Intro to the
Intel Compilers

Scalable Computing Support Center
Duke University
http://wiki.duke.edu/display/SCSC
scsc at duke.edu

John Pormann, Ph.D.
jbpm1 at duke.edu

Outline

- Compiler Reports
- Basic optimization flags
- Inter-Procedure Optimization (IPO)
- Profile-Guided Optimization (PGO)
Compiler Reports

- The compiler can produce reports that can help you optimize your code much more quickly and easily

```
icc -opt-report1 -o runme runme.c
ifort -opt-report1 -o runme runme.f

icc -opt-report1 -opt-report-file=opt_rpt.txt -o runme runme.c
```

- Most output goes to STDERR by default, so try:

```
icc -opt-report1 -o runme runme.c |& tee stdout
```

Max is --opt-report3

---

Compiler Report, Example

```<Run.c:19:160;IPO INLINING;main:0>
INLINING REPORT: (main) [1/177=0.6%]
  -> printf(EXTERN)
  -> printf(EXTERN)
  -> DELETED: InitDebug(49) {isz = 0} {size = 4 (2+2)}
  -> clock(EXTERN)
  -> DELETED: WriteDebug(28) {isz = 0} {size = 5 (2+3)}
  -> ExitSimulation(108) {isz = 159} {size = 170 (41+129)}
  -> ExitTimeStepper(11) {isz = 11} {size = 14 (7+7)}
  -> ExitOutput_Text(17) {isz = 32} {size = 35 (10+25)}
  -> DELETED: ExitStimulus(14) {isz = 0} {size = 2 (2+0)}
  -> INLINE: ExitMembrane_DN(403) {isz = 0} {size = 2 (2+0)}
  -> DELETED: ExitDomain(7) {isz = 0} {size = 2 (2+0)}
  -> Output_Text(16) {isz = 721} {size = 734 (250+484)}
  -> Checkpoint(107) {isz = 273} {size = 290 (117+173)}
  -> DELETED: ExitDebug(73) {isz = 0} {size = 2 (2+0)}
  -> INLINE: Update(395) {isz = 617} {size = 633 (170+463)}
  -> DELETED: CWINT_DebugEnter(71) {isz = 0} {size = 6 (2+4)}
  -> DELETED: CWINT_DebugEnter(71) {isz = 0} {size = 6 (2+4)}
  -> INLINE: sprvec(402) {isz = 38} {size = 53 (17+36)}
```
Basic Optimization

- Step 1 is always to compile with `-O` ... usually implements a variety of compiler-detected/compiler-driven optimizations
  - `-O1` - `-O2` - `-O3` (other compilers may have `-O4` - `-O5`)
  - In-lining functions
  - Loop unrolling
  - Instruction re-ordering

- If you’re compiling on the host you want to run on, try adding `-xHost`
  - Kicks in additional CPU-specific optimizations
  - But the code may not be optimal for other CPU-types

Basic Optimization, cont’d

- If you’re compiling for a machine that you don’t have access to, try:
  - `-xSSE`, `-xSSE2`, `-xSSE3`, `-xSSE4.1`, `-xSSE4.2`, `-xAVX`
  - `-axSSE`, `-axSSE2`, `-axSSE3`, `-axSSE4.1`, `-axSSE4.2`, `-axAVX`
    - `-ax` adds a code-path for SSE## and also a generic code-path
    - can add multiple, comma-separated entries
  - This should allow the compiler to be even more aggressive in unrolling loops, reordering instructions, etc.

- Some others to experiment with:
  - `-mtune=pentium4`
  - `-march=pentium4`
Inter-Procedure Optimization

- Normally, when a compiler optimizes a piece of code, it stops when it hits a function or subroutine call
  - In many cases, it is hard for the compiler to make assumptions about program behavior at any higher or lower level
    
    E.g. is the calc_x() function 5 instructions, or 5000?

