Serial Optimization Tips

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu

Slides stolen From Yifeng Cui, Ph.D. of SDSC

Why optimize your code for single CPU performance?

- Advantages
- More calculation in a given time.
- Less cost per simulation.
- Faster time to solution.
- Better chance of successful proposals.

In the past, it was not clear if it was worth the parallel effort, as you could wait for faster uniprocessors. If you care about performance, there is now no choice but to parallelize. - David Patterson

Why optimize your code for single CPU performance?

- Optimization of serial (single CPU) version is very important
  - Want to parallelize best serial version – where appropriate
- Disadvantages
  - Time consuming – Do not waste weeks optimizing a one off code that will run for 1 hour.
  - Can make the code harder to read and debug.
  - Can adversely affect parallel scaling.
  - Different architectures can respond in different ways

Remember Wallclock is Everything

- The only metric that ultimately matters is Wallclock time to solution.
  - Wallclock is how long it takes to run your simulation.
  - Wallclock is how much you get charged for.
  - Wallclock is how long your code is blocking other users from using the machine.

Remember Wallclock is Everything

- The only metric that ultimately matters is Wallclock time to solution.
  - Wallclock is how long it takes to run your simulation.
  - Wallclock is how much you get charged for.
  - Wallclock is how long your code is blocking other users from using the machine.

You can easily write a code that gets very high Flops numbers but has a longer time to solution.

When to optimize?

- Code optimization is an iterative process requiring time, energy and thought.
- Performance tuning is recommended for:
  - Production codes that are widely distributed and often used in the research community
  - Projects with limited allocation (to maximize available compute hours).

Optimization Strategy

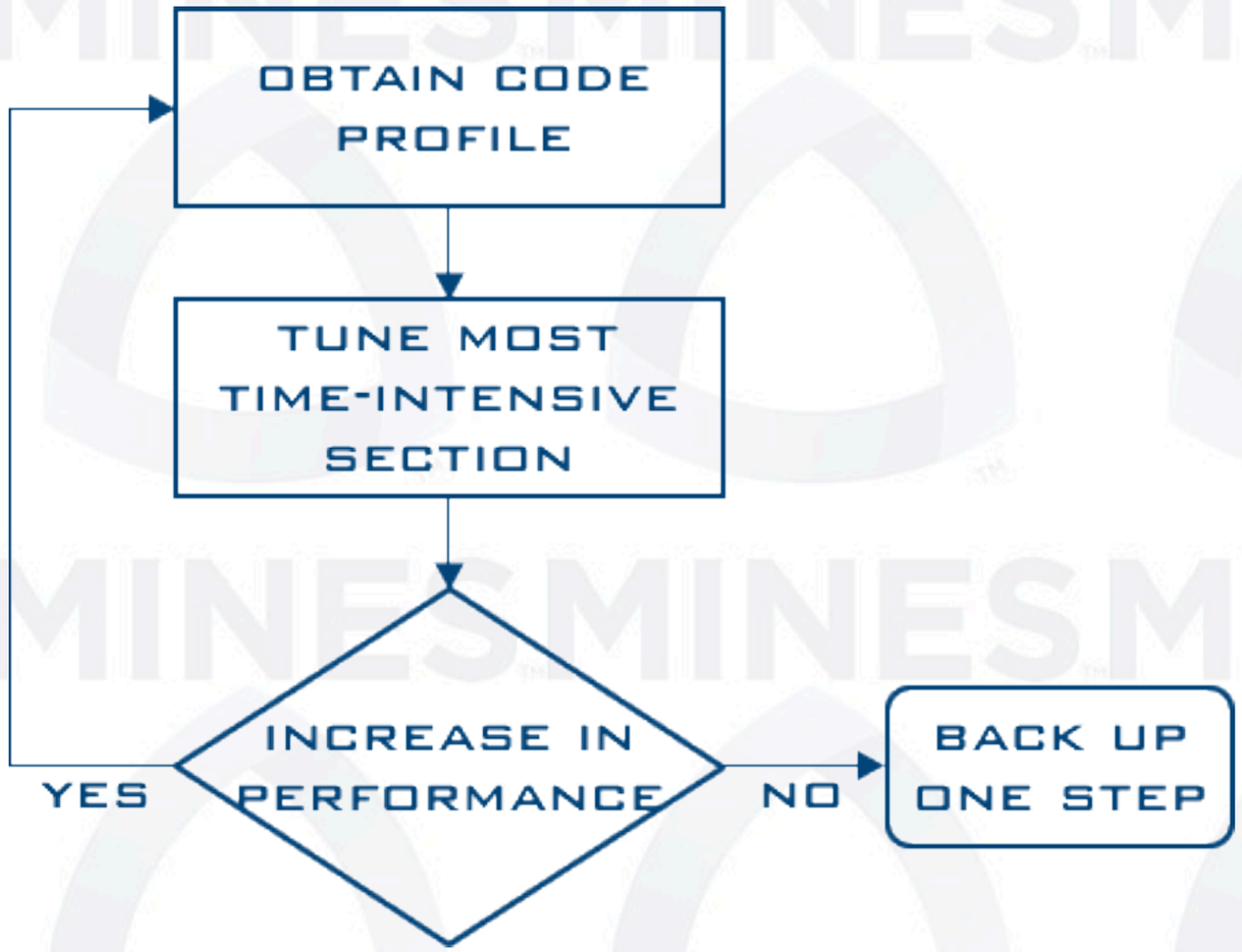
Aim: To try to minimize the amount of work the compiler is required to do.

