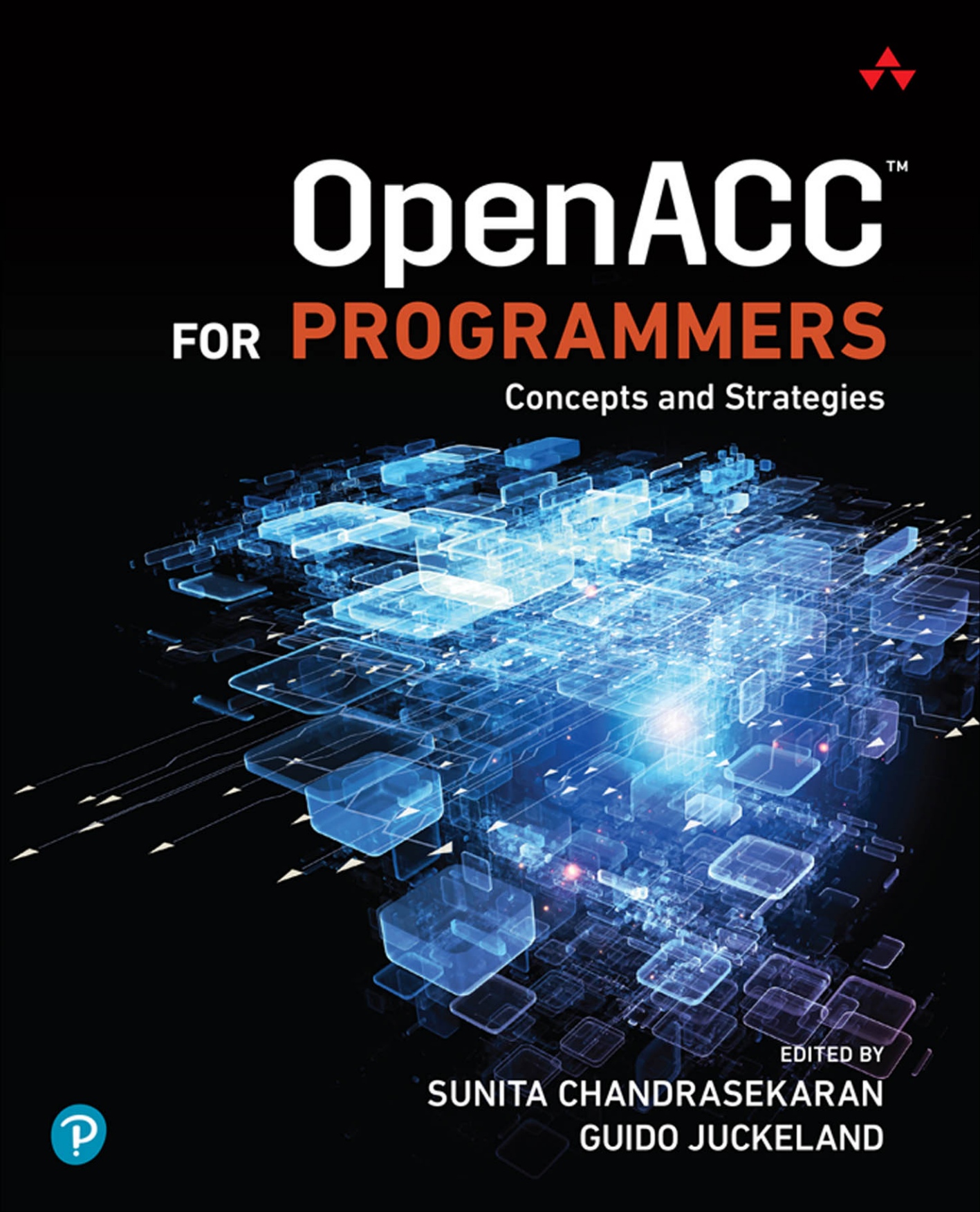
OpenACC™
FOR PROGRAMMERS
Concepts and Strategies
EDITED BY
SUNITA CHANDRASEKARAN
GUIDO JUCKELAND

OpenACC™ for Programmers

It might also be nice to track your progress as you go; it's much better than staring at a blank screen for the duration. So, every 100 iterations, let's call a modest output routine.

