Hey Derp, can you help out on this project? We have to get some results by monday
Me, programming happily at work
Boss
Sure! I'm happy to help!
Oh...and one more thing...
The code we're working on was written in FORTRAN
FFFFFFFFFF FFFFFFF FFOOOO RRRRR TTTT RRRRA AAAN NNNNN

Performance Programming with Fortran

Jeremy Roberts

01/31/2012

Outline

References, etc.

1. C. Evangelinos, 12.950 course slides
2. Chapman, B., *et al.*, **Using OpenMP**, The MIT Press (2008)
3. `www.openmp.org/mp-documents/OpenMP3.0-FortranCard.pdf`
4. `https://computing.llnl.gov/tutorials/openMP/`

There is a ton of information out there for all the stuff in these slides!

Go get the Intel compilers (for noncommercial use) at:
`http://software.intel.com/en-us/articles/non-commercial-software-download/`

Bare code in `/home/fortran12/users/robertsj`

What defines a program's performance?

Possible metrics:

1. wall time
2. CPU time
3. FLOPS (**f**loating-point **o**perations **p**er **s**econd)
4. memory efficiency (*i.e.* minimizing cache hits)
5. parallel scaling
6. (maintainability)
7. (correctness)

Others?

An Example: Matrix-Vector Multiplication

We want to compute

$$\vec{y} \leftarrow \mathbf{A}\vec{x}\,,$$

where $\mathbf{A}$ is an $n \times n$ matrix, and x and y are n-vectors. *Conceptually*, this is a bunch of dot products

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \vec{a}_{1:} \cdot \vec{x} \\ \vec{a}_{2:} \cdot \vec{x} \\ \vec{a}_{3:} \cdot \vec{x} \end{bmatrix}$$

or (better) a linear combination of the columns of $\mathbf{A}$

$$\mathbf{A}\vec{x} = x_1\vec{a}_{:1} + x_2\vec{a}_{:2} + x_3\vec{a}_{:3}\,.$$

Numerically, we compute

$$y_i = \sum_{j=1}^{n} a_{ij}x_j\,,\ \ i = 1, \ldots, n\,.$$

A First Implementation

Listing 1: matvec-driver-bare.f90

```fortran
! Driver program to test matrix-vector multiplication
program matvec_driver
  implicit none
  double precision, allocatable :: A(:, :)
  double precision, allocatable :: x(:), y(:)
  double precision :: alpha = 1.0, beta = 0.0
  integer :: n, m, lda, incx = 1, incy = 1, i, num_loops
  character*1 :: trans = 'n'
  m   = 2000
  n   = m
  lda = m
  ! Create the matrix and vector

  ! Initialize A and x

  ! Loop over and apply A several times for consistent timing
  num_loops = 100
  do i = 1, num_loops
    ! Reset
    y = 0.0
    call dgemv('n', m, n, alpha, A, lda, x, incx, beta, y, incy)
  end do

end program matvec_driver
```

Fill it out and name it matvec-driver.f90.

A First Implementation

Listing 2: matvec-bare.f90

```fortran
subroutine dgemv(trans, m, n, alpha, A, lda, x, incx, beta, y, incy)
! Simplified DGEMV that does
!     y := A*x
! We follow the BLAS signature for compatibility.  For more,
! see http://netlib.org/blas/.
  implicit none
  ! input/output
  character*1, intent(in)          :: trans
  integer, intent(in)              :: m, n, lda, incx, incy
  double precision, intent(in)     :: alpha, beta
  double precision, intent(in)     :: A(m, n)
  double precision, intent(in)     :: x(n)
  double precision, intent(inout)  :: y(m)








end subroutine dgemv
```

Implement $y_i = \sum_{j=1}^{n} a_{ij} x_j \,, \; i = 1, \ldots, n$, and name it matvec-first.f90.

The Simplest Timer: `time`

First, compile

```
gfortran -o matvec-first \
  matvec-first.f90 matvec-driver.f90
```

Then on a Linux machine (maybe Macs, too), run

```
time matvec-first
```

and get

```
real    0m4.511s
user    0m4.470s
sys     0m0.020s
```

The `user` + `sys` time is **CPU time**. The `real` time is **wall time**.
Question: $t_{CPU} \geq t_{wall}$ or $t_{CPU} \leq t_{wall}$?

