

Portable Performance- Oriented Programming: Vectorization Emphasis

Mark Fahey, faheymr@ornl.gov

James B. White III, trey@ornl.gov



The National Center for
Computational Sciences

OAK RIDGE NATIONAL LABORATORY
U. S. DEPARTMENT OF ENERGY

Acknowledgement

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.



The National Center for
Computational Sciences

Agenda

1. Introduction
2. Programming basics
3. General optimizations
4. Advanced optimizations
5. Case studies



1. Introduction

- Why did we want to do this
 - Share our knowledge of porting and optimizing
 - Prevent mistakes in code development and/or maintenance
 - Expose good programming techniques for any language
- Where we are coming from
 - Combined 20+ years of writing, porting, and optimizing HPC applications on massively parallel supercomputers
 - Our definitions/viewpoints may be debatable



Insightful remarks?

- “The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.” [Dijkstra]
- “Barring some real breakthroughs in compiler technology, the computers of the 2000's will be even more finicky than the computers of the 1990's.” [Dowd]
- “The true problem with software is hardware. ... We have been shielded by hardware advances from confronting our own incompetence as software professionals and our immaturity as an engineering profession.” [Constantine]



1.1 Portability

- Porting to new platforms should only require setting compiler, libraries, *etc.*
 - Port should take $O(\text{minutes})$
- One source
 - Don't want two or more versions of any routine
 - Platform-specific “code” should only be compiler, library settings
 - Avoid high maintenance cost of moving a tuned code from one architecture to another
- Runs correctly on all computers



1.1 Portability

- Potential by-products
 - Minimal, localized machine-specific code
 - Minimal `#ifdefs`
 - Code readability
 - Minimal use of non-standard libraries
 - Unless performance gain is huge
 - Port to any machine in minutes
 - With some exceptions
 - **Lower performance**



1.1 Portability

- Stroustrup said:
 - “If your program is a success, it is likely to be ported, so someone will have to find and fix the problems related to implementation-dependent features.”
 - “Constructing programs so that improvements can be implemented through local modifications only is an important design aim.”
 - Yes, he is the creator of C++
 - Philosophy still applies



1.2 Performance

- Tuning your code to make it run fast
 - Spending as much real time as you want
- Taking advantage of fastest I/O, communication, and numerical libraries
- Willing to make changes to the code
 - Tuned for memory hierarchy and processors
 - Might be slower on other computers



1.2 Performance

- Potential by-products
 - Much machine-specific code
 - Spread across code base
 - Multiple overlapping source trees
 - Numerous `#ifdefs`
 - Unreadable code segments
 - Use of non-standard libraries
 - Ports can take days/weeks
 - Upgrades to primary machine may require whole new round of optimizations (to possibly undo previous opts)



1.2 Performance

- Alpern and Carter said:
 - Performance programming
 - Design, writing, and tuning of programs to keep processing elements as busy as possible doing useful work
 - Improve performance beyond what is achieved by programming an algorithm in most expedient manner
 - Beyond selecting algorithms with good asymptotic complexity (not discussed today), requires acute sensitivity to details of processor architecture and memory hierarchy



1.3 Portable performance

- Keep most machine-specific content in `make.inc` and possibly one source-code file (utilities)
- Leverage significant performance optimizations
- Employ poly-algorithms
 - Choose between algorithms at runtime
- Sacrifice some performance
 - Leave out small improvements disruptive to source
- Don't mess up the source code “too much”
- Use optimized vendor libraries when it makes sense
- **Must be willing to spend effort on optimizations to see what works for multiple machines**



1.3 Portable performance

- Intended consequences
 - Still port in minutes/hours rather than days/weeks
 - Code still readable
 - Pretty fast on most machines
 - At least the ones where it matters most
 - Performance tuning takes more work
 - Must test multiple machines



1.3 Portable performance

“...we can only afford to optimize (whatever that may be) provided that the program remains sufficiently manageable.” [Dijkstra]



1.4 Maintainability

- What is maintainability? Easy to:
 - Understand what the code does
 - Change your mind about design decisions
 - Add functionality
 - Uncover and fix bugs
- Typically much more time is spent maintaining code than writing new code
 - OO isn't a panacea
 - See quotes on next slide



1.4 Maintainability

- Hatton said, in one of few studies **measuring** effects of OO design:
 - “a valid silver bullet for software must lead to a massive reduction in maintenance, which is by far the life cycle’s biggest component”
 - “no significant benefits accrued from the use of an OO technique in terms of corrective-maintenance cost and the company views the resulting C++ product as much more difficult to correct and enhance”
- More light-heartedly, Kent Beck at OOPSLA '05:
 - "People who could not do a decent job with structured design went to objects so no one would notice.”



1.4 Maintainability

- Important issue
 - Needs its own tutorial
- But today we focus on portable performance-oriented programming



1.5 Fortran

- The “F” word, or “undead language”
- Why use Fortran?
 - 32% of all users of engineering and scientific workstations worldwide write in Fortran [Willard]
 - Native language of many DOE and DOD apps (new and old)
 - Compiler technology is mature
 - Minimizes dependencies, maximizes optimizability
 - Built-in arrays and simple data structures make programs simpler to parallelize
- Destined to be replaced by:
 - Algol, PL1, Pascal, Ada, C, C++, Java, ...
 - Matlab, Maple, Mathematica, ...
 - Chapel, X10, Fortress



Portable Performance-Oriented Programming

- In what follows:
 - Present basic principles of code writing and portability up front
 - Then present various optimizations with portability in mind



Agenda

1. Introduction
2. Programming basics
3. General optimizations
4. Advanced optimizations
5. Case studies



2. Programming basics

Consider the following when developing code

1. Generic principles

- Diagnostics
 - Internal timers
 - Debug checks
- Consistent programming style

2. Portability techniques

- Preprocessing
- Modules
- Modularization
- Checkpoint/restart (often a must)
- Interoperability

3. Programming for the Future



2.1 Generic principles

- Use diagnostics
 - Verbosity
 - Timers
 - Checks
- Verbosity (helps in debugging)
 - Might want multiple levels of verbosity
 - Input flag should control this

```
if (verbose_flag == 1 .and. iam == 0) then
  print *, ' time step =',time_step
  print *, '**[subA done]'
```

endif
 - Limit so performance is not adversely affected
 - Strive for near negligible (in terms of CPU time) amount of `if` tests



2.1 Generic principles

- Internal timers
 - Time major phases
 - Print out stats every m timesteps and summary at completion
 - if (modulo(step,time_skip) == 0) then
 - call write_timing('timing.out',10,mode)
 - end if
 - Get your own profile (at this level of granularity)
 - Know costly parts of code without having to use new tools



2.1 Generic principles

- Internal checks
 - Functionality/correctness
 - Check return arguments, lengths of arrays, *etc.*
 - Compile in with macro (`_DEBUG?`)
 - Will slow code down
 - So best to be a compile-time option
 - Example:

```
#ifdef _DEBUG
    if (debug_flag .EQ. 1) write(*,*) ' x= ', x
#endif
```



