Supercomputing and Science

An Introduction to High Performance Computing

Part IV: Dependency Analysis and Stupid Compiler Tricks

Henry Neeman, Director
OU Supercomputing Center for Education & Research

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 is the determination of how different parts of a program interact, and how various parts require other parts in order to operate correctly.

A control dependency governs how different routines or sets of instructions affect each other.

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

Much of this discussion is from reference [1].

Control Dependencies

Every program has a well-defined flow of control.

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 **parallelization**!

Branch Dependency Example

```
y = 7
IF (x /= 0) THEN
    y = 1.0 / x
END IF !! (x /= 0)
```

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

- If the condition evaluates to `.TRUE.`, then `y` is set to `1.0/x`.
- Otherwise, `y` remains `7`.

Loop Dependency Example

```
DO index = 2, length
 a(index) = a(index-1) + b(index)
END DO !! index = 2, length
```

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

This is sometimes called a loop carried dependency.

Why Do We Care?

- Loops are the favorite control structures of High Performance Computing, because compilers know how to optimize them using instruction-level parallelism: superscalar and pipelining give excellent speedup.
- Loop-carried dependencies affect whether a loop can be parallelized, and how much.

Loop or Branch Dependency?

Is this a loop carried dependency or an IF dependency?

```
DO index = 1, length
  IF (x(index) /= 0) THEN
    y(index) = 1.0 / x(index)
  END IF !! (x(index) /= 0)
END DO !! index = 1, length
```

Call Dependency Example

```
x = 5
y = myfunction(7)
z = 22
```

The flow of the program is interrupted by the call to **`myfunction`**, which takes the execution to somewhere else in the program.

I/O Dependency

```
X = a + b
PRINT *, x
Y = c + d
```

Typically, I/O is implemented by implied subroutine calls, so we can think of this as a call dependency.

Reductions

```
sum = 0
DO index = 1, length
  sum = sum + array(index)
END DO !! index = 1, length
```

- Other kinds of reductions: product, `.AND.`, `.OR.`, minimum, maximum, number of occurrences, etc.
- Reductions are so common that hardware and compilers are optimized to handle them.

