Parallel Programming & Cluster Computing
Stupid Compiler Tricks
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College
Tuesday October 11 2011
Outline

- Dependency Analysis
  - What is Dependency Analysis?
  - Control Dependencies
  - Data Dependencies
- Stupid Compiler Tricks
  - Tricks the Compiler Plays
  - Tricks You Play With the Compiler
  - Profiling
Dependency Analysis
What Is Dependency Analysis?

Dependency analysis describes how different parts of a program affect one another, and how various parts require other parts in order to operate correctly.

A control dependency governs how different sequences of instructions affect each other.

A data dependency governs how different pieces of data affect each other.

Much of this discussion is from references [1] and [6].
Control Dependencies

Every program has a well-defined \textit{flow of control} that moves from instruction to instruction to instruction.

This flow can be affected by several kinds of operations:

- Loops
- Branches (if, select case/switch)
- Function/subroutine calls
- I/O (typically implemented as calls)

Dependencies affect \textit{parallelization}!
Branch Dependency (F90)

\[ y = 7 \]

\[
\text{IF } (x \neq 0) \text{ THEN}
\]

\[ y = 1.0 / x \]

\[
\text{END IF}
\]

Note that \((x \neq 0)\) means “\(x\) not equal to zero.”

The value of \(y\) depends on what the condition \((x \neq 0)\) evaluates to:

- If the condition \((x \neq 0)\) evaluates to .TRUE., then \(y\) is set to \(1.0 / x\). (1 divided by \(x\)).
- Otherwise, \(y\) remains 7.
Branch Dependency (C)

```c
y = 7;
if (x != 0) {
    y = 1.0 / x;
}
```

Note that `(x != 0)` means “`x` not equal to zero.”

The value of `y` depends on what the condition `(x != 0)` evaluates to:

- If the condition `(x != 0)` evaluates to `true`, then `y` is set to `1.0 / x` (1 divided by `x`).
- Otherwise, `y` remains `7`. 

DO i = 2, length
    a(i) = a(i-1) + b(i)
END DO

Here, each iteration of the loop depends on the previous:
iteration $i=3$ depends on iteration $i=2$,
iteration $i=4$ depends on iteration $i=3$,
iteration $i=5$ depends on iteration $i=4$, etc.

This is sometimes called a \textit{loop carried dependency}.

There is no way to execute iteration $i$ until after iteration $i-1$ has completed, so this loop can’t be parallelized.
Loop Carried Dependency (C)

```c
for (i = 1; i < length; i++) {
    a[i] = a[i-1] + b[i];
}
```

Here, each iteration of the loop depends on the previous:
- iteration `i=3` depends on iteration `i=2`,
- iteration `i=4` depends on iteration `i=3`,
- iteration `i=5` depends on iteration `i=4`, etc.

This is sometimes called a *loop carried dependency*. There is no way to execute iteration `i` until after iteration `i-1` has completed, so this loop can’t be parallelized.
**Why Do We Care?**

**Loops** are the favorite control structures of High Performance Computing, because compilers know how to **optimize** their performance using instruction-level parallelism: superscalar, pipelining and vectorization can give excellent speedup.

**Loop carried dependencies** affect whether a loop can be parallelized, and how much.
Loop or Branch Dependency? (F)

Is this a **loop carried dependency** or a **branch dependency**?

```fortran
DO i = 1, length
  IF (x(i) /= 0) THEN
    y(i) = 1.0 / x(i)
  END IF
END DO
```
Loop or Branch Dependency? (C)

Is this a **loop carried dependency** or a **branch dependency**?

```c
for (i = 0; i < length; i++) {
    if (x[i] != 0) {
        y[i] = 1.0 / x[i];
    }
}
```
Call Dependency Example (F90)

\[ x = 5 \]
\[ y = \text{myfunction}(7) \]
\[ z = 22 \]

The flow of the program is interrupted by the call to \textit{myfunction}, which takes the execution to somewhere else in the program.

It’s similar to a branch dependency.
Call Dependency Example (C)

\[
x = 5;
y = \text{myfunction}(7);
z = 22;
\]

The flow of the program is interrupted by the \textit{call} to \texttt{myfunction}, which takes the execution to somewhere else in the program.

