41391 High performance computing: Miscellaneous parallel programmes in Fortran

Nilas Mandrup Hansen, Ask Hjorth Larsen

January 19, 2010

1 Introduction

This document concerns the implementation of a Fortran programme capable of calculating digits of pi, one which can do matrix-vector multiplication, and one which calculates the Mandelbrot set. The programmes are parallelized using the automatic parallelization (which the `f90` compiler supports through the `-xautopar` option) and/or explicitly with OpenMP.

2 Calculating digits of pi

It is known that

$$\pi = \int_0^1 \frac{4}{1+x^2}dx \approx \frac{4}{N}\sum_{i=1}^{\infty}\left[1+\left(\frac{i-\frac{1}{2}}{N}\right)^2\right]^{-1}. \tag{1}$$

The terms of this sum are independent of each other, and the calculation therefore embarrasingly parallel.

With OpenMP we implement this calculation using the following loop:

```
!$omp parallel do shared(n) private(i) reduction(+: pi)
    do i=1, n
       pi = pi + 1.0 / (1.0 + ((i - 0.5) / n)**2)
    end do
!$omp end parallel do
```

The exact same code is used with automatic parallelization, with the exception of the parameters `-xautopar -xreduction` instead of `-xopenmp` in the makefile.

Figure 1 shows the speedup as a function of processor count for the π calculation code using OpenMP as well as automatic parallelization, which seem to differ little in terms of performance. With a small number of iterations (2^{20}) the parallelization is, unsurprisingly, less efficient than with a larger number (2^{24})

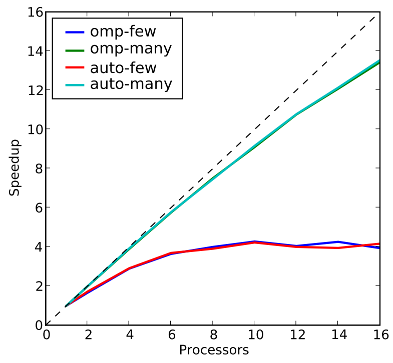


Figure 1: Speedup for parallel calculation of π with automatic parallelization and OpenMP when calculating with few or many iterations. The total time on the order of milliseconds when doing only few iterations, which means process startup overhead prevents perfect scaling of this otherwise embarrasingly parallel operation.

(both iteration counts are rather large given that the calculated value is only a `double precision` number, but the high iteration counts make timings more reliable).

The source files can be found in the appendix.

3 Matrix times vector multiplication

The product y of a matrix A and a vector x has the elements

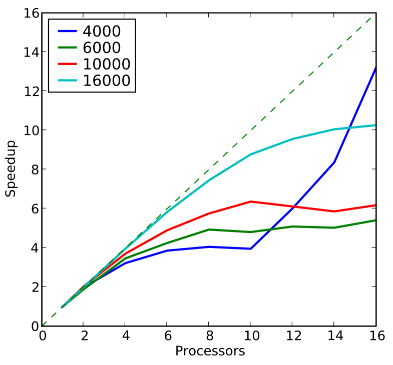
$$y_i = \sum_j A_{ij} x_j. \tag{2}$$

This we implement in terms of an outer loop over i and an inner loop over j, where we parallelize the outer loop.

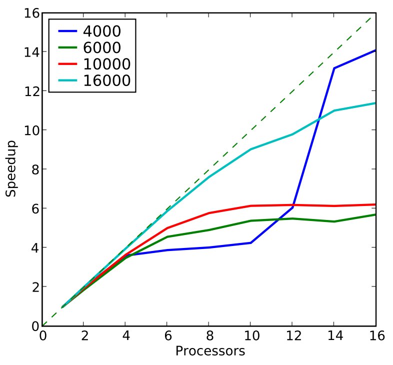
Since Fortran uses column-major ordering, and since we don't want to worry too much about transposes and definitions, we use the indexing scheme `A(i*N + j)` rather than actual two-dimensional arrays.