That is all there is to it for your serial Laplace Solver. Even with the initialization and output code, the full program clocks in at fewer than 100 lines. (See Listing 4.3 for the C code, and Listing 4.4 for Fortran.)

Listing 4.3 Serial Laplace Solver in C

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define WIDTH      1000
#define HEIGHT     1000
#define TEMP_TOLERANCE 0.01

double Temperature[HEIGHT+2][WIDTH+2];
double Temperature_previous[HEIGHT+2][WIDTH+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

  int i, j;
  int iteration=1;
  double worst_dt=100;
  struct timeval start_time, stop_time, elapsed_time;

  gettimeofday(&start_time,NULL);

  initialize();

  while ( worst_dt > TEMP_TOLERANCE ) {

    for(i = 1; i <= HEIGHT; i++) {
      for(j = 1; j <= WIDTH; j++) {
        Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                  + Temperature_previous[i-1][j]
                                  + Temperature_previous[i][j+1]
                                  + Temperature_previous[i][j-1]);
      }
    }

    worst_dt = 0.0;

    for(i = 1; i <= HEIGHT; i++){
      for(j = 1; j <= WIDTH; j++){
         worst_dt = fmax( fabs(Temperature[i][j]-
                               Temperature_previous[i][j]),
```

```
                          worst_dt);
          Temperature_previous[i][j] = Temperature[i][j];
        }
      }

      if((iteration % 100) == 0) {
        track_progress(iteration);
      }

      iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time);

    printf("\nMax error at iteration %d was %f\n",
           iteration-1, worst_dt);
    printf("Total time was %f seconds.\n",
           elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);
}

void initialize(){

    int i,j;

    for(i = 0; i <= HEIGHT+1; i++){
      for (j = 0; j <= WIDTH+1; j++){
        Temperature_previous[i][j] = 0.0;
      }
    }

    for(i = 0; i <= HEIGHT+1; i++) {
      Temperature_previous[i][0] = 0.0;
      Temperature_previous[i][WIDTH+1] = (100.0/HEIGHT)*i;
    }

    for(j = 0; j <= WIDTH+1; j++) {
      Temperature_previous[0][j] = 0.0;
      Temperature_previous[HEIGHT+1][j] = (100.0/WIDTH)*j;
    }
}

void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n",
           iteration);
    for(i = HEIGHT-5; i <= HEIGHT; i++) {
      printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

Listing 4.4 Fortran version of serial Laplace Solver

```
program serial
  implicit none

  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  double precision, parameter    :: temp_tolerance=0.01

  integer                        :: i, j, iteration=1
  double precision               :: worst_dt=100.0
  real                           :: start_time, stop_time

  double precision, dimension(0:height+1,0:width+1) :: &
                              temperature, temperature_previous

  call cpu_time(start_time)

  call initialize(temperature_previous)

  do while ( worst_dt > temp_tolerance )

    do j=1,width
      do i=1,height
        temperature(i,j) = 0.25* (temperature_previous(i+1,j)&
                            + temperature_previous(i-1,j)&
                            + temperature_previous(i,j+1)&
                            + temperature_previous(i,j-1))
      enddo
    enddo

    worst_dt=0.0

    do j=1,width
      do i=1,height
        worst_dt = max( abs(temperature(i,j) - &
                            temperature_previous(i,j)),&
                        worst_dt )
        temperature_previous(i,j) = temperature(i,j)
      enddo
    enddo

    if( mod(iteration,100).eq.0 ) then
      call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

  enddo

  call cpu_time(stop_time)

  print*, 'Max error at iteration ', iteration-1, ' was ', &
          worst_dt
  print*, 'Total time was ',stop_time-start_time, ' seconds.'
end program serial
```

```
subroutine initialize( temperature_previous )
  implicit none
  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  integer                        :: i,j
  double precision, dimension(0:height+1,0:width+1) :: &
                    temperature_previous

  temperature_previous = 0.0

  do i=0,height+1
    temperature_previous(i,0) = 0.0
    temperature_previous(i,width+1) = (100.0/height) * i
  enddo

  do j=0,width+1
    temperature_previous(0,j) = 0.0
    temperature_previous(height+1,j) = ((100.0)/width) * j
  enddo
end subroutine initialize

subroutine track_progress(temperature, iteration)
  implicit none
  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  integer                        :: i,iteration

  double precision, dimension(0:height+1,0:width+1) :: &
        temperature

  print *, '------- Iteration number: ', iteration, ' ------'
  do i=5,0,-1
    write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no') &
              height-i,width-i,temperature(height-i,width-i)
  enddo
  print *
end subroutine track_progress
```