Data Dependencies

```
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

```
x = a / b
y = x + 2
x = d - e
```

Notice that **x** is assigned two different values, but only one of them is retained after these statements. In this context, the final value of **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

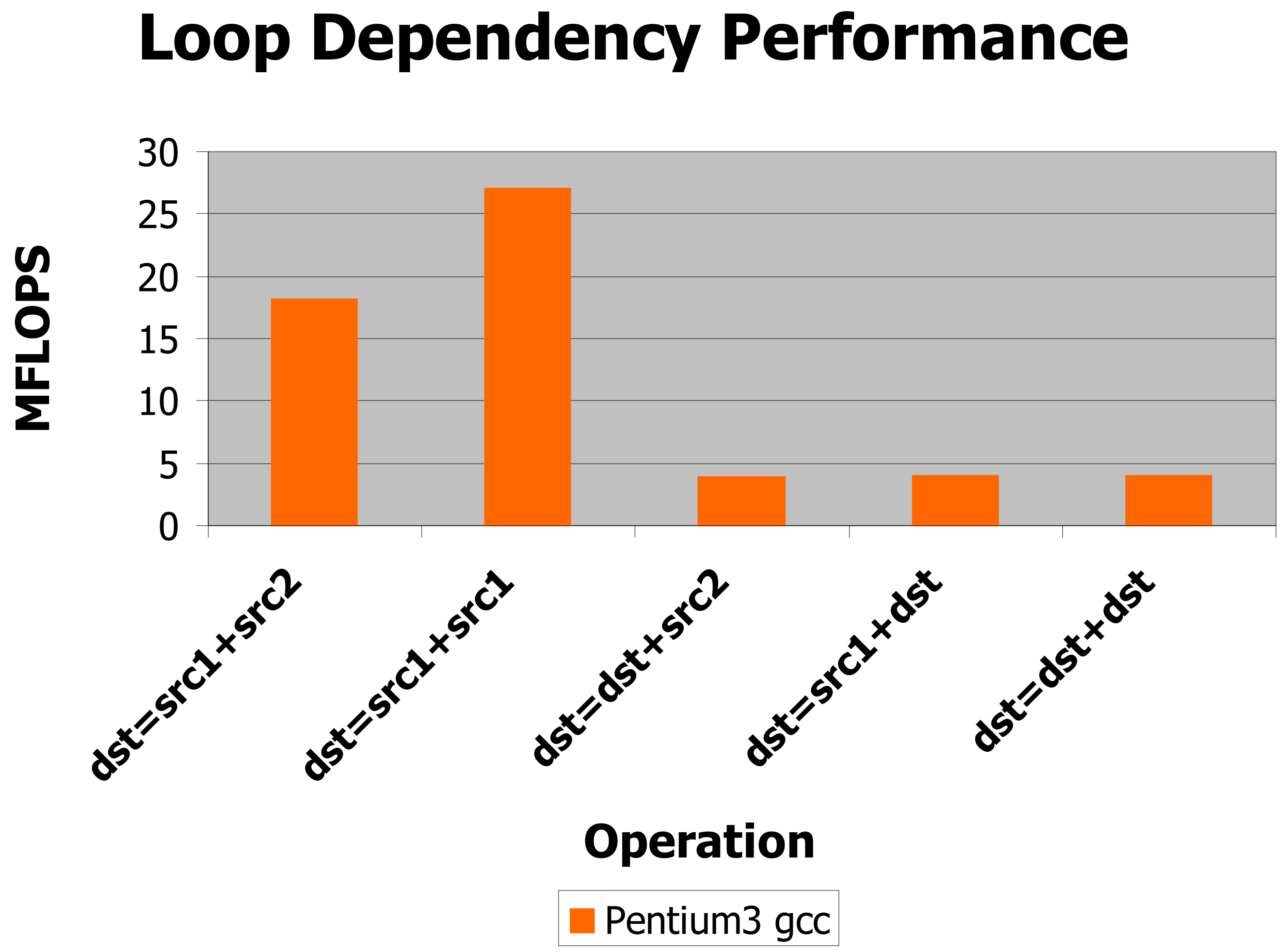
```
 if ((dst == src1) && (dst == src2)) {
   for (index = 1; index < length; index++) {
     dst[index] = dst[index-1]  + dst[index];
   } /* for index */
} /* if ((dst == src1) && (dst == src2)) */
else if (dst == src1) {
   for (index = 1; index < length; index++) {
     dst[index] = dst[index-1]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

Loop Dep Example (cont'd)

```
…
else {
  for (index = 1; index < length; index++) {
    dst[index] = src1[index-1]+src2[index];
  } /* for index */
} /* if (dst == src2)...else */
```

The various versions of the loop either do have loop-carried dependencies or do not.

Loop Dep Performance



Stupid Compiler Tricks

Stupid Compiler Tricks

- Tricks Compilers Play
  - Scalar Optimizations
  - Loop Optimizations
  - Inlining
- Tricks You Can Play with Compilers

Compiler Design

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

- Fortran: 40ish years
- C: 30ish years
- C++: 15ish 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

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

Much of this discussion is from [2].

Copy Propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before	After

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

Before

```
var = 5
PRINT *, var
STOP
PRINT *, var * 2
```

After

```
var = 5
PRINT *, var
STOP
```

Since the last statement never executes, the compiler can eliminate it.

Strength Reduction

Before	After
`x = y ** 2.0`	`x = y * y`
`a = c / 2.0`	`a = c * 0.5`

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.

Common Subexpressions

Before

```
d = c*(a+b)
e = (a+b)*2.0
```

After

```
aplusb = a + b
d = c*aplusb
e = aplusb*2.0
```

The subexpression `(a+b)` occurs in both assignment statements, so there's no point in calculating it twice.