2.1 Generic principles

- Internal checks

- “I spent a lot of time talking about how not to need a debugger in the first place. If you know something that has to be true in your code, assert it.”
Kate Hedstrom, ARSC HPC Newsletter 326
- The following example is not a C++ assertion, but similar in spirit

```
#ifdef _DEBUG
    if((i .lt. lbound(arrayA,dim=1)) .or. &
        (i .gt.ubound(arrayA,dim=1))) then
        write(*,*) "i outside range of arrayA:  i=", i
        stop
#endif
```



2.1 Generic principles

– Numeric checks

- Watch out for catastrophic cancellation
 - When operands nearly cancel one another out
- Effects of catastrophic cancellation can easily be magnified
 - `sqrt(1-x)`: possible loss of half significant digits
 - If `x` is nearly 1, then `sqrt(1-x)` should be 0

- Example (Gyro): `make_omegas.f90`

```
temp = sqrt(abs(energy*(1.0-lambda(i,k)*b0(i,k,m))))  
if (abs(temp) < 1e-5) e_temp_p = 0.0
```

- Output norms, like `MPI_reduce(sum(abs(x)))`
 - Helpful in debugging wrong answers
 - Or check that a norm is within an expected range
- Check return values from math library calls



2.1 Generic principles

- Consistent programming style
 - Easy to read, easy to do search/replace
 - Indenting (use spaces)
 - Use descriptive variable names (don't get carried away though)
 - Comment-based data structures
 - Group variables and described them with comments
 - Similarly, a loop structure or other code segment may be described by one comment



2.2 Portability techniques

- Preprocessing
 - Minimize and localize
 - At odds with compile-time debugging checks
 - Use meaningful names
 - LINUX too general
 - NEED_UNDERSCORE or ADD_ better
 - “Almost every macro demonstrates a flaw in the programming language, in the program, or in the programmer.” [Stroustrup]
 - That may be, but still a necessary evil
 - “If you must use macros, use ugly names with lots of capital letters” [Stroustrup]
 - Ugly → meaningful
 - Possibly start and end with “_”
 - *What about system-defined architecture-specific macros?*



Conditional compilation

- Various controls can easily combine in unforeseen ways
 - Thus the advice to minimize and localize
- If you use `#ifdef` for machine dependencies
 - Make sure that when no machine is specified, the result is an error, not a default machine
 - `#error` directive is useful for this purpose
- If you use `#ifdef` for optimizations
 - Default should be unoptimized code rather than an uncompilable or incorrect program
 - Be sure to test the unoptimized code



Modules

- Benefits
 - Code reusability
 - Type checking
- Kinds module
 - Define kinds in one place, and use throughout code
 - If changed, will require whole code to be compiled, which is what you want
 - It does not change input files or MPI data types though!
 - **Advice: use only if you expect to need different kinds**
- Cross-dependent source files - legal but not nice to some compilers that try to do inlining
 - File X contains module A and C, and A “use”s B
 - File Y contains module B which “use”s C



Modularization

- Modularizing communication at roughly the block-synchronous level
 - Your own communication library
 - Some leeway for optimization since some “physics” is still included
 - Aim for potential overlapping communication and computation
 - Co-Array Fortran naturally gives you overlapping communication and computation
- Similarly, modularize I/O at a block level
- Wrap low-level system utilities, keep in utilities file that is easily modified when porting
- Note for MPI codes:
 - Assume your code may be a piece of a larger code someday → don't use `MPI_COMM_WORLD`
 - Make your own world communicator
Ex: Duplicate `MPI_COMM_WORLD` to `my_world`



Utility example

- Put the following in a utilities file
- If porting issues arise, only fix one thing/file

```
subroutine execute_command(cmd)
```

```
    character(*), intent(in)  :: cmd
```

```
# ifdef SGI || SP2 || CPQ  
    call system(trim(cmd))
```

```
# endif
```

```
# ifdef T3E || X1  
    call ishell(trim(cmd))
```

```
# endif
```

```
end subroutine execute_command
```

← good macros?



Does ifdef code
follow earlier advice?



Checkpoint/restart

- Usually a must
- Lets you use an unstable system from day one
- At allocated sites, can result in bonus hours if machine crashes during run
- Consider ramifications of how you do this
 - Unformatted or formatted
 - 1 file or many files
 - 1 checkpoint or checkpoints every m steps
 - **1 checkpoint is never enough**
Do at least 2, current and previous
 - The answers to these may depend on the filesystems and/or machine
- Do you need files to be portable?
 - Big endian/little endian (often a compile-time option for I/O)
 - HDF5, NetCDF



C/Fortran interoperability

- Minimize C/Fortran interoperability
 - Porting can be troublesome
- If required (e.g. Fortran program calling C library)
 - Localize interactions
 - Keep interface in a easily recognizable file to be reviewed when porting
- **Or** use modern interoperability features
 - Standard C interoperability (Fortran 2003)
 - Allows Fortran programs to call C functions and access C global objects
 - And *vice versa*
 - `ISO_C_BINDING` module provides interoperable kind parameters for C types and Fortran intrinsic types
 - **Requires** modern compiler (Fortran standard compliant)



Interoperability example

```
$ more libc_defs.f
module libc_defs
use,intrinsic iso_c_binding

interface
  function kill(pid, sig),bind(c) result(return_val)
  import c_int, c_int32_t
  integer(c_int)          :: return_val
  integer(c_int32_t),value :: pid
  integer(c_int),      value :: sig
end function kill

  function getpid(),bind(c) result(pid)
  import c_int32_t
  integer(c_int32_t)      :: pid
end function getpid
```



Interoperability example

```
function system(syscall),bind(c) result(rval)
import c_int,c_char
CHARACTER(len=*, KIND=c_char) :: syscall
integer(c_int) :: rval
end function system
```

```
function sleep(seconds),bind(c) result(rval)
import c_int
integer(c_int) :: rval
integer(c_int),value :: seconds
end function sleep
```

```
end interface
```

```
end module libc_defs
```



Interoperability example

```
$ more tst.f
  use libc_defs
  use,intrinsic iso_c_binding

  integer(c_int)      :: sig,res
  integer(c_int32_t) :: pid

  pid = getpid()
  sig = 9
  res = kill(pid,sig)

end
```



2.3 Programming for the future

- Gate counts keep increasing
 - Floating-point units get cheaper
 - More fine-grained parallelism
- Clock-speed increases are stalling (Heat!)
- Bandwidth may be catching up
 - Wire signal rates continue to increase
 - Optical communication will get cheaper
- Programming implications
 - Clearly present fine-grained parallelism
 - Allow latency hiding (local and remote)



2.3 Programming for the future

- Operate on adjustable sub-aggregates (blocks, tiles, *etc.*)
 - Not scalars (to allow vectorization and pipelining)
 - Not the whole domain (to allow caching)
- Avoid false dependencies
 - Pointers!
 - I/O statements inside loops (for debugging)