DO NOT rely on the compiler to optimize bad code.

Use a two pronged approach:

1) Write easily optimizable code to begin with.

2) Once the code is debugged and working try more aggressive optimizations.



What compilers will not do.

- They will not reorder large memory arrays.
- They will not factor/simplify large equations.
- They will not replace inefficient logic.
- They will not vectorize large complex loops.
- They will not fix inefficient memory access.
- They will not detect and remove excess work.
- They will not automatically invert divisions.
- ...

Step 1 – Select best compiler/library

- Our program has lots of sqrt's and exponentials.
- We also have multiple loops.
- Hence it is essential that we compile with optimization and we link again IBM's accelerated math library (mass).
- xlf90 –o original.x –O3 –qarch=pwr4 –qtune=pwr4 –L/usr/lib –lmass original.f

Compiler Options	Execution time (s)
-O0	552.0
-O3	183.3
-O5	179.3
-O3 -lmass	176.8
-O5 -lmass	177.3

Math Libraries on Ra

- MKL
  - LAPACK
  - ScaLAPACK
- PETsc

How can we help the compiler?

- We should aim to help the compiler as much as possible by designing our code to match the machine architecture.
  - Try to always move linearly in memory (Cache hits).
  - Factor / Simplify equations as much as possible.
  - Think carefully about how we do things. Is there a more efficient approach?
  - Remove excess work.
  - Avoid branching inside loops.

Step 2 – Code Optimization

- Minor source code modifications
- Use of best compiler optimization options
- Use of high performance library and algorithm
- Tuning code for a particular system
- I will take you through a series of examples, some of them are from a earthquake FD model

Optimization Tip 1

- Avoid expensive mathematical operations as much as possible.
  - Addition, Subtraction, Multiplication = Very Fast
  - Division and Exponentiation = Slow
  - Square Root = Very Slow
    (Inverse square root can sometimes be faster)

Operation	Min Cycles per iteration (L1 Cache)
x(i) = y(i)	1.7
x(i) = x(i) + y(i)	1.7
x(i) = x(i) + s*y(i)	1.7
x(i) = 1/y(i)	15.1
x(i) = sqrt(y(i))	18.1

Optimization Tip 2

- Do not rely on the compiler to simplify / factorize equations for you. You should manually do this yourself. E.g.:

$$e^{R_{ij}q_i} \times e^{R_{ij}q_j} \equiv e^{R_{ij}\left(q_i + q_j\right)}$$

- This replaces 2 expensive exponentials with just 1.
- The same is true for excessive divisions, square roots etc.
- Try expanding the equation that your code represents and then refactorizing it with the intention of avoiding as many square roots, exponentials and divisions as possible.

Optimization Tip 3

- Invert divisions wherever possible.
- This replaces N^2 divisions with N divisions and N multiplications.

```
do i = 1, N
  do j = 1, N
    y(j,i)=x(j,i)/r(i)
  end do
end do
```

```
do i = 1, N
oner = 1.0d0/r(i)
  do j = 1, N
    y(j,i)=x(j,i)*oner
  end do
end do
```