- With the Intel compilers, you can request that the compiler go back and re-evaluate optimizations that might occur between different subroutines
  - e.g. in-lining small functions, array dimension padding, stack frame alignment, structure/field reordering, passing arguments in registers (vs. stack), prefetch analysis

IPO command-line flags

- The basic command is:

  icc -ipo -o runme runme.c
  ifort -ipo -o runme runme.f

  Note that IPO actually occurs during final linking, but must be specified during compilation:

  icc -ipo -c runme.c
  icc -o runme runme.o

- `-ipo-c` and `-ipo-S` can be used to see intermediate outputs

- `-opt-report1 -opt-report-phase=ipo` for a report on what the compiler did (or didn’t do, and why)
IPO, cont’d

- IPO can require more memory during compilation
  - If you’re running IPO across many files, it is possible that you might see compiler problems
  - One option is to only run IPO across subsets of files

```
  icc -ipo2 -o runme run1.o run2.o
  icc -ipo-separate -o runme run1.o run2.o
```

- Only go this route if you need to – it can be less effective than `-ipo` by itself

Profile-Guided Optimization

- In optimization, the compiler must make some assumptions about how your code might behave
  - E.g. if a ‘for’ loop is very small, it might flatten it
  - If the loop is small, it may unroll it by 8x
  - If the loop is not-too-small, it may unroll it by 4x
  - If the loop is large, it may unroll it by 2x

- If those assumptions are wrong, the “optimizations” may not be effective ... maybe even slow you down

- So why not use a real program trace to tell the compiler what to expect?
  - i.e. run the program once (on a reasonably-sized problem), catch relevant info as it runs, then have the compiler import that info and re-compile

“reasonably-sized” refers more to data-set size, loop size, etc. and NOT to the number of time-steps in an ODE
PGO

- To compile with PGO:
  ```
  icc -prof-gen -o runme runme.c
  ```

- Then run your code
  - With a "reasonably-sized" problem
    ```
    ./runme problem.dat
    ```
  - You should see a file “pgopti.dpi”

- Then RE-compile with PGO again:
  ```
  icc -prof-use -o runme runme.c
  ```

-O, IPO and PGO

- Basic optimizations, IPO, and PGO can all be used together
  - In fact, {-O,IPO}+PGO can provide better assumptions about when the compiler should do function in-lining, etc.
  - During the 2nd compilation, turn both IPO and PGO on:
    ```
    icc -ipo -prof-use -o runme runme.c
    ```
    - Make sure to recompile everything with both IPO and PGO
      - Be careful with Makefiles... they may detect (old) .o files and NOT recompile them (try ‘make clean’ or ‘rm *.o’ first)
Aside: Round-Off Error

- Optimizations (-O, IPO, PGO) should not significantly alter your algorithms
- BUT .. they may alter the low-order bits in your mathematics
  - Especially on x86 hardware, analyzing round-off error is complicated
    - Most math is done in 80-bit precision, and only converted to the (more standard) 64-bit precision when it is written to memory
    - If the compiler re-arranges your math instructions, then more, or less, of the math could be done in 80-bit precision
    - If the compiler moves you to SSE (we’ll discuss later), then you might get all-64-bit math
  - If your research depends on the 20th digit in your results, you (already) need to make sure that you know what you’re doing w.r.t. round-off error!!

Summary

- No changes to code needed!
  - -opt-report1
  - -O3 -mtune=…
  - -ipo
  - -prof-gen … -O3 -ipo -prof-use
- Consider using a makefile to keep all of this straight
  - and use ‘make clean’ or ‘rm *.o’ between changes to flags
Thank You!

- Please check out our other seminars and workshops:
  - Intro to Pthreads/OpenMP (shared memory programming)
  - Intro to MPI (distributed memory programming)
  - Intro to CUDA (GPGPU programming)
  - Parallel Programming Techniques
  - Software engineering tools

- Or keep in touch with us:
  - Wiki: http://wiki.duke.edu/display/SCSC
  - Blog: http://sites.duke.edu/scsc/
  - Email: scsc at duke.edu
Now we have to open up the code

- Basic optimization, PGO, and IPO are automatic or pseudo-automatic mechanisms for speeding up your code

- To go beyond that ... we have to open up the code
  - But you may not have to severely hack your code
  - A few compiler “pragmas” may be all that you need
    - These are compiler-specific instructions or suggestions
    - They should not change the algorithms in the code
      - -O3, PGO, IPO can change the round-off error behavior of your program
        (i.e. the output may be slightly different)
  - And if you give the code to someone else (who doesn’t have the Intel compilers), the pragmas should just be ignored
Loop-Level Optimizations