2.3 Programming for the future

- Use modules instead of passing arguments (if you always pass the same object)
 - Easier promotion of scalar procedures
 - Easier promotion of variables to co-arrays (Fortran 2008)
 - Compilers can “see” the variables better
 - Adding “arguments” is a local modification (not throughout call stack)
- Use modules instead of user-defined types
 - Easy promotion of variables to co-arrays
 - Avoid artificial dependencies
 - Encourage operations on aggregates
 - Simpler for others to understand
 - Simpler for compilers to understand



Modules caveat: side effects

- Assume
 - sub1 gets x and y passed in as arguments
 - sub1 calls sub2
 - Sub2 has some arguments (not x and y)
 - Sub2 uses x and y imported via modules and modifies y
- Then
 - cannot easily tell when looking at sub1 what might be changed in sub2 (side effects)
 - Not consistent in how vars are passed, confusing
 - Modules can hide information from the reader



Side-effects example

- VMEC2000 (fusion)

```
SUBROUTINE sweep3_blocks (xc, xcdot, gc, nmax_jog)
  USE vmec_main, ONLY: r01, z01
  REAL(8), DIMENSION(ns,0:ntor,0:mpoll,ntyptot) :: &
    xc, xcdot, gc, xstore
  CALL FUNCT3D(istat)
  xstore = xc
  N2D: DO n_2d = 0, ntor
    M2D: DO m_2d = 0, mpoll
      DO i = 1, nsize
        js = radial_pts(i)
        xc(js,n_2d,m_2d) = xstore(js,n_2d,m_2d) + hj
        xcdot(js,n_2d,m_2d) = hj
      ENDDO
      CALL FUNCT3D(istat)           ! xc is input, gc is output
      xc = xstore
      xcdot = 0
      ! gc is used to update other arrays not shown
    ENDDO
  ENDDO
```



Side-effects example

- Code is legal, but hard to figure out
- The comments aren't there in the real code
 - Should be!
Necessary for understanding



Agenda

1. Introduction
2. Programming basics
3. **General optimizations**
4. Advanced optimizations
5. Case studies



3. General optimizations

- Focus on single-processor performance
- Use following strategies (in order of increasing effort and difficulty)
 - Minor source code modifications
 - Best compiler optimization options
 - High-performance library and algorithm
 - Tuning code for a particular system
- Will not cover compiler options, libraries, or algorithms here



3. General optimizations

1. Removing clutter [Dowd]

- Subroutine overhead
- Branches
- Other

2. FP/loop optimizations

- Unrolling, *etc.*

3. Data locality

- Blocking/clumping
 - BLAS 2 and 3 - careful about overhead
- Array re-indexing
- Ambiguity in memory references

4. Directives



3.1 Removing clutter

- Subroutine overhead
 - Very large on vector machines, prevents vectorization → very important
 - Also a factor on superscalar machines
- Two techniques
 1. Some compilers can do automatic inlining
 - Further gains can be had by doing it yourself
 - Manual inlining is not necessarily recommended
 2. Push loops down into subroutines
 - Eliminates subroutine overhead and allows for more efficient vectorization in the subroutines
 - Will look at this more later



3.1 Removing clutter

```
do j = 1,n
  if (test(j) .eq. 1) then
    do i =1,n
      a(i,j) = a(i,j) + b(i,j)
    enddo
  else
    call STOP_PROGRAM(); endif
enddo
```

- Call to STOP_PROGRAM prevents parallelization

```
do j = 1,n
  if (test(j) .eq. 1) call
    STOP_PROGRAM()
enddo
do j = 1,n
  do i =1,n
    a(i,j) = a(i,j) + b(i,j)
  enddo; enddo
```

- STOP_PROGRAM almost never called, separate it
- “a” does not end up the same



3.1 Removing clutter

- Manual inlining example: S3D

```
        DO 90 K = 2, KK
#ifdef VECTORVERSION
C   Manually inline for X1.
        DO J = 1, K-1
            DJK(J,K) = (((COFD(4,J,K) * ALOGT) +
$                               COFD(3,J,K) ) * ALOGT +
$                               COFD(2,J,K) ) * ALOGT +
COFD(1,J,K)
        ENDDO
#else
        CALL MCEVAL4 (ALOGT, K-1, COFD(1,1,K), DJK(1,K) )
#endif
    90    CONTINUE
```

← Is this a good macro name?

- Compiler could inline MCEVAL4
 - But doing it manually yielded even more speedup



3.1 Removing clutter

- Both techniques can result in
 - More efficient code on most machines
 - More-readable **or** less-readable code →
Be careful in their use



3.1 Rearrange clutter?

```
do j = 1,n2
  do i = 1, n1
    a(i,j) = a(i+1,j+1) +
    LARGE_FUNCTION(b,c,d,..)
  enddo; enddo
```

- Inner loop will vectorize
- Nothing will stream

```
do j = 1,n2
  do i = 1, n1
    atemp(i,j) =
    LARGE_FUNCTION(b,c,d,..)
  enddo; enddo
do j = 1,n2
  do i = 1, n1
    a(i,j) = a(i+1,j+1) + atemp(i,j)
  enddo; enddo
```

- Inner loops will vectorize: Most of the work streams
- Potentially uses more memory



3.1 Removing clutter

- Branches
 - Be clear and concise with conditionals
 - Put most likely to fail/pass test first for and/or tests, respectively
 - Don't be too wordy, don't be redundant
 - Within loops
 - Loops with `if` tests can vectorize, but still best to move them out if at all possible
 - There are ways to deal with some `if`-tests in loops
 - See Dowd or Goedecker
 - “you don't want anything inside a loop that doesn't have to be there, especially an `if`-statement,” [Dowd]
- We'll talk about “filters” later



3.1 Removing clutter

- Data type conversions
 - Cost several instructions
 - Remove superfluous mixing of datatypes
- Sign conversions
 - Remove superfluous conversions
 - A sign conversion can take several cycles
- Fortran copy overheads
 - Passing a slice (substructure) of an array often copies the data into a work array (memory bandwidth)



3.1 Removing clutter

- Floating-point exceptions
 - Handled differently by vendors
 - Execution may stop, or continue with nonnumeric values
 - Execution can be much slower with NaNs
 - Might be result of incorrect programming, or result of compiler optimizations
- Recommendation is this must be watched out for
 - Either with internal checks in code (compile- or run-time) or compiler switches
 - If it happens, your code can run extremely slow



3.2 FP/Loop optimization

- Loop unrolling
 - Positives
 - Exposes parallelism by fattening up the loop
 - Potential negatives
 - Unrolled by wrong factor (machine dependent)
 - Register spilling
 - Instruction-cache misses
 - Other hardware delays
 - Shared memory machines: false sharing
 - Less readable (unless using directives)
- Don't do this manually
 - Use directives instead