With OpenMP and automatic parallelization we use the same loop:

```
!$omp parallel do shared(A, x, y, M, N) private(tmp, i, j)
    do i = 1, M
       tmp = 0.0
       do j = 1, N
          tmp = tmp + A(j + N * i) * x(j)
       end do
       y(i) = tmp
```



(a) Automatically parallelized



(b) OpenMP

Figure 2: Speedup of matrix times vector function using `-xautopar` (a) and explicit OpenMP (b).

```
    end do
!$omp end parallel do
```

Each processor writes to different areas of the destination matrix, so no synchronization is required.

Figure 2 shows the speedup of the matrix times vector function with `-xautopar` and OpenMP for different matrix sizes and processor counts. With a 4000×4000 matrix, most or all of the matrix fits into the L2 cache when using more than 10 CPUs (7.6 MB for 16 CPUs with a cache size of 8 MB), which results in a dramatic speedup due to fewer cache misses.

The larger matrices will not fit into the cache (the 6000×6000 uses 17 MB on 16 CPUs), and thus behave in a more usual manner: Larger matrices parallelize better as the CPUs have more work compared to the (constant) amount of synchronization. We think the imperfect scaling observed for very large matrices is caused by having too many processes read data from the main memory.

We note that smaller matrices (of size around 500-2000) tend to scale superlinearly as they start fitting better into L1 and L2 caches (not shown in the figure).

Source files can be found in the appendix.

4 Mandelbrot

For this exercise, the Mandelbrot set is computed and visualized by using the provided mandelbrot.zip file, which contains the following FORTRAN files;

- `main.f90`: The main program for Mandelbrot.

- `mandel.f90`: Contains the Mandelbrot calculations.
- `timestamp.f90`: Simply prints the current YMDHMS date as a time stamp.

Also provided is a `Makefile` as well as several other files which are not important to mention.

For the first part of the exercise, the Mandelbrot program is simply to be compiled by the `gmake` command in order to produce a serial version of Mandelbrot. The output of the executable is shown in fig. 3

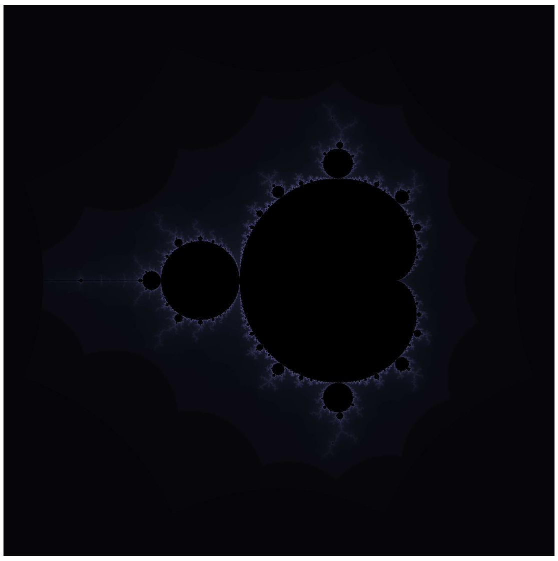


Figure 3: The Mandelbrot set

4.1 Parallelizing the Code

For the second part of the exercise, the `Mandelbrot` executable should be generated as a parallel version by using the OpenMP worksharing constructs. Most of the resources are used in the triple do-loop calculating the Mandelbrot set. In order to parallelize the code, a team of threads is created by using the command `!$parallel do`. Also the loop counters are set to private, while all other variables are set to shared (see source code in app. C).

When compiling the code with the mentioned modifications, one obtains a parallel version of the executable `mandelbrot`. The runtimes for a different number of threads is shown in Fig. 4.
It can be seen from the figure that the code does not scale well.

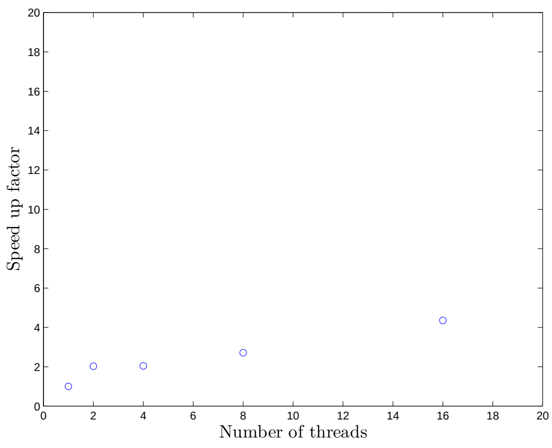


Figure 4: Speed up vs. number of threads. Using static workload distribution.