Optimization Tip 4

- Avoid branching in inner loops – This leads to cache misses and also prevents the compiler from vectorizing loops and making use of the multiple floating point units.

```
do k=nbgz,nedz,nskpz
  do j=nbgy,nedy,nskpy
    do i=nbgx,nedx,nskpx
      r=r+1
      nxp = int((i-1)/nxt)
      nyp = int((j-1)/nyt)
      nzp = int((nz-k)/nzt)
      if(coords(1)==nxp .and.
         coords(2)==nyp .and.
         coords(3)==nzp) then
        if(mod(i,nxt)/=0) then
          tprec(l,1) = mod(i,nxt)
        else
          tprec(l,1) = nxt
        end if
          … …
        tpmap(l) = r
        recproc = rank
        l=l+1
      end if
    end do
  end do
end do
```

```
do k=nbgz,nedz,nskpz
  nzp = int((nz-k)/nzt)
  if (coords(3)==nzp) then
    do j=nbgy,nedy,nskpy
      nyp = int((j-1)/nyt)
        if (coords(2)==nyp) then
          do i=nbgx,nedx,nskpx
            nxp = int((i-1)/nxt)
            if (coords(1)==nxp) then
              a = mod(i,nxt)
              b = ANINT(a/(a+1))
              tprec(l,1) = a + (1-b) * nxt
              … …
              Li = int((nedx-nbgx)/nskpx) + 1
              … …
              r = Li*Lj*int((k-nbgz)/nskpz) +
                  Li*int((j-nbgy)/nskpy) + int((i-nbgx)/nskpx)
              tpmap(l) = r
              recproc = rank
              l=l+1
          endif
        end do
      endif
    end do
  endif
end do
```

Optimization Tip 5

- Ensure that loop indexes are constant while a loop is executing.

Some compilers will not recognize that i-1 is constant during the j loop and so will re-evaluate i-1 on each loop iteration. We can help the compiler out here as follows:

```
do i = 2, N
  …
  do j = 1, i-1
    …
  end do
end do
```

```
do i = 2, N
…
iminus = i-1
do j = 1, iminus
  …
end do
end do
```

Optimization Tip 6

- Avoid excessive array lookups – Factor array lookups that are not dependent on the inner loop out of the loop.
  - Some compilers will do this for you. Others will not.

```
do i = 1, N
  do j = 1,N
    ...
    sum = sum+x(j)*x(i)
  end do
end do
```

```
do i = 1, N
! x(i) becomes a scalar in the inner loo
  xi = x(i)
  do j = 1,N
    ...
    sum = sum+x(j)*xi
  end do
end do
```

Optimization Tip 7

- Always try to use stride 1 memory access.
- Traverse linearly in memory.
- Maximizes Cache hits.
- In Fortran this means you should always loop over the first array index in the inner loop

Wrong!

```
do i=1, Ni
  do j=1,Nj
    do k=1,Nk
      v=x(i,j,k)
      ...
    end do
  end do
end do
```

This way round each iteration of loop k jumps a total of i*j*8 bytes meaning it almost always misses the cache

By changing the order of the loop we can ensure that we always stride linearly through the x array.

```
do k=1, Nk
  do j=1,Nj
    do i=1,Ni
      v=x(i,j,k)
      ...
    end do
  end do
end do
```

Or best

```
Nitr = Nk*Nj*Ni
do k=1, Nitr
  v=x(k)
  ...
end do
```

Optimization Tip 8

Avoid function calls inside loop

```
subroutine do_force (i1,i2,j1,j2)
! sets the force conditions
! input is the grid and the indices for the interior cells
    use numz
    use constants, only:for,dy
    use face, only : force
    implicit none
    integer,intent(in):: i1,i2,j1,j2
    real(b8) y
    integer i,j
    do i=i1,i2
        do j=j1,j2
            y=j*dy
            for(i,j)=force(y)
        enddo
    enddo
end subroutine
```

```
function force(y)
    use numz
    use input
    use constants
    implicit none
    real(b8) force,y
    force=-alpha*sin(y*a6)
end function force
```

Optimization Tip 9

Reading input in block

```
do a=1,dims(3)
 do b=1,dims(2)
  do c=1,dims(1)
   do i=1,nzt
    do j=1,nyt
     do k=1,nxt
      m =(a-1)*nxt*nyt*nzt*dims(1)*dims(2)+ &
         (b-1)*nxt*nyt*dims(1)+ &
         (c-1)*nxt + k + r*nx + p*nx*ny  &
      read(27,'(3I5,5F9.2)',rec=m) &
           nxn,nyn,nzn,vp,vs,dd,pq,sq
      ...
     end do
    end do
   end do
  end do
 end do
end do
```

```
do a=1,dims(3)
 do b=1,dims(2)
  do c=1,dims(1)
   do i=1,nzt
    do j=1,nyt
     m = (a-1)*nyt_l*nzt_l*dims_l(1)*dims_l(2) +
     +   (b-1)*nyt_l*dims_l(1) +
     +   (c-1) +1+r*dims_l(1) + p*dims_l(1)*ny
     read(27,rec=m) tmpx
     do k=1,nxt
     xn = tmpx((k-1)*8+1)
     yn = tmpx((k-1)*8+2)
     zn = tmpx((k-1)*8+3)
     vp = tmpx((k-1)*8+4)
     vs = tmpx((k-1)*8+5)
     dd = tmpx((k-1)*8+6)
     pq = tmpx((k-1)*8+7)
     sq = tmpx((k-1)*8+8)
     nxn = transfer(xn,nxn)
     nyn = transfer(yn,nyn)
     nzn = transfer(zn,nzn)
     end do
    end do
   end do
  end do
 end do
end do
```

Vectorization

- Sometimes if you have a large amount of computation occurring inside a loop it can be beneficial to explicitly vectorize the code and make use of specific vector math libraries.