Loop-unrolling example

```
if(na.gt.40*nb) then
!DIR$ PREFERSTREAM
  do ia=1,na
!DIR$ SHORTLOOP
  do ib=1,nb,4
    sum00 = (0.0,0.0)
    sum01 = (0.0,0.0)
    sum02 = (0.0,0.0)
    sum03 = (0.0,0.0)
!DIR$ PREFERVECTOR
  do ic=1,na
    sum00 = sum00 + Xj(ib,ic)*AA(ic,ia)
    sum01 = sum01 + Xj(ib+1,ic)*AA(ic,ia)
    sum02 = sum02 + Xj(ib+2,ic)*AA(ic,ia)
    sum03 = sum03 + Xj(ib+3,ic)*AA(ic,ia)
  enddo
  XjAA(ib,ia) = sum00
  XjAA(ib+1,ia) = sum01
  XjAA(ib+2,ia) = sum02
  XjAA(ib+3,ia) = sum03
  enddo
enddo
```

```
...
! also have the remainder case
  do ib=nb-mod(nb,4)+1,nb
...
else
  XjAA(1:nb,1:na) = matmul( Xj(1:nb,1:na),
    AA(1:na,1:na) )
endif
```

Don't do this!



3.2 FP/Loop optimization

- Associative transformations
 - Numerically not equivalent (potential to alter answers)
 - Vector reduction
 - Calculate several iterations at a time independently, or
 - Calculate partial sums then assemble
 - Usually done by compiler at high optimization levels or in optimized math libraries



3.2 FP/Loop optimization

- Loop interchange
 - Rearrange loop nest so the right stuff is at the center
 - Swap high trip counts for low
 - Increase parallelism (via unrolling)
 - Improve memory-access patterns
 - Unit-stride access
 - Reuse cache and registers
 - See next section



3.3 Data locality

- Memory access is a major bottleneck on machines with a memory hierarchy
- Optimizing memory access has a large potential for performance improvements



3.3 Data locality

- Potential optimization issues
 - Strides
 - Loop reordering for optimal locality
 - Loop fusion to reduce unnecessary memory references
 - Data structures
 - Blocking
 - Cache thrashing
 - Ambiguity in memory references



Strides

- Unit stride is still the best
 - Conserves cache entries
- Can't eliminate strided memory accesses
 - Try restructuring loops to minimize cache and TLB misses
 - Try not to get too ugly



Beware low trip counts

- Assume $n_1=n_2>100$, $n_3<20$

```
do j = 1,n2
  do i = 1,n1
    do k = 1, n3
      atemp( k ) = f(i,j,k)+...
    enddo
  do k=1,n3
    c(i,j,k) = c(i,j,k) + atemp(k)+...
  enddo
enddo; enddo
```

- k loop parallel; i, and j are not
- Short trip count on k makes code less efficient

- Promote atemp

```
do j = 1,n2
  !dir$ prefervector
  do i = 1,n1
    do k = 1, n3
      atemp( i,j,k ) = f(i,j,k)+...
    enddo
  do k=1,n3
    c(i,j,k) = c(i,j,k) + atemp(i,j,k)+...
  enddo
enddo; enddo
```

- Now i and j parallel; much more efficient
- **Increased memory usage**



Loop reordering

- Two aspects:
 - Interchanging loops
 - Simple (possibly just a directive)
 - But usually not enough
 - Swapping array indices
 - If declared in a module, could be quite simple to do (assuming Fortran array syntax)
 - In general, tedious and error prone



Loop interchange

- Some compilers can interchange loops
 - May need to use directive

```
ir-----<          do j = 1,200
ir MVs--<           do i = 1,200
ir MVs              a(i) = a(i) + b(i,j) * c(j)
ir MVs-->           end do
ir----->          end do
```

- X1E compiler can “hoist” $a(i)$ after interchange



Loop interchange

- Before

```
do nn=0,n_max
  do i=1,n_x
    do n1=-n_max+nn,n_max
      ! f dg/dr - g df/dr
      fgr(nn,i) = fgr(nn,i)+&
        fn(n1,i)*gn_r(nn-n1,i)-&
        gn(n1,i)*fn_r(nn-n1,i)
      ! g df/dp - f dg/dp
      afgp(nn,i) = fgp(nn,i)+&
        gn(n1,i)*fn_p(nn-n1,i)-&
        fn(n1,i)*gn_p(nn-n1,i)
      ! df/dp dg/dr - df/dr dg/dp
      fg2(nn,i) = fg2(nn,i)+&
        fn_p(n1,i)*gn_r(nn-n1,i)-&
        fn_r(n1,i)*gn_p(nn-n1,i)
    enddo ! n1
  enddo ! i
enddo ! nn
```

n_max=63 and n_x=400

- After

```
do i=1,n_x
  do nn=0,n_max
    do n1=-n_max+nn,n_max
      ! f dg/dr - g df/dr
      fgr(nn,i) = fgr(nn,i)+&
        fn(n1,i)*gn_r(nn-n1,i)-&
        gn(n1,i)*fn_r(nn-n1,i)
      ! g df/dp - f dg/dp
      afgp(nn,i) = fgp(nn,i)+&
        gn(n1,i)*fn_p(nn-n1,i)-&
        fn(n1,i)*gn_p(nn-n1,i)
      ! df/dp dg/dr - df/dr dg/dp
      fg2(nn,i) = fg2(nn,i)+&
        fn_p(n1,i)*gn_r(nn-n1,i)-&
        fn_r(n1,i)*gn_p(nn-n1,i)
    enddo ! n1
  enddo ! nn
enddo ! i
```

1.2x faster on X1E
2x faster on XT3



Index swap

- Gyro before

```
complex, dimension(-n_max:n_max,n_x) :: fn, fn_r, gn, gn_r
do i_diff=-m_dx,m_dx
  do i=1,n_x
    do nn=0,n_max
      fn_r(nn,i) = fn_r(nn,i)+w_d1(i_diff)*fn(nn,i+i_diff)
      gn_r(nn,i) = gn_r(nn,i)+w_d1(i_diff)*gn(nn,i+i_diff)
    enddo ! nn
  enddo ! i
enddo ! i_diff
do i=1,n_x
  do nn=1,n_max
    fn_r(-nn,i) = conjg(fn_r(nn,i))
    gn_r(-nn,i) = conjg(gn_r(nn,i))
  enddo ! nn
enddo ! i
```



Index swap

- Gyro after

```
complex, dimension(n_x,-n_max:n_max) :: fn, fn_r, gn, gn_r
do i_diff=-m_dx,m_dx
  do i=1,n_x
    do nn=0,n_max
      fn_r(i,nn) = fn_r(i,nn)+w_d1(i_diff)*fn(i+i_diff,nn)
      gn_r(i,nn) = gn_r(i,nn)+w_d1(i_diff)*gn(i+i_diff,nn)
    enddo ! nn
  enddo ! i
enddo ! i_diff
do i=1,n_x
  do nn=1,n_max
    fn_r(i,-nn) = conjg(fn_r(i,nn))
    gn_r(i,-nn) = conjg(gn_r(i,nn))
  enddo ! nn
enddo ! i
```