4.2 Modifications the to the Parallel version of the Mandelbrot

From the preceding section it was seen that the code did not scale well. In order to optimize the parallel version, we profiled the program by utilizing the `collect` command. The result was visualized in the Analyzer program. The result showed that approximately 95% of the workload was distributed to only two threads even though four threads were available.

In order to overcome the problem a dynamic distribution of the workload was specified in the code by specifying the *clause* `schedule(dynamic,5)` in the parallel `do` environment (see app. D). The result is presented in Fig. 5
From the figure it can be seen that the speed-up factor has improved considerably when using dynamic workload distribution.

4.3 Using Orphaning

As a last part of this exercise, the code should utilize orphaning. This is a useful feature of the OpenMP, since it allows the user to declare workshare or synchronization directives which are not located within a parallel region. This means that the programmer has the possibility to run the same subroutine with or without parallelization.

An example is shown in app. E. Here a flag has been inserted in the main document which allows the user to run the `subroutine mandel` in either serial or parallel computation.

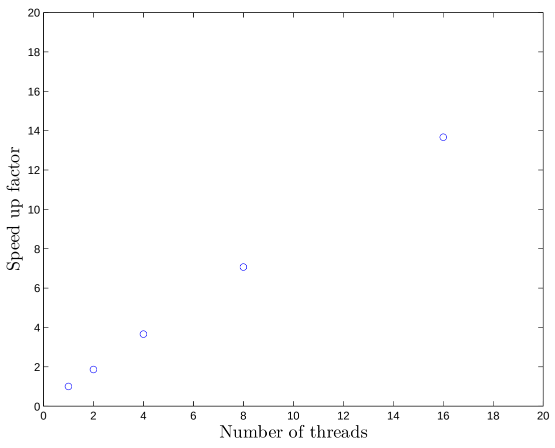


Figure 5: Speed up vs. number of threads. Using dynamic workload distribution.

A Pi calculation source files

A.1 `pi/omp/pi.f90`

```
module picalc

contains

  subroutine findpi(n, pi)
    implicit none

    integer, intent(in) :: n
    double precision, intent(out) :: pi
    call findpi_ompreduce(n, pi)
    !call findpi_ourreduce(n, pi)
    pi = pi * 4.0 / n
  end subroutine findpi

  subroutine findpi_ompreduce(n, pi)
    implicit none

    integer, intent(in) :: n
    double precision, intent(out) :: pi

    integer :: i

    pi = 0.0

!$omp parallel do shared(n) private(i) reduction(+: pi)
    do i=1, n
       pi = pi + 1.0 / (1.0 + ((i - 0.5) / n)**2)
    end do
!$omp end parallel do
  end subroutine findpi_ompreduce

  subroutine findpi_ourreduce(n, pi)
    implicit none

    integer, intent(in) :: n
    double precision, intent(out) :: pi

    integer :: i
    double precision :: local_pi

    pi = 0.0
!$omp parallel shared(n, pi) private(local_pi)
    local_pi = 0.0
!$omp do private(i)
    do i=1, n
       local_pi = local_pi + 1.0 / (1.0 + ((i - 0.5) / n)**2)
    end do
!$omp end do

!$omp critical (sum)
    pi = pi + local_pi
!$omp end critical (sum)
!$omp end parallel
  end subroutine findpi_ourreduce

  subroutine iterate(i, n, tmp)
    implicit none

    integer, intent(in) :: i, n
    double precision, intent(out) :: tmp

    tmp = 1.0 / (1.0 + ((i - 0.5) / n)**2)
  end subroutine iterate

end module picalc
```