It’s similar to a branch dependency.
I/O Dependency (F90)

\[ x = a + b \]
\[ \text{PRINT } *, x \]
\[ y = c + d \]

Typically, I/O is implemented by hidden subroutine calls, so we can think of this as equivalent to a call dependency.
I/O Dependency (C)

\[ x = a + b; \]
\[ \text{printf}("%f", x); \]
\[ y = c + d; \]

Typically, I/O is implemented by hidden subroutine calls, so we can think of this as equivalent to a call dependency.
Reductions Aren’t Dependencies

\begin{verbatim}
array_sum = 0
DO i = 1, length
   array_sum = array_sum + array(i)
END DO
\end{verbatim}

A \textit{reduction} is an operation that converts an array to a scalar. Other kinds of reductions: product, \texttt{AND}, \texttt{OR}, minimum, maximum, index of minimum, index of maximum, number of occurrences of a particular value, etc.

Reductions are so common that hardware and compilers are optimized to handle them.

Also, they aren’t really dependencies, because the order in which the individual operations are performed doesn’t matter.
Reductions Aren’t Dependencies

```c
array_sum = 0;
for (i = 0; i < length; i++) {
    array_sum = array_sum + array[i];
}
```

A **reduction** is an operation that converts an array to a scalar. Other kinds of reductions: product, &&, ||, minimum, maximum, index of minimum, index of maximum, number of occurrences of a particular value, etc.

Reductions are so common that hardware and compilers are optimized to handle them.

Also, they aren’t really dependencies, because the order in which the individual operations are performed doesn’t matter.
Data Dependencies (F90)

“A data dependence occurs when an instruction is dependent on data from a previous instruction and therefore cannot be moved before the earlier instruction [or executed in parallel].” [7]

\[
\begin{align*}
    a &= x + y + \cos(z) \\
    b &= a \times c
\end{align*}
\]

The value of \(b\) depends on the value of \(a\), so these two statements must be executed in order.
Data Dependencies (C)

“A data dependence occurs when an instruction is dependent on data from a previous instruction and therefore cannot be moved before the earlier instruction [or executed in parallel].” [7]

\[ a = x + y + \cos(z) ; \]
\[ b = a * c ; \]

The value of \( b \) depends on the value of \( a \), so these two statements **must** be executed in order.
Output Dependencies (F90)

\[
x = a / b \\
y = x + 2 \\
x = d - e
\]

Notice that \( x \) is assigned \textbf{two different values}, but only one of them is retained after these statements are done executing. In this context, the final value of \( x \) is the “output.”

Again, we are forced to execute in order.
Output Dependencies (C)

\[
x = a / b;
\]
\[
y = x + 2;
\]
\[
x = d - e;
\]

Notice that \textbf{x} is assigned \textbf{two different values}, but only one of them is retained after these statements are done executing. In this context, the final value of \textbf{x} is the “output.”

Again, we are forced to execute in order.
Why Does Order Matter?

- Dependencies can affect whether we can execute a particular part of the program in parallel.
- If we cannot execute that part of the program in parallel, then it’ll be **SLOW**.
Loop Dependency Example

if ((dst == src1) && (dst == src2)) {
    for (index = 1; index < length; index++) {
        dst[index] = dst[index-1] + dst[index];
    }
} else if (dst == src1) {
    for (index = 1; index < length; index++) {
        dst[index] = dst[index-1] + src2[index];
    }
} else if (dst == src2) {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + dst[index];
    }
} else if (src1 == src2) {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + src1[index];
    }
} else {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + src2[index];
    }
}
Loop Dep Example (cont’d)

if ((dst == src1) && (dst == src2)) {
    for (index = 1; index < length; index++) {
        dst[index] = dst[index-1] + dst[index];
    }
} else if (dst == src1) {
    for (index = 1; index < length; index++) {
        dst[index] = dst[index-1] + src2[index];
    }
} else if (dst == src2) {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + dst[index];
    }
} else if (src1 == src2) {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + src1[index];
    }
} else {
    for (index = 1; index < length; index++) {
        dst[index] = src1[index-1] + src2[index];
    }
}