Loop fusion

- Fusing loops together can result in better reuse of loaded data
- Idea is to issue as few loads of array elements as possible before storing results and flushing the cache
- Many compilers do this at highest optimization levels



Loop fusion, before

```
do i=1,n_x
  do nn=0,n_max
    fn_p(nn,i) = -i_c*n_p(nn)*fn(nn,i)
    gn_p(nn,i) = -i_c*n_p(nn)*gn(nn,i)
  enddo
enddo
fn_r = (0.0,0.0)
gn_r = (0.0,0.0)
do i_diff=-m_dx,m_dx
  do i=1,n_x
    do nn=0,n_max
      fn_r(nn,i) = fn_r(nn,i) + &
        w_d1(i_diff)*fn(nn,i+i_diff)
      gn_r(nn,i) = gn_r(nn,i) + &
        w_d1(i_diff)*gn(nn,i+i_diff)
    enddo ! nn
  enddo ! i
enddo ! i_diff
```

```
x_fft(:,:) = (0.0,0.0)
do i=1,n_x
  do nn=0,n_max
    x_fft(nn,i) = fn(nn,i)
    x_fft(nn,n_x+i) = gn(nn,i)
    x_fft(nn,2*n_x+i) = fn_p(nn,i)
    x_fft(nn,3*n_x+i) = gn_p(nn,i)
    x_fft(nn,4*n_x+i) = fn_r(nn,i)
    x_fft(nn,5*n_x+i) = gn_r(nn,i)
  enddo
enddo
```



Loop fusion, after

```
x_fft(:, :) = (0.0, 0.0)
do nn=0, n_max
  do i=1, n_x
    fn_r = (0.0, 0.0)
    gn_r = (0.0, 0.0)
    do i_diff=-m_dx, m_dx
      fn_r = fn_r + w_d1(i_diff) * fn(nn, i+i_diff)
      gn_r = gn_r + w_d1(i_diff) * gn(nn, i+i_diff)
    enddo ! i_diff
    fn_p = -i_c * n_p(nn) * fn(nn, i)
    gn_p = -i_c * n_p(nn) * gn(nn, i)
    x_fft(nn, i) = fn(nn, i)
    x_fft(nn, n_x+i) = gn(nn, i)
    x_fft(nn, 2*n_x+i) = fn_p
    x_fft(nn, 3*n_x+i) = gn_p
    x_fft(nn, 4*n_x+i) = fn_r
    x_fft(nn, 5*n_x+i) = gn_r
  enddo
enddo
```

Reduced memory-
bandwidth requirement

Moral: Might need to
combine techniques



Blocking

- Retrieve as much data as possible with as few cache misses as possible
- Rearrange loop nests to work on neighborhoods of data - *blocks or submatrices*
- Block size (blocking parameter) depends on the cache size or vector length - **machine dependent**
- Design resulting code to be portable
 - Make block size an input or compile-time parameter
- **WARNING: Don't write hand-coded versions of common computational kernels if more efficient implementations exist.**



Matrix-multiplication example*

```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
  do jj=1,n,nb
    do kk=1,n,nb
      do i=ii,min(n,ii+nb-1)
        do j=jj,min(n,jj+nb-1)
          do k=kk,min(n,kk+nb-1)
            c(i,j)=c(i,j)+a(j,k)*b(k,i)
          end do
        end do
      end do
    end do
  end do
end do
```

*** Required in any performance tutorial.
(Use BLAS3 instead!)**



Blocking example: CLM

- Community Land Model
- Pass loops bounds to physics routines
- Introduce new outer loop with large stride
 - Use loop index and stride to define array blocks
 - Tunable for different systems
 - Small blocks for cache-dependent superscalar systems
 - Full-size blocks for vector-only systems
 - Large blocks for vector systems with additional dimensions of parallelization (threads/streams)
 - Implicitly controls the vector length



Blocking example: CLM

```
nclumps = get_proc_clumps()  
do nc = 1, nclumps  
  call get_clump_bounds(nc, ...,  
    begc, endc, ...)  
  
  ...  
  
  call Hydrology1(begc, endc, ...)  
  
  ...  
  
end do
```



Cache thrashing

- Effective size of cache is much smaller than physical size because of mapping rules and access pattern
 - For example, direct mapping or set associative
- Memory references are mapped to same set of cache slots while other slots remain unused
- FFTs, multipole methods, wavelet transforms where leading dimensions are a high power of 2
- Padding arrays usually fixes the problem



Reference ambiguity

- Difficult for compiler to distinguish from other, possibly conflicting references
- Compiler cannot determine if two index expressions point to the same location
 - Can't tell → can't optimize
 - Prevents parallelism
- Use directives
- See filters, next section



3.4 Directives

- Easy way to give compiler more information so it can do its job
- Mostly portable
 - Just comments
 - Some vendors' compilers recognize other vendors' directives
 - Could be a gotcha?



Agenda

1. Introduction
2. Programming basics
3. General optimizations
4. **Advanced optimizations**
5. Case studies



4. Advanced optimizations

1. Pushing loops down
2. Data structures
3. Filters
4. False dependencies
5. Vector replication



4.1 Pushing loops down

- Push loops down into subroutines
 - Eliminates subroutine overhead and allows for more efficient vectorization in the subroutines
 - Examples: Gyro, S3D, CLM



4.1 Pushing loops down

- Gyro before

```
complex :: RHS_overshoot, RHS_drift, RHS_star
[...]
! PERIODIC
do i=1,n_x
  do m=1,n
    m0 = m_phys(ck,m)
    call manage_overshoot(fh0(m,i),RHS_overshoot)
    RHS_drift = o_d1(m0,i,p_nek_loc,is)*fh(m,i)
    RHS_star = o_star(in_1,ie,is,i)*density(is,i)*&
      gyro_u(m,i,p_nek_loc,is)
    RHS(m,i,p_nek_loc,is) = RHS(m,i,p_nek_loc,is)+&
      RHS_overshoot+i_c*(RHS_drift-RHS_star)
  enddo ! m
enddo ! i
```



4.1 Pushing loops down

- Gyro after

```
complex, dimension(n,i1:i2) :: RHS_overshoot
complex :: RHS_drift, RHS_star
[...]
! PERIODIC
call manage_overshoot(fh0,RHS_overshoot)
do i=1,n_x
  do m=1,n
    m0 = m_phys(ck,m)
    RHS_drift = o_d1(m0,i,p_nek_loc,is)*fh(m,i)
    RHS_star = o_star(in_1,ie,is,i)*density(is,i)*&
      gyro_u(m,i,p_nek_loc,is)
    RHS(m,i,p_nek_loc,is) = RHS(m,i,p_nek_loc,is)+&
      RHS_overshoot(m,i)+i_c*(RHS_drift-RHS_star)
  enddo ! m
enddo ! i
```