A.2 pi/omp/main.f90

```
program calculate_pi
use picalc, only: findpi
use omp_lib

integer :: n! = 2**27
double precision :: pi, t1, t2

read*, n

t1 = omp_get_wtime()
call findpi(n, pi)
t2 = omp_get_wtime()

print*, omp_get_max_threads(), t2 - t1

end program calculate_pi
```

A.3 pi/omp/makefile

```
ARGS=-O3 -openmp
AUTOPAR_ARGS = -xautopar -xloopinfo -xreduction

findpi: pi.o main.f90
 f90 $(ARGS) pi.o main.f90 -o findpi

pi.o:
 f90 -c $(ARGS) pi.f90

clean:
 rm -f pi.o picalc.mod
```

B Matrix times vector source files

B.1 mxv/omp/main.f90

```
program matrix
  use m_mxv
  use omp_lib

  integer :: M, N

  double precision, dimension(:), allocatable :: A
  double precision, dimension(:), allocatable :: y
  double precision, dimension(:), allocatable :: x
  double precision :: tmp
  integer :: i, j
  double precision :: time1, time2

  integer :: ops_per_mxv, approx_opcount, number_of_runs, opcount, iter

  read*, M

  N = M

  ops_per_mxv = M * N
  approx_opcount = 8000 * 8000 * 4 ! probably reasonable in terms of real-
      time
  number_of_runs = (approx_opcount / ops_per_mxv) * omp_get_max_threads()
  opcount = number_of_runs * ops_per_mxv

  allocate(A(N * M), y(M), x(N))

  write(unit=0, *) 'size:', M, '; runs:', number_of_runs, '; ops:', opcount,
      &
       '; cpus:', omp_get_max_threads()

!$omp parallel
!$omp do
  do i=1, M
     do j=1, N
        A(j + M * i) = 1.0
     enddo
  enddo
!$omp end do
!$omp do
  do i=1, N
     x(i) = 1.0
  enddo
!$omp end do
!$omp end parallel
  time1 = omp_get_wtime()
  do iter=1, number_of_runs
     call mxv(A, M, N, x, y)
  enddo
  time2 = omp_get_wtime()
  print*, omp_get_max_threads(), ((time2 - time1) / real(number_of_runs))
end program matrix
```

B.2 mxv/omp/mxv.f90

```
module m_mxv

contains

  subroutine mxv(A, M, N, x, y)
    integer, intent(in) :: M, N
    double precision, dimension(N * M), intent(in) :: A
    double precision, dimension(N), intent(in) :: x
    double precision, dimension(M), intent(out) :: y

    integer :: i, j

    double precision :: tmp
!$omp parallel do shared(A, x, y, M, N) private(tmp, i, j)
    do i = 1, M
       tmp = 0.0
       do j = 1, N
          tmp = tmp + A(j + N * i) * x(j)
       end do
       y(i) = tmp
    end do
!$omp end parallel do

  end subroutine mxv

end module m_mxv
```

B.3 mxv/omp/makefile

```
ARGS = -g -fast -xopenmp -xloopinfo

run: mxv.o main.f90
 f90 $(ARGS) main.f90 mxv.o -o run

mxv.o:
 f90 -c $(ARGS) mxv.f90

clean:
 rm -f run mxv.o m_mxv.mod
```

C Source code - mandel.f90

```
      subroutine mandel(n, image, max_iter)

      integer   ( kind = 4 ) :: n
      integer   ( kind = 4 ) :: image(n,n)
      integer   ( kind = 4 ) :: max_iter
      integer   ( kind = 4 ) :: i, j, k
      real      ( kind = 8 ) :: x, x1, x2
      real      ( kind = 8 ) :: y, y1, y2
      real      ( kind = 8 ) :: x_max =   1.25D+00
      real      ( kind = 8 ) :: x_min = - 2.25D+00
      real      ( kind = 8 ) :: y_max =   1.75D+00
      real      ( kind = 8 ) :: y_min = - 1.75D+00

      image = 0.0

!$omp parallel default(none) shared(n,image,max_iter,x,y,x_min,x_max,y_min,
    y_max,x1,y1,x2,y2) private(i,j,k)
!$omp do
     do i = 1, n
        do j = 1, n

          x = (  real (     j - 1, kind = 8 ) * x_max   &
               + real ( n - j,     kind = 8 ) * x_min ) &
               / real ( n     - 1, kind = 8 )

          y = (  real (     i - 1, kind = 8 ) * y_max   &
               + real ( n - i,     kind = 8 ) * y_min ) &
               / real ( n     - 1, kind = 8 )