The various versions of the loop either:
- **do have loop carried dependencies**, or
- **don’t have loop carried dependencies**.
Loop Dependency Performance

Loop Carried Dependency Performance

Better

MFLOPs

dst=src1+src2

dst=src1+src1

dst=dst+src2

dst=src1+dst

dst=dst+dst

Pentium3 500 MHz

POWER4

Pentium4 2GHz

EM64T 3.2 GHz

Parallel Programming: Compilers
OK Supercomputing Symposium, Tue Oct 11 2011
Stupid Compiler Tricks
Stupid Compiler Tricks

- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- Tricks You Can Play with Compilers
  - Profiling
  - Hardware counters
Compiler Design

The people who design compilers have a lot of experience working with the languages commonly used in High Performance Computing:

- Fortran: 50ish years
- C: 40ish years
- C++: 25ish years, plus C experience

So, they’ve come up with clever ways to make programs run faster.
Tricks Compilers Play
Scalar Optimizations

- Copy Propagation
- Constant Folding
- Dead Code Removal
- Strength Reduction
- Common Subexpression Elimination
- Variable Renaming
- Loop Optimizations

Not every compiler does all of these, so it sometimes can be worth doing these by hand.

Much of this discussion is from [2] and [6].
Copy Propagation (F90)

Before

\[ x = y \]
\[ z = 1 + x \]

Has data dependency

Compile

After

\[ x = y \]
\[ z = 1 + y \]

No data dependency
Copy Propagation (C)

Before
\[
x = y;
\]
\[
z = 1 + x;
\]

Has data dependency

Compile

After
\[
x = y;
\]
\[
z = 1 + y;
\]

No data dependency
Constant Folding (F90)

Before

\[
\begin{align*}
\text{add} &= 100 \\
\text{aug} &= 200 \\
\text{sum} &= \text{add} + \text{aug}
\end{align*}
\]

After

\[
\begin{align*}
\text{sum} &= 300
\end{align*}
\]

Notice that \text{sum} is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.
Constant Folding (C)

**Before**

```c
add = 100;
aug = 200;
sum = add + aug;
```

**After**

```c
sum = 300;
```

Notice that `sum` is actually the sum of two constants, so the compiler can precalculate it, eliminating the addition that otherwise would be performed at runtime.
Dead Code Removal (F90)

Before

\[ \text{var} = 5 \]
\[ \text{PRINT } *, \text{ var} \]
\[ \text{STOP} \]
\[ \text{PRINT } *, \text{ var } \ast 2 \]

After

\[ \text{var} = 5 \]
\[ \text{PRINT } *, \text{ var} \]
\[ \text{STOP} \]

Since the last statement never executes, the compiler can eliminate it.
Dead Code Removal (C)

**Before**

```c
var = 5;
printf("%d", var);
exit(-1);
printf("%d", var * 2);
```

**After**

```c
var = 5;
printf("%d", var);
exit(-1);
```

Since the last statement never executes, the compiler can eliminate it.
Raising one value to the power of another, or dividing, is more expensive than multiplying. If the compiler can tell that the power is a small integer, or that the denominator is a constant, it’ll use multiplication instead.

Note: In Fortran, “y ** 2.0” means “y to the power 2.”
**Strength Reduction (C)**

**Before**

\[
\begin{align*}
x & = \text{pow}(y, 2.0); \\
a & = \frac{c}{2.0};
\end{align*}
\]

**After**

\[
\begin{align*}
x & = y \times y; \\
a & = c \times 0.5;
\end{align*}
\]

Raising one value to the power of another, or dividing, is more expensive than multiplying. If the compiler can tell that the power is a small integer, or that the denominator is a constant, it’ll use multiplication instead.