- In large or complex loops, the compiler may not be able to tell if all the vectors are truly independent
  - Fortran: ‘do’ loops are fairly rigid; arrays are well-defined objects
  - C/C++: ‘for’ loops are very general; arrays are just pointers
    - Two arrays could actually point to the same memory area, in which case some optimizations could produce wrong answers
      
      ```
      for(i=0;i<N;i++) {
          a[i] = b[i+1]*c[i] + d[i-1];
      }
      ```
    - If ‘a’ and ‘b’ are actually the same array, then we cannot calculate a[i] and a[i+1] at the same time (since a[i+1] is the same as b[i+1] which is needed for the a[i] calculation)

Vector Independence (ivdep)

- To identify to the compiler that you are guaranteeing independence of the vectors/arrays:
  ```
  #pragma ivdep
  for(i=0;i<N;i++) {
      a[i] = b[i+1]*c[i] + d[i-1];
  }
  ```
  - This says that ‘a’, ‘b’, ‘c’, and ‘d’ are all independent arrays (non-overlapping memory areas)

  (Aside: for strict ANSI C, the ‘#pragma’ should occur in the first column of the line, no tabs/spaces, don’t match indentation)
ivdep, cont’d

- Compile with:

  ```
  icc -ivdep-parallel -o runme runme.c
  icc -ivdep-parallel -opt-report 3 -o runme runme.c
  ```

  - This tells the compiler to respect ‘#pragma ivdep’

‘restrict’ keyword

- Where ‘ivdep’ has to be declared for every loop, you can also identify arrays (pointers) as “restricted” (non-aliased)
  - Then it applies to all loops over those arrays
  - Often used in function arguments or local variables w/in a function

```c
void mathstuff( double* restrict a, double* restrict b, double* restrict c, double* restrict d ) {
  for(i=0;i<N;i++) {
    a[i] = b[i+1]*c[i] + d[i-1];
  }
  for(i=0;i<N;i++) {
    b[i] = c[i-5]*d[i+4];
  }
}
```

```
icc -restrict -o runme runme.c
```
Other anti-aliasing options

- If you never alias vectors (never have multiple pointers to the same block of memory), then you can try:
  
  icc -fargument-alias -o runme runme.c  
icc -fno-fnalias -o runme runme.c  
icc -fno-alias -o runme runme.c

  - This says that all function arguments are un-aliased
  - No need for ‘#pragma ivdep’ or ‘restrict’ at all
  - Probably should use ‘-opt-report=3’ to make sure you’re getting what you expect

Loop “Size”

- Modern CPUs often have multiple Arithmetic/Logic Units (ALUs)
  
  - Meaning they can perform multiple add/sub/mul/div operations simultaneously … possibly 2-8 ops per clock-tick

  - If your loop is too “small,” it only uses only 1 ALU
    
    - So you might be losing 1/2 to 7/8 of the computational power of the CPU

- Modern CPUs only have a few registers (what the ALUs operate on)
  
  - When you run out of registers, your program has to write something to main memory (slow)

  - If your loop is too “big,” it may run out of registers and not be able to keep the ALUs full
Loop Unrolling

- For all loops, there is some overhead in executing the (machine-language) instructions for the loop itself
  - Incrementing the counter == addition operation
  - Testing the bounds == subtraction operation
    - For large loops, this could be a lot of extra calculations

- If you know a loop is large, you could do “groups” of 4 or 8 calculations before incrementing the counter/testing the bounds
  - 4 or 8x less overhead

- Unrolling also helps make “small” loops look “bigger”

---

Loop Unrolling, Manually

```c
for(i=0;i<N;i++) {
    y[i] = alpha*x[i]+y[i];
}
```

```c
for(i=0;i<N;i+=4) {
    y[i] = alpha*x[i]+y[i];
    y[i+1] = alpha*x[i+1]+y[i+1];
    y[i+2] = alpha*x[i+2]+y[i+2];
    y[i+3] = alpha*x[i+3]+y[i+3];
}
```
Loop Unrolling with Pragmas

```c
#pragma unroll
for(i=0;i<N;i++) {
    y[i] = alpha*x[i]+y[i];
}
```

```c
#pragma unroll(8)
for(i=0;i<N;i++) {
    y[i] = alpha*x[i]+y[i];
}
```