Compiler-based Timer

Listing 3: wtime.f90

```
double precision function wtime()
! WTIME returns a reading of the wall clock time.  Use as follows
!
!   double precision :: t
!   t = wtime()
!   ! do some work
!   print *, "elapsed time = ", wtime() - t, " seconds."
!
! This is a slight modification of John Burkardt's function of the same
! name.  See http://people.sc.fsu.edu/~jburkardt for this and other useful
! source.
  implicit none
  integer count       ! processor-dependent value based on current value
                      ! of processor clock
  integer count_rate  ! clock counts per second
  integer count_max   ! maximum value that count can take
  call system_clock(count, count_rate, count_max)
  wtime = dble(count) / dble (count_rate)
end function wtime
```

Add this timer to the matvec-driver.f90.

Compiler-based Timer

Compile

```
gfortran -o matvec-first wtime.f90 \
  matvec-first.f90 matvec-driver.f90
```

and then run

```
matvec-first
```

and get

```
elapsed time =    4.4969999999739230       seconds.
```

Which time is this? Also, note the poor precision. Apparently gfortran's clock isn't too precise.

Other Timers

Other timing utilities are provided by

1. OpenMP (`omp_get_wtime()`; see later slides)
2. MPI (`MPI_Wtime()`)
3. Hardware vendors
4. Software vendors

For most scientific applications, the OpenMP and MPI functions should suffice; YMMV.

FLOPS - Counting

Massachusetts Institute of Technology

How many floating point operations in $\mathbf{A}x$? (Assume square.)

$$y_i = \sum_{j=1}^{n} a_{ij} x_j\,, \; i = 1, \ldots, n\,.$$

```
do i = 1, n
  do j = 1, n
    y(i) = y(i) + A(i, j) * x(j)
  end do
end do
```

For $m = n = 2000$ and 100 runs, $t_{CPU} \approx 4.5$ seconds. How many FLOPS?

Add a line to compute FLOPS in matvec-driver.f90.

FLOPS - Bad news

Linux users: learn about your machine. Execute `sudo lshw > out.txt`. Digging through, I find

```
*-cpu
     description: CPU
     product: Intel(R) Core(TM) i7 CPU         970  @ 3.20GHz
...
        description: L1 cache
...
        slot: L1-Cache
        size: 192KiB
...
        description: L2 cache
...
        slot: L2-Cache
        size: 1536KiB
...
        description: L3 cache
...
        slot: L3-Cache
        size: 12MiB
```

So I crunch $3.2 \cdot 10^9$ times per second. That means I should get 3.2 GFLOPS, right?

FLOPS - Keep trying

Optimize:

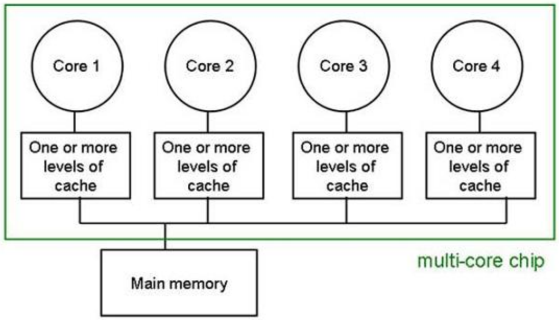
```
gfortran -O3 -o matvec-first matvec-first.f90 matvec-driver.f90
matvec-first
elapsed time =    3.5709999999962747        seconds.
```

or about 225 MFLOPS. Still over an order of magnitude off. Another compiler?

```
ifort -O3 -o matvec-first matvec-first.f90 matvec-driver.f90
matvec-first
elapsed time =    0.367999999987660        seconds.
```

or about 2.2 GFLOPS. Wow! But not yet optimal.

Interlude - Memory


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
multi-core chip

Now remember L1, L2, and L3 cache sizes (192 KiB, 1536 KiB, and 12 MB). A double precision number is 8 bytes. For $n = 2000$, one vector is only 16 KiB but the matrix is 32 MB! What's our issue?