Note: In C, “\texttt{pow}(y, 2.0)” means “y to the power 2.”
Common Subexpression Elimination (F90)

Before
\[
\begin{align*}
  d &= c \times (a / b) \\
  e &= (a / b) \times 2.0
\end{align*}
\]

After
\[
\begin{align*}
  \text{adivb} &= a / b \\
  d &= c \times \text{adivb} \\
  e &= \text{adivb} \times 2.0
\end{align*}
\]

The subexpression \((a / b)\) occurs in both assignment statements, so there’s no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate.
Common Subexpression Elimination (C)

Before

\[ d = c * (a / b); \]
\[ e = (a / b) * 2.0; \]

After

\[ \text{adivb} = a / b; \]
\[ d = c * \text{adivb}; \]
\[ e = \text{adivb} * 2.0; \]

The subexpression \((a / b)\) occurs in both assignment statements, so there’s no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate.
Variable Renaming (F90)

\[
\begin{align*}
\text{Before} \\
x &= y \times z \\
q &= r + x \times 2 \\
x &= a + b
\end{align*}
\]

\[
\begin{align*}
\text{After} \\
x_0 &= y \times z \\
q &= r + x_0 \times 2 \\
x &= a + b
\end{align*}
\]

The original code has an **output dependency**, while the new code **doesn’t** – but the final value of \(x\) is still correct.
Variable Renaming (C)

**Before**

\[
x = y \times z;
q = r + x \times 2;
x = a + b;
\]

**After**

\[
x0 = y \times z;
q = r + x0 \times 2;
x = a + b;
\]

The original code has an **output dependency**, while the new code **doesn’t** – but the final value of \( x \) is still correct.
Loop Optimizations

- Hoisting Loop Invariant Code
- Unswitching
- Iteration Peeling
- Index Set Splitting
- Loop Interchange
- Unrolling
- Loop Fusion
- Loop Fission

Not every compiler does all of these, so it sometimes can be worth doing some of these by hand.

Much of this discussion is from [3] and [6].
Before

\[ \text{DO } i = 1, n \]
\[ a(i) = b(i) + c \times d \]
\[ e = g(n) \]
\[ \text{END DO} \]

After

\[ \text{temp} = c \times d \]
\[ \text{DO } i = 1, n \]
\[ a(i) = b(i) + \text{temp} \]
\[ \text{END DO} \]
\[ e = g(n) \]

Code that doesn’t change inside the loop is known as loop invariant. It doesn’t need to be calculated over and over.
Code that doesn’t change inside the loop is known as *loop invariant*. It doesn’t need to be calculated over and over.

**Before**

```c
for (i = 0; i < n; i++) {
    a[i] = b[i] + c * d;
    e = g(n);
}
```

**After**

```c
temp = c * d;
for (i = 0; i < n; i++) {
    a[i] = b[i] + temp;
}
e = g(n);
```
Unswitching (F90)

```fortran
DO i = 1, n
  DO j = 2, n
    IF (t(i) > 0) THEN
      a(i,j) = a(i,j) * t(i) + b(j)
    ELSE
      a(i,j) = 0.0
    END IF
  END DO
END DO
```

The condition is j-independent.

Before

So, it can migrate outside the j loop.

After
Unswitching (C)

for (i = 0; i < n; i++) {
    for (j = 1; j < n; j++) {
        if (t[i] > 0)
            a[i][j] = a[i][j] * t[i] + b[j];
        else {
            a[i][j] = 0.0;
        }
    }
}

for (i = 0; i < n; i++) {
    if (t[i] > 0) {
        for (j = 1; j < n; j++) {
            a[i][j] = a[i][j] * t[i] + b[j];
        }
    } else {
        for (j = 1; j < n; j++) {
            a[i][j] = 0.0;
        }
    }
}
Iteration Peeling (F90)

Before

\[
\text{DO } i = 1, n \\
\quad \text{IF } ((i == 1) \text{ OR. } (i == n)) \text{ THEN} \\
\quad \quad x(i) = y(i) \\
\quad \text{ELSE} \\
\quad \quad x(i) = y(i + 1) + y(i - 1) \\
\quad \text{END IF} \\
\text{END DO}
\]

We can eliminate the IF by \textit{peeling} the weird iterations.

\[
\text{After} \\
\quad x(1) = y(1) \\
\text{DO } i = 2, n - 1 \\
\quad x(i) = y(i + 1) + y(i - 1) \\
\text{END DO} \\
\quad x(n) = y(n)
\]
Iteration Peeling (C)

```c
for (i = 0; i < n; i++) {
    if ((i == 0) || (i == (n - 1))) {
        x[i] = y[i];
    } else {
        x[i] = y[i + 1] + y[i - 1];
    }
}
```