Variable Renaming

Before

```
x = y * z
q = r + x * 2
x = a + b
```

After

```
x0 = y * z
q = r + x0 * 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 and Sinking
- Induction Variable Simplification
- Iteration Peeling
- Loop Interchange
- Unrolling

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

Much of this discussion is from [3].

Hoisting and Sinking

Code that doesn't change inside the loop is called <u>loop invariant</u>.

Before

```
DO i = 1, n                          Hoist
  a(i) = b(i) + c * d
  e = g(n)
END DO !! i = 1, n
   Sink
```

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO !! i = 1, n
e = g(n)
```

Induction Variable Simplifying

Before

```
DO i = 1, n
  k = i*4+m
  …
END DO
```

After

```
k = m
DO i = 1, n
  k = k + 4
  …
END DO
```

One operation can be cheaper than two. On the other hand, this strategy can create a new dependency.

Iteration Peeling

Before

```
DO i = 1, n
  IF ((i == 1) .OR. (i == n)) THEN
    x(i) = y(i)
  ELSE
    x(i) = y(i + 1) + y(i - 1)
  END IF
END DO
```

After

```
x(1) = y(1)
DO i = 1, n
  x(i) = y(i + 1) + y(i - 1)
END DO
x(n) = y(n)
```

Loop Interchange

Before

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

After

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

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.

Unrolling

Before

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

After

```
DO i = 1, n, 4
  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 !! i
```

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 much.

Inlining

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 <u>inlined</u>, 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

- SGI Origin2000

```
f90 –Ofast –ipa
```

- Sun UltraSPARC

```
f90 –fast
```

- CrayJ90

```
f90 –O 3,aggress,pattern,recurrence
```

- Portland Group f90

```
pgf90 -O2 –Mdalign –Mvect=assoc
```

- NAG f95

```
f95 –O4 –Ounsafe –ieee=nonstd
```

What Does the Compiler Do?

