thing

- The simplest one is just to put **!$** at the start of a line (followed by a space). Everything on the line after this is ignored unless the compiler is running in OpenMP mode

  - In non OpenMP compilers this is just another comment

  - It doesn't automatically respect **&** line continuation - you'll need another !$ on the continuation line

OpenMP optional

- Many modern Fortran compilers support C/C++ style preprocessors

  - Many of them use the file extension to enable the preprocessor ".f90" files aren't preprocessed ".F90" files are

- The OpenMP standard requires Fortran compilers that support preprocessors to define **_OPENMP** the same as the C/C++ compiler

- So you can write code using the same #ifdef approach as a C programmer

OpenMP optional

- You can get the OpenMP version from the _OPENMP variable

- The variable is defined to be yyyymm format date for the version of OpenMP that the compiler supports

- Because of the ordering of this format you can just check for it being >= the version that your code needs

- Nowadays if you're using OpenMP3 or older then there's no real point in doing this check but it is good form

Last Few Bits

OpenMP Sections

- Mostly OpenMP is used to do the same work on different data

- Sometimes you want to have each thread working on different tasks

- You can do this by just using
  if (omp_get_thread_num() == 0) {…} etc.

  - Inelegant and hard to make work on arbitrary number of threads

- Better way is OpenMP **sections**

OpenMP Sections

```
PROGRAM hello
  USE OMP_LIB
  IMPLICIT NONE
  PRINT *, "Hello in different sections"
!$OMP PARALLEL
!$OMP SECTIONS
    !$OMP SECTION
    PRINT *, "Hello ", omp_get_thread_num()
    !$OMP SECTION
    PRINT *, "Olá ", omp_get_thread_num()
    !$OMP SECTION
    PRINT *, "Hola", omp_get_thread_num()
    !$OMP SECTION
    PRINT *, "Heghlu'meH QaQ jajvam", &
        omp_get_thread_num()
!$OMP END SECTIONS
!$OMP END PARALLEL
END PROGRAM hello
```

OpenMP Sections

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

int main()
{
  printf("Hello in different sections\n");
#pragma omp parallel
  {
#pragma omp sections
    {
    #pragma omp section
    printf("Hello %i\n", omp_get_thread_num());
    #pragma omp section
    printf("Olá %i\n",omp_get_thread_num());
    #pragma omp section
    printf("Hola %i\n",omp_get_thread_num());
    #pragma omp section
    printf("Heghlu'meH QaQ jajvam %i\n",omp_get_thread_num());
    }
  }
  return 0;
}
```

OpenMP Sections

- You start an **OMP SECTIONS** section and within it you have multiple **OMP SECTION**

- All of the code within each section is run in parallel by a different single thread

- No more threads are used than there are sections

- If there are more sections than threads then as many as possible are run in parallel but all are run eventually

OpenMP Sections

- If you want to split up into *groups* of threads so that you can work on multiple tasks where each task uses multiple threads then you want OpenMP **teams**

- Teams are rather more complex so we're not covering them here

OpenMP Mutex

- OpenMP does have it's own Mutual Exclusion objects called **locks**
- You create a variable for the lock (**omp_lock_t** in C, **INTEGER(OMP_LOCK_KIND)** Fortran)
  - You have to initialise with **omp_lock_init(lock)**
  - Then you can use **omp_set_lock(lock)** to enter a critical section
  - **omp_unset_lock(lock)** to exit a critical section
  - **omp_test_lock(lock)** to see if you can enter a critical section
    - If **omp_test_lock** returns **true** then true then you have entered the critical section and must call **omp_unset_lock** to leave it

Set nthreads at runtime

- There are various ways of setting the number of threads while the program is running

- There's the function **omp_set_num_threads(nthreads)** that takes an integer number of threads. It can be invoked as many times as you want and will set the number of threads

- You can also specify **NUM_THREADS(n_threads)** on any PARALLEL directives (including PARALLEL DO/FOR). n_threads is a normal variable or literal constant

Turn off dynamic teams

- We mentioned earlier that technically specifying the number of threads only specifies the **maximum** threads that can be used

- If the OpenMP system thinks that it can improve performance by using fewer then it will

- You can turn this off by calling **omp_set_dynamic(on)**

  - **on** should be 1 to allow dynamic teams and 0 to prevent them in C

  - In Fortran, use .TRUE. and .FALSE.

OpenMP for non - parallel work

- While by **far** the most common use of OpenMP is to do parallel programming you can also use OpenMP for other jobs
  - Vectorisation directives
  - Device directives
    - GPUs
    - CPUs
    - FPGAs (research project level)

The End