We can eliminate the if by **peeling** the weird iterations.

```c
x[0] = y[0];
for (i = 1; i < n - 1; i++) {
    x[i] = y[i + 1] + y[i - 1];
}
x[n-1] = y[n-1];
```
Index Set Splitting (F90)

DO i = 1, n
    a(i) = b(i) + c(i)
    IF (i > 10) THEN
        d(i) = a(i) + b(i - 10)
    END IF
END DO

DO i = 1, 10
    a(i) = b(i) + c(i)
END DO

DO i = 11, n
    a(i) = b(i) + c(i)
    d(i) = a(i) + b(i - 10)
END DO

Note that this is a generalization of peeling.
Index Set Splitting (C)

```c
for (i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
    if (i >= 10) {
        d[i] = a[i] + b[i - 10];
    }
}

for (i = 0; i < 10; i++) {
    a[i] = b[i] + c[i];
}
for (i = 10; i < n; i++) {
    a[i] = b[i] + c[i];
    d[i] = a[i] + b[i - 10];
}
```

Before

After

Note that this is a generalization of peeling.
Loop Interchange (F90)

**Before**

```fortran
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

**After**

```fortran
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements \(a(i,j)\) and \(a(i+1,j)\) are near each other in memory, while \(a(i,j+1)\) may be far, so it makes sense to make the \(i\) loop be the inner loop. (This is reversed in C, C++ and Java.)
Loop Interchange (C)

Before

```c
for (j = 0; j < nj; j++) {
    for (i = 0; i < ni; i++) {
        a[i][j] = b[i][j];
    }
}
```

Array elements `a[i][j]` and `a[i][j+1]` are near each other in memory, while `a[i+1][j]` may be far, so it makes sense to make the `j` loop be the inner loop. (This is reversed in Fortran.)

After

```c
for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
        a[i][j] = b[i][j];
    }
}
```
Unrolling (F90)

DO i = 1, n

Before  a(i) = a(i) + b(i)

END DO

DO i = 1, n, 4

After

a(i) = a(i) + b(i)

a(i+1) = a(i+1) + b(i+1)

a(i+2) = a(i+2) + b(i+2)

a(i+3) = a(i+3) + b(i+3)

END DO

You generally shouldn’t unroll by hand.
Unrolling (C)

for (i = 0; i < n; i++) {

  Before a[i] = a[i] + b[i];

}

for (i = 0; i < n; i += 4) {

  After a[i] = a[i] + b[i];
  a[i+1] = a[i+1] + b[i+1];
  a[i+2] = a[i+2] + b[i+2];
  a[i+3] = a[i+3] + b[i+3];

}

You generally shouldn’t unroll by hand.
Why Do Compilers Unroll?

We saw last time that a loop with a lot of operations gets better performance (up to some point), especially if there are lots of arithmetic operations but few main memory loads and stores.

Unrolling creates multiple operations that typically load from the same, or adjacent, cache lines.

So, an unrolled loop has more operations without increasing the memory accesses by much.

Also, unrolling decreases the number of comparisons on the loop counter variable, and the number of branches to the top of the loop.
Loop Fusion (F90)

```
DO i = 1, n
  a(i) = b(i) + 1
END DO
DO i = 1, n
  c(i) = a(i) / 2
END DO
DO i = 1, n
  d(i) = 1 / c(i)
END DO
```

Before

```
DO i = 1, n
  a(i) = b(i) + 1
  c(i) = a(i) / 2
  d(i) = 1 / c(i)
END DO
```

After

As with unrolling, this has fewer branches. It also has fewer total memory references.
Loop Fusion (C)

```c
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
}
for (i = 0; i < n; i++) {
    c[i] = a[i] / 2;
}
for (i = 0; i < n; i++) {
    d[i] = 1 / c[i];
}

for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
    d[i] = 1 / c[i];
}
```

As with unrolling, this has fewer branches. It also has fewer total memory references.