Example: NAG f95 compiler

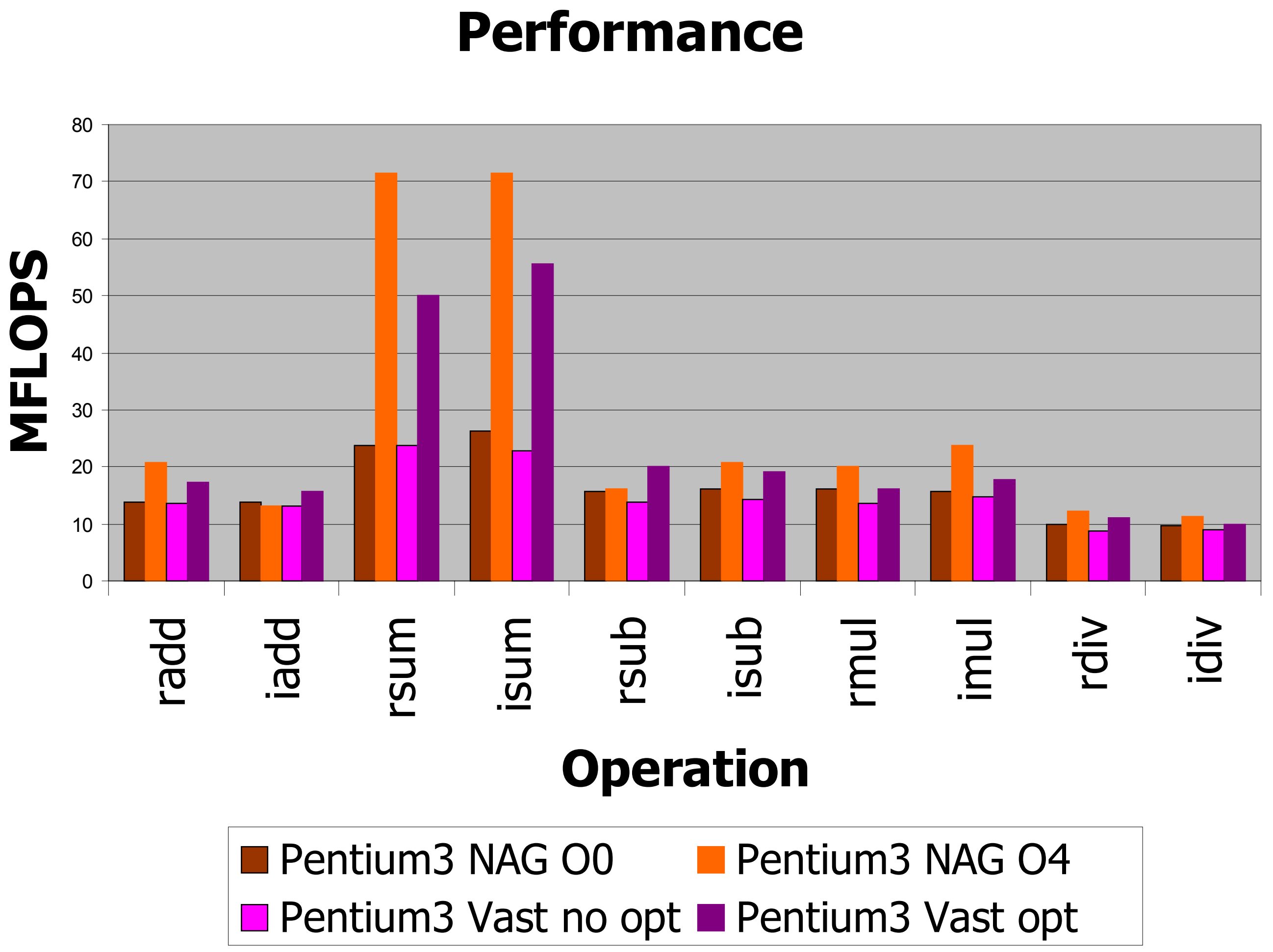
f95 –O<level> source.f90

Possible levels are –O0, -O1, -O2, -O3, -O4:

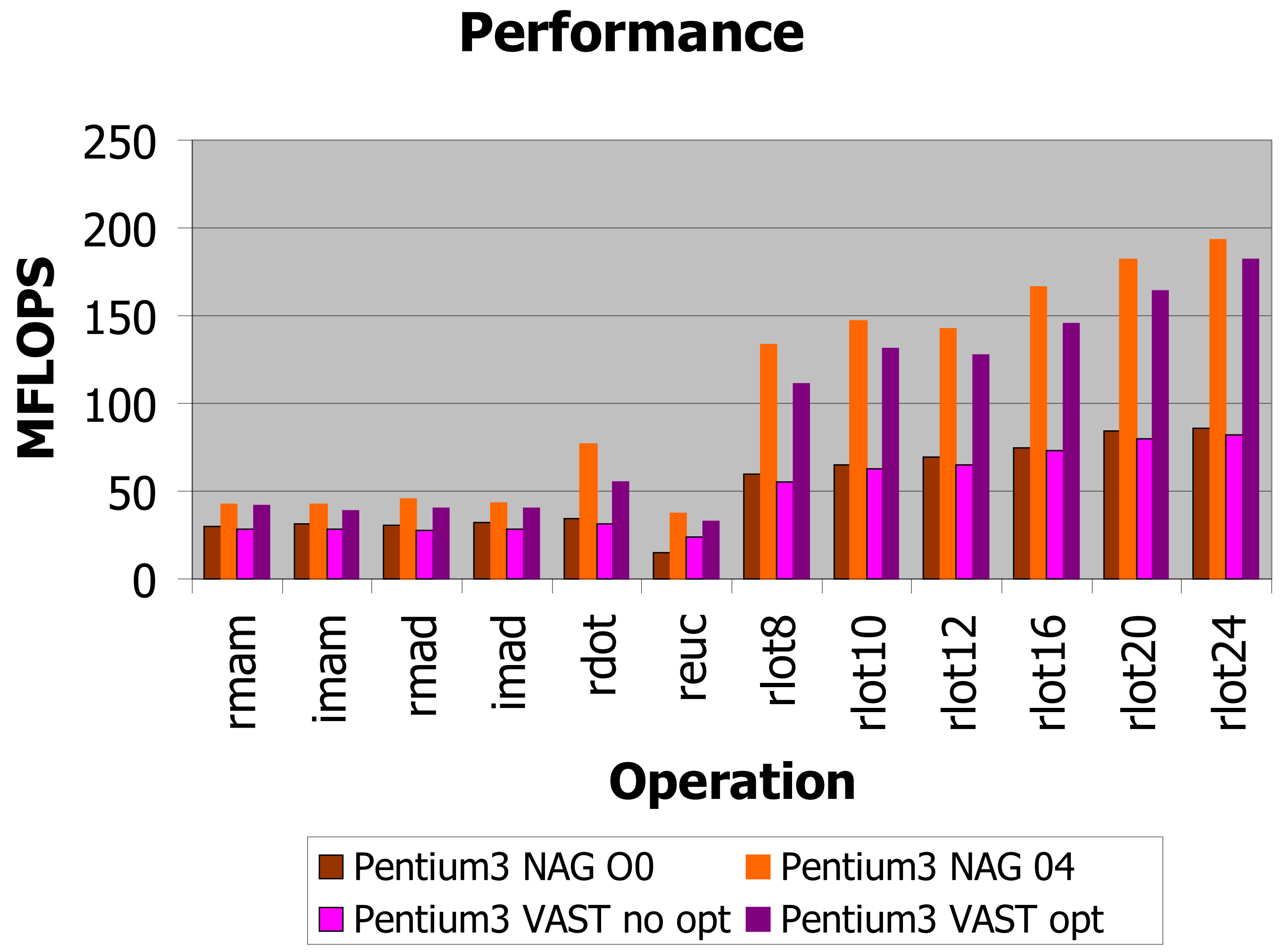
```
-O0      No optimisation. …
-O1      Minimal quick optimisation.
-O2      Normal optimisation.
-O3      Further optimisation.
-O4      Maximal optimisation.[4]
```