4.1.2 COMPILING THE CODE

Take a few minutes to make sure you understand the code fully. In addition to the main loop, you have a small bit of initialization, a timer to aid in optimizing, and a basic output routine. This code compiles as simply as

```
pgcc laplace.c
```

Here it is for the PGI compiler:

```
pgcc laplace.f90
```

We use PGI for performance consistency in this chapter. Any other standard compiler would work the same. If you run the resulting executable, you will see something like this:

```
. . .
. . .
---------- Iteration number: 3200 ------------
. . . [998,998]: 99.18  [999,999]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
. . . [998,998]: 99.19  [999,999]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 21.344162 seconds.
```

The output shows that the simulation looped 3,372 times before all the elements stabilized (to within our 0.01 degree tolerance). If you examine the full output, you can see the elements converge from their zero-degree starting point.

The times for both the C and the Fortran version will be very close here and as you progress throughout optimization. Of course, the time will vary depending on the CPU you are using. In this case, we are using an Intel Broadwell running at 3.0 GHz. At the time of this writing, it is a very good processor, so our eventual speedups won't be compared against a poor serial baseline.

This is the last time you will look at any code outside the main loop. You will henceforth exploit the wonderful ability of OpenACC to allow you to focus on a small portion of your code—be it a single routine, or even a single loop—and ignore the rest. You will return to this point when you are finished.

4.2 Creating a Naive Parallel Version

In many other types of parallel programming, you would be wise to stare at your code and plot various approaches and alternative algorithms before you even consider starting to type. With OpenACC, the low effort and quick feedback allow you to dive right in and try some things without much risk of wasted effort.

4.2.1 FIND THE HOT SPOT

Almost always the first thing to do is find the **hot spot:** the point of highest numerical intensity in your code. A profiler like those you've read about will quickly locate and

rank these spots. Often, as is the case here, it is obvious where to start. A large loop is a big flag, and you have two of them within the main loop. This is where we focus.

4.2.2 IS IT SAFE TO USE KERNELS?

The biggest hammer in your toolbox is the `kernels` directive. Refer to Chapter 1 for full details on `kernels`. Don't resist the urge to put it in front of some large, nested loop. One nice feature about this directive is that it is safe out of the box; until you start to override its default behavior with additional directives, the compiler will be able to see whether there are any code-breaking dependencies, and it will make sure that the device has access to all the required data.

4.2.3 OPENACC IMPLEMENTATIONS

Let's charge ahead and put `kernels` directives in front of the two big loops. The C and Fortran codes become the code shown in Listings 4.5 and 4.6.

Listing 4.5 C Laplace code main loop with kernels directives

```
while ( worst_dt > TEMP_TOLERANCE ) {

   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++) {
      for(j = 1; j <= WIDTH; j++) {
         Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                   + Temperature_previous[i-1][j]
                                   + Temperature_previous[i][j+1]
                                   + Temperature_previous[i][j-1]);
      }
   }

   worst_dt = 0.0;

   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++){
      for(j = 1; j <= WIDTH; j++){
         worst_dt = fmax( fabs(Temperature[i][j]-
                               Temperature_previous[i][j]),
                          worst_dt);
        Temperature_previous[i][j] = Temperature[i][j];
      }
   }

   if((iteration % 100) == 0) {
      track_progress(iteration);
   }

   iteration++;
}
```