- Often the compiler can do this vectorization for you.

- Compiler can typically only vectorize small inner loops. Complex loop structures will not get vectorized. Here your code can benefit from explicit vectorization.

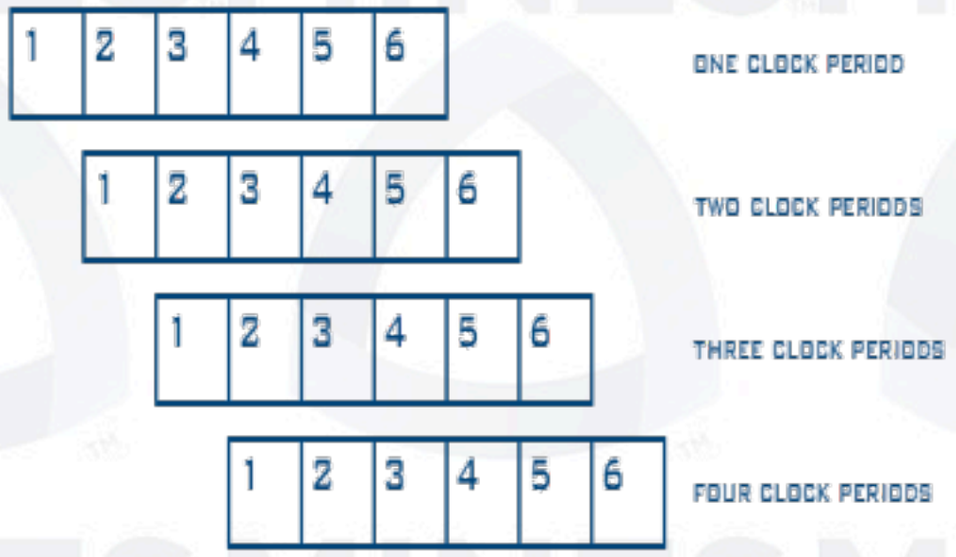
Why Does Vectorization Help on a Scalar Machine?

- Machine is actually super scalar...
- Has a pipeline for doing computation.
- Has multiple floating point units.
  - Vectorization makes it easy for the compiler to use multiple floating point units as we have a large number of operations that are all independent of each other.
  - Vectorization makes it easy to fill the pipeline.

On Power 4 a single FMA has to go through 6 stages in the pipeline.

After 1 clock cycle the first FMA operation has completed stage 1 and moves on to stage 2.

Processor can now start processing stage 1 of the second FMA operation in parallel with stage 2 of first operation etc etc...



Why Does Vectorization Help on a Scalar Machine?

After 6 operations pipelining of subsequent FMA operations gives one result every clock cycle.
Pipeline latency is thus hidden beyond 6 operations.

Power 4 chip has two floating point units so needs a minimum of 12 independent FMA operations to be fully pipelined.

Thus if we can split our calculation up into a series of long 'independent' vector calculations we can maximize the floating point performance of the chip.

Trade complexity & memory for work?

```
do i=1,steps
    call do_jacobi(psi,new_psi,diff,i1,i2,j1,j2)
enddo
```

We have put "force" into an array

How can we change calling loop to eliminate a significant memory copy?

Left as an exercise

```
subroutine do_jacobi(psi,new_psi,diff,i1,i2,j1,j2)
! does a single Jacobi iteration step
! input is the grid and the indices for the interior cells
! new_psi is temp storage for the the updated grid
! output is the updated grid in psi and diff which is
! the sum of the differences between the old and new grids
    use numz
    use constants
    implicit none
    integer,intent(in) :: i1,i2,j1,j2
    real(b8),dimension(i1-1:i2+1,j1-1:j2+1):: psi
    real(b8),dimension(i1-1:i2+1,j1-1:j2+1):: new_psi
    real(b8) diff
    integer i,j
    real(b8) y
    diff=0.0_b8
    do j=j1,j2
        do i=i1,i2
! y=j*dy
            new_psi(i,j)=a1*psi(i+1,j) + a2*psi(i-1,j) + &
                         a3*psi(i,j+1) + a4*psi(i,j-1) - &
                         a5*for(i,j)
! a5*force(y)
            diff=diff+abs(new_psi(i,j)-psi(i,j))
        enddo
    enddo
    psi(i1:i2,j1:j2)=new_psi(i1:i2,j1:j2)
end subroutine do_jacobi
```