The man page is pretty cryptic.

Optimization Performance



More Optimized Performance



Profiling

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.
- 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.

Profiling Example

On an SGI Origin2000:

```
f90 -Ofast -g3 …
```

The **-g3** option tells the compiler to set the executable up to collect profiling information.

ssrun -fpcsampx *executable*

This command generates a file named *executable*.**m240981** (the number is the process number of the run).

Profiling Example (cont'd)

When the run has complete, **`ssrun`** has generated the .m#### file.

Then:

```
prof executable.m240981
```

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

Profiling Result

```
    secs      %     cum.%    samples   function

1323.386  29.1%  29.1%    1323386   complib_sgemm_
1113.284  24.5%  53.7%    1113284   SGEMTV_AXPY
 872.614  19.2%  72.9%     872614   SGEMV_DOT
 338.191   7.4%  80.3%     338191   COMPLIB_SGEMM_HOISTC
 223.375   4.9%  85.3%     223375   _sd2uge
 133.531   2.9%  88.2%     133531   funds
. . .
```

Hardware Timing

In addition to learning about which routines dominate in your program, you might also want to know how the hardware behaves; e.g., you might want to know how often you get a cache miss.

Many supercomputer CPUs have special hardware that measures such events, called <u>event counters</u>.

Hardware Timing Example

On SGI Origin2000:

`perfex -x -a` *executable*

This command produces a list of hardware counts.

Hardware Timing Results

```
Cycles.......................        1350795704000
Decoded instructions..........       1847206417136
Decoded loads..................... 448877703072
Decoded stores...................     76766538224
Grad floating point instructions... 575482548960
Primary data cache misses.........    36090853008
Secondary data cache misses...         5537223904
. . .
```

Next Time

Part V:

Shared Memory Multiprocessing

References

[1] Kevin Dowd and Charles Severance, *High Performance Computing,* 2nd ed. O'Reilly, 1998, p. 173-191.
[2] Ibid, p. 91-99.
[3] Ibid, p. 146-157.
[4] NAG f95 man page.