```

```fortran
      !    image(i,j) = 0

           x1 = x
           y1 = y

           do k = 1, max_iter

              x2 = x1 * x1 - y1 * y1 + x
              y2 =  2 * x1 * y1 + y

              if ( x2 < -2.0D+00 .or. &
                    2.0D+00 < x2 .or. &
                   y2 < -2.0D+00 .or. &
                     2.0D+00 < y2 ) then

                     image(i,j) = k
                     exit

              end if

              x1 = x2
              y1 = y2

           end do

         end do
      end do
!$omp end do
!$omp end parallel
      end subroutine
```

D Source code - `mandel.f90`

```fortran
      subroutine mandel(n, image, max_iter)

      integer   ( kind = 4 ) :: n
      integer   ( kind = 4 ) :: image(n,n)
      integer   ( kind = 4 ) :: max_iter
      integer   ( kind = 4 ) :: i, j, k
      real      ( kind = 8 ) :: x, x1, x2
      real      ( kind = 8 ) :: y, y1, y2
      real      ( kind = 8 ) :: x_max =   1.25D+00
      real      ( kind = 8 ) :: x_min = - 2.25D+00
      real      ( kind = 8 ) :: y_max =   1.75D+00
      real      ( kind = 8 ) :: y_min = - 1.75D+00

      image = 0.0

!$omp parallel default(none) shared(n,image,max_iter,x,y,x_min,x_max,y_min,
     y_max,x1,y1,x2,y2) private(i,j,k)
!$omp do schedule(dynamic,5)
     do i = 1, n
        do j = 1, n

           x = (  real (     j - 1, kind = 8 ) * x_max   &
                + real ( n - j,     kind = 8 ) * x_min ) &
                / real ( n     - 1, kind = 8 )

           y = (  real (     i - 1, kind = 8 ) * y_max   &
                + real ( n - i,     kind = 8 ) * y_min ) &
                / real ( n     - 1, kind = 8 )

      !    image(i,j) = 0

           x1 = x
           y1 = y
```

```

          do k = 1, max_iter

             x2 = x1 * x1 - y1 * y1 + x
             y2 =  2 * x1 * y1 + y

             if ( x2 < -2.0D+00 .or. &
                   2.0D+00 < x2 .or. &
                  y2 < -2.0D+00 .or. &
                    2.0D+00 < y2 ) then

                    image(i,j) = k
                    exit

             end if

             x1 = x2
             y1 = y2

          end do

        end do
      end do
!$omp end do
!$omp end parallel
      end subroutine
```