4.1 Pushing loops down

- Portability comments
 - Increased memory usage
 - `RHS_overshoot` from scalar to 2D array
- Performance comments
 - Huge win on vectors
 - Same speed or faster on superscalars
- Otherwise
 - No harder to read/understand code
 - No harder to port
 - No machine-specific code
 - `manage_overshoot` now works on arrays instead of scalars

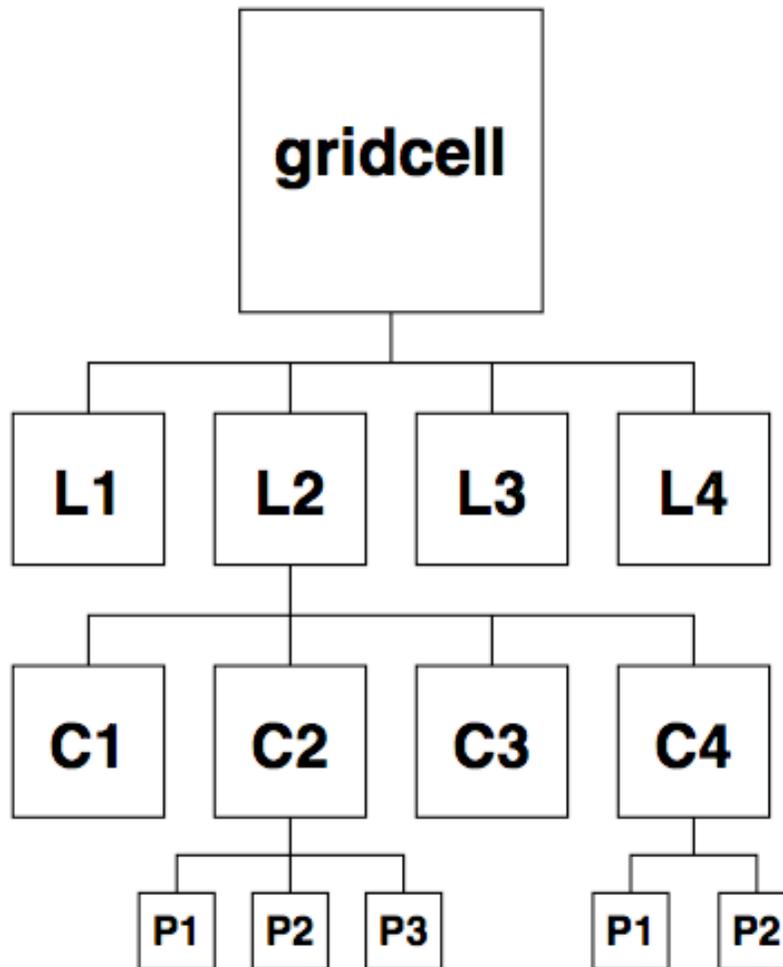


4.2 Data structures

- Data structures may prevent optimizations
 - Arrays of pointers to derived types
 - Variables implemented as scalars in each instance of a derived type
 - Science routines called for each grid or subgrid
- Pros?
 - Object-oriented design
 - Not too bad on cache-based scalar platforms
- Cons
 - Leads to large, unpredictable strides
 - Not conducive to vector processing or superscalar processing



CLM data structure



Gridcells:

Computational grid shared with the atmosphere model.

Landunits:

Geomorphologically distinct land cover types (glacier, lake, crop, urban, etc.).

Columns:

Water, snow, and soil state variables.

Plant functional types:

Vegetation state variables. PFTs may compete for column-level resources.

[Hoffman, 2005]



The National Center for
Computational Sciences

CLM 2.1

- Arrays of pointers to derived types
- Outer loops over each element
- Many `if` tests
- Strided memory accesses
- Unvectorizable



CLM vector prototype

- Prototype only implemented for part of model
 - See CUG 2003 paper
- Arrays grouped in modules
 - No derived types
 - Index arrays implement hierarchy
- Outer loops over “clumps” of elements (shown earlier)
- Scalar blocks become loops over elements of a clump
- Index filters replace many `if` tests (see next section)
- Vectorizes automatically
- Also faster on superscalar architectures
- Fewer lines of code



CLM 3.0

- Derived types with array pointers
 - Pointing into contiguous arrays
- Outer loops over “clumps” of elements
- Scalar blocks become loops over elements of a clump
- Index filters replace many `if` tests
- Vectorizes
 - Requires many `concurrent` directives, thanks to pointers
- Also faster on superscalar architectures



4.3 Filters

- `if` statements reduce parallelism
 - Masks vector operations → redundant ops
- Implement index filter instead



4.3 Filters

```
!dir$ permutation(filterp)
  fn =0
  do pi = plb, pub
    if (<test>) then
      fn = fn+1
      filterp(fn) = pi
    end if
  end do
do fi =1, fn
  pi = filterp(fi)
  oi = pcolumn(pi)
  gi = pgridcell(pi)
  ...
```



4.3 Filters

- Portability comments:
 - It's personal whether filters are harder to read than the original loop with `if`-test code
 - Potentially increases memory usage, but not much
 - No machine-specific code
- Performance:
 - Much better on vector
 - Often better on superscalar



4.4 False dependency

- Code can inhibit parallelism (serializes execution) though iterations are completely independent
- Example: temporary arrays
- Note: here we are not talking about cache-related false dependency



4.4 False dependency

```
common /something/ atemp(n)
```

```
do j = 1,m
```

```
  do i = 1, n
```

```
    atemp( i ) = sqrt( b(i,j) )
```

```
    c(i,j) = c(i,j) + atemp(i)
```

```
  enddo; enddo
```

- Outer loop does not parallelize due to false dependency on atemp

```
real stemp
```

```
do j = 1,m
```

```
  do i = 1, n
```

```
    stemp = sqrt( b(i,j) )
```

```
    c(i,j) = c(i,j) + stemp
```

```
  enddo; enddo
```

- Outer loop parallelizes; More efficient
- May manually fuse loops to remove temporary arrays



VMEC2000 example

```
CALL FUNCT3D(istat)
xstore = xc
N2D: DO n_2d = 0, ntor
  M2D: DO m_2d = 0, mpol1
    DO i = 1, nsize
      js = radial_pts(i)
      xc(js,n_2d,m_2d) = xstore(js,n_2d,m_2d) + hj
      xcdot(js,n_2d,m_2d) = hj
    ENDDO
  CALL FUNCT3D(istat)      ! xc is input, gc is output
  xc = xstore
  xcdot = 0
  ! gc is used to update other arrays not shown
ENDDO
ENDDO
```



VMEC2000 example

- Outer loops are independent
- Can any compiler parallelize this?
- Must be rewritten to parallelize



4.5 Vector replication

- Replicate an array to vectorize multiple updates to the same elements
- Similar trick at a smaller scale for OpenMP by privatizing the array
- Notice `#ifdef _UNICOSMP`