FLOPS - Keep trying harder

Fortran uses *column-major* storage. Thus, the elements of matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

are stored as $a_{11}, a_{21}, a_{31}, a_{12}$, and so on. But look at

```
do i = 1, n
  do j = 1, n
    y(i) = y(i) + A(i, j) * x(j)
  end do
end do
```

Implement our fix, naming it matvec-second.f90.

FLOPS - A Fix?

Switch order:

```
do j = 1, n     ! j and i switched
  do i = 1, n
    y(i) = y(i) + A(i, j) * x(j)
```

Compile and run

```
gfortran -O3 -o matvec-second matvec-second.f90 matvec-driver.f90
you@pc:~/fpp_code/$ matvec-second < input.txt
elapsed time =   0.38800000003539026       seconds.
```

which is much better for gfortran, but

```
ifort -O3 -o matvec-first matvec-second.f90 matvec-driver.f90
you@pc:~/fpp_code/$ matvec-second < input.txt
elapsed time =   0.374899999995250       seconds.
```

is the same as before! Why? Are we optimal yet?

FLOPS - OMILAT

Setting 1000 loops and using AMD's Math Core Library:

```
gfortran -O3 -o matvex-amcl wtime.f90 matvec-driver.f90
  -L /home/robertsj/opt/acml/acml4.4.0/gfortran64/lib/
  -lacml -lgfortran
matvec-amcl < input.txt
elapsed time =    3.4669999999459833       seconds.
```

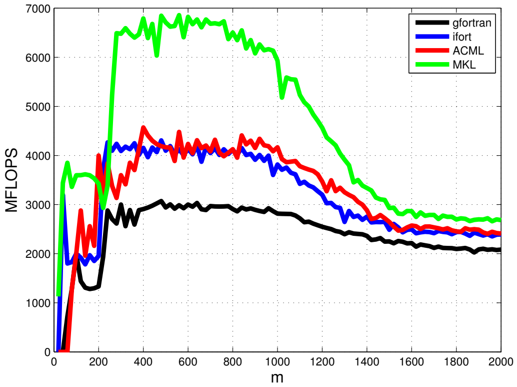
again, about 2.25 GFlops. Using Intel's Math Kernel Library:

```
ifort -O3 -o matvex-mkl wtime.f90  matvec-driver.f90 -i8
 -I$MKLROOT/include -Wl,--start-group
 $MKLROOT/lib/intel64/libmkl_intel_ilp64.a
 $MKLROOT/lib/intel64/libmkl_sequential.a
 $MKLROOT/lib/intel64/libmkl_core.a -Wl,--end-group -lpthread -lm
matvec-mkl < input.txt
elapsed time =    3.0593469142913 8       seconds.
```

or about 2.61 GFlops.

Only Masochists Implement Linear Algebra Themselves! Leave it to the pros!

FLOPS - Final Thoughts



Problem and cache size are important: *the CPU can only do its fastest work if it gets the data just as fast.*

Suggested Homework: Adapt matvec-driver.f99 to produce a graph like the one shown.

Libraries

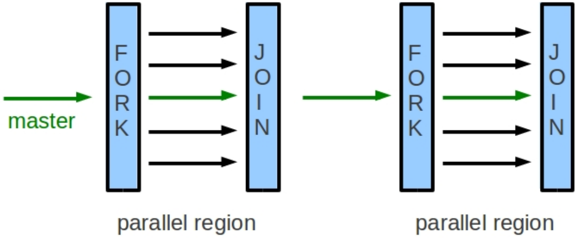
If you do numerical anything, you might run into these:

- BLAS, LAPACK, *etc.*: **standard** linear algebra libraries with many high performance vendor implementations (AMD, Intel, ATLAS)
- PETSc: parallel linear and nonlinear solvers
- SLEPSc: parallel eigensolvers
- Trilinos: parallel solvers
- MOOSE: framework for multiphysics PDE's
- HDF5: library for scientific data

... plus many, many more. Don't reinvent the wheel!