E Source code - `main.f90` and `mandel.f90`

```
program main

!*****************************************************************************80
!
!  MAIN is the main program for MANDELBROT.
!
!  Discussion:
!
!    MANDELBROT computes an image of the Mandelbrot set.
!
!  Licensing:
!
!    This code is distributed under the GNU LGPL license.
!
!  Modified:
!
!    08 August 2009
!
!  Author:
!
!    John Burkardt
!
!  Modified by:
!    Bernd Dammann
!    Boyan Lazarov
!
!  Local Parameters:
!
!    Local, integer COUNT_MAX, the maximum number of iterations taken
!    for a particular pixel.
!
  implicit none

  integer   ( kind = 4 ) :: n = 2501
  integer   ( kind = 4 ) :: count_max = 800

  integer   ( kind = 4 ) :: c
  real      ( kind = 8 ) :: c_max, c_max_inv
  integer   ( kind = 4 ), dimension(:,:), allocatable :: image
  character ( len = 255 ) :: filename

  real      ( kind = 8 ) :: x_max =   1.25D+00
  real      ( kind = 8 ) :: x_min = - 2.25D+00
  real      ( kind = 8 ) :: y_max =   1.75D+00
  real      ( kind = 8 ) :: y_min = - 1.75D+00
  real:: flag
  flag=10.1

  allocate(image(n,n))

  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) 'MANDELBROT'
  write ( *, '(a)' ) '  FORTRAN90 version'
  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) '  Create an PNG image of the Mandelbrot set.'
  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) '  For each point C = X + i*Y'
  write ( *, '(a,g14.6,a,g14.6,a)' ) '  with X range [', x_min, ',', x_max,
      ']'
  write ( *, '(a,g14.6,a,g14.6,a)' ) '  and  Y range [', y_min, ',', y_max,
      ']'
  write ( *, '(a,i8,a)' ) '  carry out ', count_max, ' iterations of the map'
  write ( *, '(a)' ) '  Z(n+1) = Z(n)^2 + C.'
  write ( *, '(a)' ) '  If the iterates stay bounded (norm less than 2)'
  write ( *, '(a)' ) '  then C is taken to be a member of the set.'
  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) '  A PNG image of the set is created using'
  write ( *, '(a,i8,a)' ) '    N = ', n, ' pixels in the X direction and'
  write ( *, '(a,i8,a)' ) '    N = ', n, ' pixels in the Y direction.'
  write ( *, '(a)' ) ' '
!
!  Carry out the iteration for each pixel, determining COUNT.

 call timestamp ( )

!! private(i,j,k)
if(flag.GT.1) then
!$omp parallel default(none) shared(n,image,count_max)
 call mandel(n,image,count_max)
!$omp end parallel
else
call mandel(n,image,count_max)
endif

 write ( *, '(a)' ) ' '
 write ( *, '(a)' ) ' Calculation of the image finished. '
 call timestamp ( )

! uncomment the following line, if you don't need the PNG output
! stop
!
! call writepng to save a PNG image in filename

  filename = "mandelbrot.png"//CHAR(0)
  call writepng(filename, image, n, n)
  deallocate(image)

  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) &
    '  PNG image data stored in "' // trim ( filename ) // '".'
  write ( *, '(a)' ) ' '
  write ( *, '(a)' ) 'MANDELBROT'
  write ( *, '(a)' ) '  Normal end of execution.'
  write ( *, '(a)' ) ' '
  call timestamp ( )

  stop
end
```

```
      subroutine mandel(n, image, max_iter)

      integer   ( kind = 4 ) :: n
      integer   ( kind = 4 ) :: image(n,n)
      integer   ( kind = 4 ) :: max_iter
      integer   ( kind = 4 ) :: i, j, k
      real      ( kind = 8 ) :: x, x1, x2
      real      ( kind = 8 ) :: y, y1, y2
      real      ( kind = 8 ) :: x_max =   1.25D+00
      real      ( kind = 8 ) :: x_min = - 2.25D+00
      real      ( kind = 8 ) :: y_max =   1.75D+00
      real      ( kind = 8 ) :: y_min = - 1.75D+00

      image = 0.0


!$omp do schedule(dynamic,5)
     do i = 1, n
        do j = 1, n

          x = (  real (     j - 1, kind = 8 ) * x_max   &
               + real ( n - j,     kind = 8 ) * x_min ) &
               / real ( n     - 1, kind = 8 )

          y = (  real (     i - 1, kind = 8 ) * y_max   &
               + real ( n - i,     kind = 8 ) * y_min ) &
               / real ( n     - 1, kind = 8 )

      !    image(i,j) = 0

          x1 = x
          y1 = y

          do k = 1, max_iter

             x2 = x1 * x1 - y1 * y1 + x
             y2 =  2 * x1 * y1 + y

             if ( x2 < -2.0D+00 .or. &
                   2.0D+00 < x2 .or. &
                  y2 < -2.0D+00 .or. &
                    2.0D+00 < y2 ) then

                    image(i,j) = k
                    exit

             end if

             x1 = x2
             y1 = y2

          end do

        end do
      end do
!$omp end do

      end subroutine
```