Loop Fission (F90)

Before

DO i = 1, n
   a(i) = b(i) + 1
   c(i) = a(i) / 2
   d(i) = 1 / c(i)
END DO

After

DO i = 1, n
   a(i) = b(i) + 1
END DO

DO i = 1, n
   c(i) = a(i) / 2
END DO

DO i = 1, n
   d(i) = 1 / c(i)
END DO

Fission reduces the cache footprint and the number of operations per iteration.
Loop Fission (C)

```c
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] / 2;
    d[i] = 1 / c[i];
}
```

Before

```c
for (i = 0; i < n; i++) {
    a[i] = b[i] + 1;
}
```

```c
for (i = 0; i < n; i++) {
    c[i] = a[i] / 2;
}
```

```c
for (i = 0; i < n; i++) {
    d[i] = 1 / c[i];
}
```

After

Fission reduces the cache footprint and the number of operations per iteration.
To Fuse or to Fizz?

The question of when to perform fusion versus when to perform fission, like many many optimization questions, is highly dependent on the application, the platform and a lot of other issues that get very, very complicated.

Compilers don’t always make the right choices.

That’s why it’s important to examine the actual behavior of the executable.
Inlining (F90)

**Before**

```
DO i = 1, n
  a(i) = func(i)
END DO
...
REAL FUNCTION func (x)
  ...
  func = x * 3
END FUNCTION func
```

**After**

```
DO i = 1, n
  a(i) = i * 3
END DO
```

When a function or subroutine is *inlined*, its contents are transferred directly into the calling routine, eliminating the overhead of making the call.
Inlining (C)

**Before**

```c
for (i = 0; i < n; i++) {
    a[i] = func(i+1);
}
...
float func(x) {
    return x * 3;
}
```

**After**

```c
for (i = 0; i < n; i++) {
    a[i] = (i+1) * 3;
}
```

When a function or subroutine is **inlined**, its contents are transferred directly into the calling routine, eliminating the overhead of making the call.
Tricks You Can Play with Compilers
The Joy of Compiler Options

Every compiler has a different set of options that you can set. Among these are options that control single processor optimization: superscalar, pipelining, vectorization, scalar optimizations, loop optimizations, inlining and so on.
Example Compile Lines

- IBM XL
  ```
  xlf90 -O -qmaxmem=-1 -qarch=auto
  -qtune=auto -qcache=auto -qhot
  ```

- Intel
  ```
  ifort -O -march=core2 -mtune=core2
  ```

- Portland Group f90
  ```
  pgf90 -O3 -fastsse -tp core2-64
  ```

- NAG f95
  ```
  f95 -O4 -Ounsafe -ieee=nonstd
  ```
What Does the Compiler Do? #1

Example: NAG Fortran 77/90/95 compiler

\texttt{nagfor -O<level> source.f90}

Possible levels are \texttt{-O0, -O1, -O2, -O3, -O4}:

\begin{itemize}
  \item \texttt{-O0} No optimisation. ...
  \item \texttt{-O1} Minimal quick optimisation.
  \item \texttt{-O2} Normal optimisation.
  \item \texttt{-O3} Further optimisation.
  \item \texttt{-O4} Maximal optimisation.
\end{itemize}

The manual page is pretty cryptic.
Example: Intel ifort compiler \cite{5}

\texttt{ifort -O<level> source.f90}

Possible levels are \texttt{-O0}, \texttt{-O1}, \texttt{-O2}, \texttt{-O3}:

\begin{itemize}
  \item \texttt{-O0} Disables all \texttt{-O<n>} optimizations. ...
  \item \texttt{-O1} \ldots \texttt{[E]nables optimizations for speed. \ldots}
  \item \texttt{-O2} \ldots
    \begin{itemize}
      \item Inlining of intrinsics.
      \item Intra-file interprocedural optimizations, which include: inlining, constant propagation, forward substitution, routine attribute propagation, variable address-taken analysis, dead static function elimination, and removal of unreferenced variables.
    \end{itemize}
  \item \texttt{-O3} Enables \texttt{-O2} optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop transformations. Enables optimizations for maximum speed, but does not guarantee higher performance unless loop and memory access transformations take place. \ldots
\end{itemize}
Arithmetic Operation Speeds