The OpenMP Programming Model

- OpenMP is a standard defining an API for *shared memory* applications
- Employs the "fork-join" model
- Encourages incremental parallelization
- Fortran (original), C/C++



OpenMP: Conceptual Overview

When performing work in parallel

- threads read and write *shared* data
- threads can read and write private copies
- synchronization is required to avoid “race conditions”
- loop-carried dependencies must be avoided

Note, memory is multi-level: cache is *not* shared. One must be careful to make sure all threads see what they should see.

Example: Computing π in Parallel

Listing 4: pi-bare.f90

```fortran
! Computing pi in parallel by numerical integration
program pi
  use omp_lib
  implicit none
  double precision :: pie, total, x, dx, t
  integer          :: i, number_steps

end program pi
```

Implement the integration and some print statements. Name it pi-first.f90

See the OpenMP crib sheet.

Example: Computing π in Parallel

```
gfortran -O3 -fopenmp pi-first.f90 -o pi-first
export OMP_NUM_THREADS=4
./pi-first
```

and get

```
 hi from thread           0 of           4
 hi from thread           1 of           4
 hi from thread           2 of           4
 hi from thread           3 of           4
           pi =    2.2703528624655167
 elapsed time =   0.17734825599472970         seconds.
```

Whoops! Our first mistake in parallel programming. It will happen again.

Righting our Wrongs

Now we get

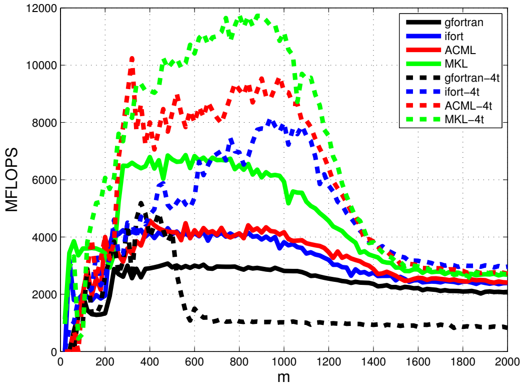
```
 gfortran -O3 -fopenmp pi-first.f90 -o pi-first
 export OMP_NUM_THREADS=1
 ./pi-second
hi from thread           0 of           1
          pi =    3.1415926453617899
elapsed time =   0.66723693604581058       seconds.
export OMP_NUM_THREADS=2
./pi-second
hi from thread           0 of           2
hi from thread           1 of           2
          pi =    3.1415926469906479
elapsed time =   0.34848214697558433       seconds.
export OMP_NUM_THREADS=4
./pi-second
hi from thread           0 of           4
hi from thread           3 of           4
hi from thread           2 of           4
hi from thread           1 of           4
          pi =    3.1415926454477359
elapsed time =   0.17908076802268624       seconds.
```

Look at that *weak scaling*! **Suggested homework: test its *strong scaling*.**

Matrix-Vector Revisited

Add OpenMP to the matvec subroutine.

FLOPS - Final Thoughts Revisited



Curiously, the best approach is to switch back *i* and *j*! See the text by Chapman et al. for details. (Note, things like MKL have threaded versions, too, so still avoid reinventing the wheel.)

The Monte Carlo Slab Problem

Homework (ha!): parallelize your program.
Given: all the code from yesterday plus a parallel-ready random number generator (if not already provided).
Solution Sketch: The solution looks a lot like the π problem above. You need to

- wrap the problem in a `parallel` directive
- allocate and initiate local tally arrays
- wrap the histories in a `do` directive
- note the global object `tal` is already given the attribute `threadprivate` (like `private` but for things in a potentially global scope such as modules)
- carefully select which variables are denoted `private` (e.g. the `mat` variable) and which are `public`
- *safely* add the local tally information to the global `tal` variable. Hint: look up the `critical` directive.

Note: I changed roughly 15 lines of code to parallelize the serial code, and most of this was in replacing `tal` with a private variable.

Summary

Key takeaways:

- Speed isn't the be all end all; memory is often limiting
- The compiler and library choice is important for peak performance
- Don't reinvent the wheel (in particular, OMILAT)
- OpenMP is easy (mostly)
- Computing is fun!

Help is available if you decide to attempt the problem...