GTC vector replication

```
15.         #ifdef _UNICOSMP
16.             integer, parameter :: vlen = 256
17.             integer :: mv, v
18.             real(wp) vdensityi(mgrid,0:mzeta,vlen)
19.         #endif
20.             real(wp) dnitmp(0:mzeta,mgrid)
21.
32.  r V M-----<><><>  densityi=0.0
81.         #ifdef _OPENMP
91.             !$omp parallel private(dnitmp)
93.                 dnitmp=0.    ! Set array to zero
94.         #elif defined _UNICOSMP
95.  r V M-----<><><>  vdensityi=0.
96.         #endif
```



GTC vector replication

```
122.  m MVs 3      #ifdef _OPENMP
123.  m MVs 3      ! Use thread-private temp array dnitmp to store the
    results
124.  m MVs 3      ij=jtion0(larmor,m)
125.  m MVs 3      dnitmp(kk,ij) = dnitmp(kk,ij) + wz0*wt00
126.  m MVs 3      dnitmp(kk+1,ij)= dnitmp(kk+1,ij) + wz1*wt00
128.  m MVs 3      ij=ij+1
129.  m MVs 3      dnitmp(kk,ij) = dnitmp(kk,ij) + wz0*wt10
130.  m MVs 3      dnitmp(kk+1,ij)= dnitmp(kk+1,ij) + wz1*wt10
132.  m MVs 3      ij=jtion1(larmor,m)
133.  m MVs 3      dnitmp(kk,ij) = dnitmp(kk,ij) + wz0*wt01
134.  m MVs 3      dnitmp(kk+1,ij)= dnitmp(kk+1,ij) + wz1*wt01
136.  m MVs 3      ij=ij+1
137.  m MVs 3      dnitmp(kk,ij) = dnitmp(kk,ij) + wz0*wt11
138.  m MVs 3      dnitmp(kk+1,ij)= dnitmp(kk+1,ij) + wz1*wt11
```



GTC vector replication

```
139.  m MVs 3  #elif defined _UNICOSMP
140.  m MVs 3      ij=jtion0(larmor,m)
141.  m MVs 3      vdensityi(ij,kk,v) = vdensityi(ij,kk,v) + wz0*wt00
142.  m MVs 3      vdensityi(ij,kk+1,v)= vdensityi(ij,kk+1,v) +
    wz1*wt00
144.  m MVs 3      ij=ij+1
145.  m MVs 3      vdensityi(ij,kk,v) = vdensityi(ij,kk,v) + wz0*wt10
146.  m MVs 3      vdensityi(ij,kk+1,v)= vdensityi(ij,kk+1,v) +
    wz1*wt10
148.  m MVs 3      ij=jtion1(larmor,m)
149.  m MVs 3      vdensityi(ij,kk,v) = vdensityi(ij,kk,v) + wz0*wt01
150.  m MVs 3      vdensityi(ij,kk+1,v)= vdensityi(ij,kk+1,v) +
    wz1*wt01
152.  m MVs 3      ij=ij+1
153.  m MVs 3      vdensityi(ij,kk,v) = vdensityi(ij,kk,v) + wz0*wt11
154.  m MVs 3      vdensityi(ij,kk+1,v)= vdensityi(ij,kk+1,v) +
    wz1*wt11
173.  m MVs 3  #endif
```



GTC vector replication

```
181.         #ifdef _OPENMP
182.         ! accumulate results from each thread-private
183.         ! array dnitmp() into the shared array densityi
185.         !$omp critical
186.             do ij=1,mgrid
187.                 do kk=0,mzeta
188.                     densityi(kk,ij)=densityi(kk,ij)+dnitmp(kk,ij)
189.                 enddo
190.             enddo
191.         !$omp end critical
193.         #elif defined _UNICOSMP
194.   ir-----<   do v=1,vlen
195.   ir 2-----<       do kk=0,mzeta
196.   ir 2           !dir$ preferstream
197.   ir 2 MV--<       do ij=1,mgrid
198.   ir 2 MV           densityi(kk,ij) = densityi(kk,ij) +
vdensityi(ij,kk,v)
199.   ir 2 MV-->       enddo
200.   ir 2----->       enddo
201.   ir----->   enddo
202.         #endif
```



GTC vector replication

- Portability comments:
 - Increases memory usage
 - No harder to read/understand than OpenMP section
 - Overall code is getting ugly
 - OpenMP, UNICOS/mp, and serial
 - What could be done better?
 - Could macro names be better?
- Performance gain:
 - Huge on vector machines
 - SMP gains for OpenMP



Agenda

1. Introduction
2. Programming basics
3. General optimizations
4. Advanced optimizations
5. **Case studies**



5.0 Case studies

- The following are “case studies” of some DOE codes
- “Case study” does not necessarily mean a short highly energized study of a code
 - Some will be summaries of the evolution of codes over a several year timeframe



CLM 3.0



The National Center for
Computational Sciences

Portable Performance-Oriented Programming 104

OAK RIDGE NATIONAL LABORATORY
U. S. DEPARTMENT OF ENERGY

Gyro

- Gyro is a [fusion] microturbulence code, [Candy]
 - Continuum (Eulerian)
 - 5-D
 - Runs on a variety of machines: IBM Power4, Cray X1E and XT3, SGI Altix, Opteron clusters
- Summary covers revisions of code from early 2.x versions to 4.
 - Some revisions were direct result of optimizations discussed earlier
 - Some portability techniques also evident in Gyro



Positive code features of Gyro

- DEBUG and VERBOSE input flags
- Checkpointing
 - Current and previous checkpoints
- Prints out norms of arrays
- No derived types or pointers
 - Just arrays
- Uses modules to pass arguments
 - Easy promotion/demotion of arrays
- Consistent programming style
 - Consistent naming scheme of vars and files
- Comment-based data structures
- Simple but effective make system
 - Some support Python scripts
 - No preprocessing (multiple sources controlled by make)



Gyro optimizations

- Directives
- Checkpoints were originally formatted, now unformatted
- Pushed loops down
- Fused loops/reduced temporary memory usage
 - 25% gain in nonlinear-advance FFT routine
- Vectorized across tridiagonal solves
 - With reworking data structures and reworking setup loops, big win on X1E



Gyro optimizations

- Swapped indices
 - 10% gain on X1E, slower on Opterons
- Fix for $\text{sqrt}(1-x)$ where $x \sim 1$
- Pseudo-poly-algorithmic
 - Different sources for a few (core) computationally intense routines (nonlinear advance +/- FFTs)
 - Controlled by make system
- New parallel “distribution” algorithm
 - Big win on all machines



S3D

- Combustion code, PI: Jackie Chen
- Direct numerical simulation of 3D turbulent non-premixed flames
- Runs on variety of machines including IBM SP, Cray X1E, Cray XT3, Opteron cluster, SGI Altix



Positive code features of S3D

- Checkpoints at regular intervals
 - Useful for postprocessing/movies
 - Eats up disk space
- Consistent programming style
- Uses modules to pass arguments
 - Easy promotion/demotion of arrays
- Simple and effective make system
- Sparingly uses (descriptive) `#ifdef` macros
 - Some for machine specific opts: `VECTORVERSION`
 - Some for alternate method: `SAVEFILESINSEPDIR`