Ordered Arithmetic Operations

Better
More Optimized Performance

Better

Performance

Operation

MFLOP/s

0
50
100
150
200
250

rmam
inam
rmad
inad
rdot
reuc
rlot8
rlot10
rlot12
rlot16
rlot20
rlot24

Pentium3 NAG O0
Pentium3 NAG 04
Pentium3 VAST no opt
Pentium3 VAST opt

Parallel Programming: Compilers
OK Supercomputing Symposium, Tue Oct 11 2011
Profiling
Profiling means collecting data about how a program executes. The two major kinds of profiling are:

- Subroutine profiling
- Hardware timing
Subroutine Profiling

*Subroutine profiling* means finding out how much time is spent in each routine.

**The 90-10 Rule:** Typically, a program spends 90% of its runtime in 10% of the code.

Subroutine profiling tells you what parts of the program to spend time optimizing and what parts you can ignore.

Specifically, at regular intervals (e.g., every millisecond), the program takes note of what instruction it’s currently on.
Profiling Example

On GNU compilers systems:

```
gcc -O -g -pg ...
```

The `-g` `-pg` options tell the compiler to set the executable up to collect profiling information.

Running the executable generates a file named `gmon.out`, which contains the profiling information.
Profiling Example (cont’d)

When the run has completed, a file named `gmon.out` has been generated.

Then:

```bash
 gprof executable
```

produces a list of all of the routines and how much time was spent in each.
## Profiling Result

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>total calls</th>
<th>ms/call self</th>
<th>ms/call total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.6</td>
<td>52.72</td>
<td>52.72</td>
<td>480000</td>
<td>0.11</td>
<td>0.11</td>
<td>longwave_ [5]</td>
<td></td>
</tr>
<tr>
<td>24.3</td>
<td>99.06</td>
<td>46.35</td>
<td>897</td>
<td>51.67</td>
<td>51.67</td>
<td>mpdata3_ [8]</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>114.19</td>
<td>15.13</td>
<td>300</td>
<td>50.43</td>
<td>50.43</td>
<td>turb_ [9]</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>127.94</td>
<td>13.75</td>
<td>299</td>
<td>45.98</td>
<td>45.98</td>
<td>turb_scalar_ [10]</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>136.91</td>
<td>8.96</td>
<td>300</td>
<td>29.88</td>
<td>29.88</td>
<td>advect2_z_ [12]</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>144.79</td>
<td>7.88</td>
<td>300</td>
<td>26.27</td>
<td>31.52</td>
<td>cloud_ [11]</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>152.22</td>
<td>7.43</td>
<td>300</td>
<td>24.77</td>
<td>212.36</td>
<td>radiation_ [3]</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>156.65</td>
<td>4.43</td>
<td>897</td>
<td>4.94</td>
<td>56.61</td>
<td>smlr_ [7]</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>160.77</td>
<td>4.12</td>
<td>300</td>
<td>13.73</td>
<td>24.39</td>
<td>tke_full_ [13]</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>163.97</td>
<td>3.20</td>
<td>300</td>
<td>10.66</td>
<td>10.66</td>
<td>shear_prod_ [15]</td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>166.79</td>
<td>2.82</td>
<td>300</td>
<td>9.40</td>
<td>9.40</td>
<td>rhs_ [16]</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>169.53</td>
<td>2.74</td>
<td>300</td>
<td>9.13</td>
<td>9.13</td>
<td>advect2_xy_ [17]</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>172.00</td>
<td>2.47</td>
<td>300</td>
<td>8.23</td>
<td>15.33</td>
<td>poisson_ [14]</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>174.27</td>
<td>2.27</td>
<td>480000</td>
<td>0.00</td>
<td>0.12</td>
<td>long_wave_ [4]</td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>176.13</td>
<td>1.86</td>
<td>299</td>
<td>6.22</td>
<td>177.45</td>
<td>advect_scalar_ [6]</td>
<td></td>
</tr>
<tr>
<td>0.9</td>
<td>177.94</td>
<td>1.81</td>
<td>300</td>
<td>6.04</td>
<td>6.04</td>
<td>buoy_ [19]</td>
<td></td>
</tr>
</tbody>
</table>
Thanks for your attention!

Questions?
References