Loop Unrolling, Automated

- The compiler can do this kind of loop unrolling automatically:

  ```
  icc -unroll -o runme runme.c
  icc -funroll-all-loops -o runme runme.c
  ```

- Unrolling small loops may not be efficient, so the compiler sometimes will skip them ... use the ‘-funroll-all-loops’ to force it to unroll everything
  - But use with caution
  - Or try to restrict it to single-file compilations
Loop Fusion/Loop Jamming

- Another approach for “small” loops is to fuse them with another loop (or to “jam” the two loops together)
  - Instead of:
    
    ```
    for(i=0;i<N;i++) {
        a[i] = b[i-1] + c[i];
    }
    for(i=0;i<N;i++) {
        x[i] = b[i+1] - d[i];
    }
    ```
  - We do:
    
    ```
    for(i=0;i<N;i++) {
        a[i] = b[i-1] + c[i];
        x[i] = b[i+1] - d[i];
    }
    ```
  - In this case, we may get better cache re-use for the ‘b’ array as well

Unroll and Jam

- Loop unrolling can also help inflate “small” loops to “big” loops
- We can do both: unroll the loop and jam it together with another loop:
  
  ```
  #pragma unroll_and_jam (6)
  for(i=1;i<N;i++) {
      #pragma unroll_and_jam (6)
      for(j=1;j<N;j++) {
          for (k = 1; k < n; k++){
              a[i][j] += b[i][k]*c[k][j];
          }
      }
  }
  ```
- Need to compile with ‘-O3’
Loop Distribution

- For “big” loops, you can sometimes “distribute” the loop into several smaller loops:

```c
for(i=0;i<N;i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
}
```

```c
#pragma distribute_point
for(i=0;i<N;i++) {
    a[i] = a[i] + i;
    b[i] = b[i] + i;
    c[i] = c[i] + i;
    x[i] = x[i] + i;
    y[i] = y[i] + i;
    z[i] = z[i] + i;
}
```

Vectorization/SSE Instruction Use

- For small/compact/math intensive loops, you may want to use the Intel SSE unit (kind of an extra-large ALU)
  - An SSE-add operation is actually 2-4 simultaneous adds
    - So you could get 2x to 4x speed-up!
  - The loop must be “unroll-able” ... there must be more than one thing that can be computed in parallel
    ```c
    #pragma vector
    for(i=0;i<N;i++) {
        y[i] = alpha*x[i] + y[i];
    }
    ```
  - Generally speaking, this is just a hint to the compiler ... to “force” the compiler, try:
    ```c
    #pragma vector always
    ```
Prefetching

- For vector-like operations, we know that we’re going to run through a whole array, in order: a[0], a[1], a[2], a[3], a[4], a[5], …
  - We can ask the compiler to “pre-fetch” memory locations to help reduce delays caused by main memory speed (vs. cache speed)

- Compiler-driven prefetching:
  ```
  icc -opt-prefetch -o runme runme.c
  ```

- Programmer-driven prefetching:
  ```
  #pragma prefetch x,y
  for(i=0;i<N;i++) {
    y[i] = alpha*x[i] + y[i];
  }
  ```

Summary

- We’ve had to open up the source code
  - But have NOT made any changes to the algorithms (or to the science)
  - Think of pragmas as suggestions to the compiler, not “real” code

- Array independence / anti-aliasing

- Loop unrolling
- Loop jamming
- Loop distribution

- Vectorization (SSE)
- Prefetching
Thank You!

- Please check out our other seminars and workshops:
  - Intro to Pthreads/OpenMP (shared memory programming)
  - Intro to MPI (distributed memory programming)
  - Intro to CUDA (GPGPU programming)
  - Parallel Programming Techniques
  - Software engineering tools

- Or keep in touch with us:
  - Wiki: http://wiki.duke.edu/display/SCSC
  - Blog: http://sites.duke.edu/scsc/
  - Email: scsc at duke.edu