S3D optimizations

- After already ported and somewhat optimized by user
- Push 2 loops of triple nest down
 - ~2x speedup (for that version) on X1E
- Add directives
- Removal of MPI Derived Types
 - ~2x speedup (for that version) on X1E, significant gain on other machines
 - Co-Array Fortran initially a workaround
- Overall ~3x speedup on X1E



GTC

- Fusion microturbulence code
- Particle-in-cell (PIC)
- Optimizations/modifications
 - Saw vector replication earlier
 - Used filter to fix “less efficient” compiler vectorization (following)



GTC: Fixing “less efficient”

- A Cray X1[E]-ism
 - Can easily be missed, shows up in messages at the bottom of “.lst” file
- A vectorized loop contains potential conflicts due to indirect addressing at line 266, causing less efficient code to be generated.
- Moral: always check compiler messages



GTC: fixing less efficient

- Before:

```
264. MV-----<      do m=1,mi
265. MV                ip=max(1,min(mflux,1+int((wpi(1,m)-a0)*d_inv)))
266. MV                dtem(ip)=dtem(ip)+wpi(2,m)*zion(5,m)
267. MV                dden(ip)=dden(ip)+1.0
268. MV----->      enddo
```

...

ftn-6371 ftn: VECTOR File = pushi.F90, Line = 264

A vectorized loop contains potential conflicts due to indirect addressing at line 266, causing less efficient code to be generated.

ftn-6371 ftn: VECTOR File = pushi.F90, Line = 264

A vectorized loop contains potential conflicts due to indirect addressing at line 267, causing less efficient code to be generated.

ftn-6204 ftn: VECTOR File = pushi.F90, Line = 264

A loop starting at line 264 was vectorized.

ftn-6601 ftn: STREAM File = pushi.F90, Line = 264

A loop starting at line 264 was multi-streamed.



GTC: fixing less efficient

- After:

```
265. Vw V M-----<><><>      vdtem=0
266. f-----<>                vdden=0
267. m-----<                do mv=1,mi,vlen
268. m MVs-----<            do m=mv,min(mv+vlen-1,mi)
269. m MVs                    v=m-mv+1
270. m MVs                    ip=max(1,min(mflux,1+int((wpi(1,m)-a0)*d_inv)))
271. m MVs                    vdtem(v,ip)=vdtem(v,ip)+wpi(2,m)*zion(5,m)
272. m MVs                    vdden(v,ip)=vdden(v,ip)+1.0
273. m MVs----->            enddo
274. m----->                enddo
275. M-----<                do i=1,mflux
276. M Vw V 4--<><><>        dtem(i)=sum(vdtem(:,i))
277. M f-----<>            dden(i)=sum(vdden(:,i))
278. M----->                enddo
```



CAM

- Community Atmospheric Model (CAM)
- Developed at NCAR
- Used for weather and climate research
- Atmospheric component of CCSM
 - Must run efficiently on a variety of computers
 - Must port easily
- Results must be invariant wrt number of processors used
 - Must disallow some [compiler] optimizations



CAM

- Compile-time or run-time parameters to optimize performance for a given platform, problem or processor count
 - `pcols` is maximum number of columns assigned to a chunk
 - Large `pcols` gives long inner loops, good for vectorization
 - Small `pcols` effective for caching and pipelining, uses less memory
- Code fragments enabled for certain systems, controlled by `cpp` tokens
 - For example, implementations for vector and nonvector systems
- `cpp` tokens for math library routines with different calling sequences on different systems (primarily FFTs)
- **Many** load-time and run-time options for parallel load-balancing of physics



Don'ts: nameless DOE code

- Important DOE code doing production work
 - Many issues with this code (in our opinion)
 - Shall remain nameless
- Problems include
 - Poor choice of macro names
 - Poor placement of `#ifdefs`
 - Extensive mixing of C and Fortran
 - Improper use of PETSc
 - Programming style not consistent
 - Probably result of many authors over many years
 - Lots of dead code
 - No internal timers, checks
 - Lack of comments
 - No runtime verbosity



Nameless examples (just a few)

1. Used implicit none, but then did the wrong thing

```
subroutine xyz
  implicit none
  integer ierr,MPI_COMM_WORLD
```

2. CPP instead of Fortran include

```
subroutine abc3d(arg, myrank)
#include "mpif.h"
```

3. Short, cryptic variable names

```
DATATYPE2 zz,oz,tz,sz,
& con,don,e,a1,a2,a3,
& a4,a5,a6,b1,b2,b3,b4,b5,b6,
& c1,c2,bill,bob
```



Nameless examples

4. Computed gotos, spaghetti code

```
if(iop(1)-5) 201,200,201
  201 c1=w(1)
        if(iop(2)-5) 203,202,203
203 c2=w(k4)
        goto 205
200 if (n-4)300,302,302
302 a1=x(1)-x(2)
C   ... Work
        goto 201
202 if (n-4)300,303,303
303 b1=x(n)-x(n-3)
C   ... More work
        goto 203
```



Nameless examples

5. Potential MPI deadlock

```
SUB1
    all procs call MPI_SEND
SUB2
    all procs call corresponding MPI_RECV
MAIN
    call SUB1
    call SUB2
```

6. Saved variable *lmax typo?*

```
integer lmax
    save lmax
if(ncy.eq.0) lmax=lfu
write(*,*) lmax,u(lmax)
```



Acknowledgments

- Nathan Wichmann, Cray
- Jeff Candy, General Atomics



References/Resources

- Alpern and Carter, “Performance Programming,” www.research.ibm.com/perfprog
- Candy, http://web.gat.com/comp/parallel/physics_results.html
- Constantine, “Back to the Future,” *Comm. ACM*, pg. 126-129, v. 44, March 2001.
- Dijkstra, “Notes on Structured Programming,” Technological University Eindhoven, T.H. Report 70-WSK-03, April 1970.
- Dowd, “*High Performance Computing*,” O’Reilly
- Goedecker and Hoisie, “*Performance Optimization of Numerically Intensive Codes*,” SIAM
- Hatton, “Does OO sync with how we think?” *IEEE Software*, pg. 46-54, May/Jun 1998.
- Hoare, “The Emperor’s Old Clothes,” *Comm. ACM*, pg. 75-83, v. 24, 1981.



References

- Kupferschmid, “*Classical FORTRAN*,” Marcel Dekker, 2002.
- Ledgard, “The Emperor with No Clothes” *Comm. ACM*, pg 126-128, v. 44, Oct. 2001.
- “High performance computing: problem solving with parallel and vector architectures,” Ed. by Sabot
- Stroustrup, “*The C++ Programming Language*”, special Ed., Addison Wesley, 2004
- Willard, “Technology Update: High-Performance Fortran,” *Workstation and High-Performance Systems Bulletin*, IDC #12526, Vol. 2, Tab:6, Nov. 1996.

